Encrypted secrets

In this article

Encrypted secrets allow you to store sensitive information in your organization, repository, or repository environments.

GitHub CLI   Web browser

> **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.

About encrypted secrets 🔗

Secrets are variables that you create in an organization, repository, or repository environment. The secrets that you create are available to use in GitHub Actions workflows. GitHub Actions can only read a secret if you explicitly include the secret in a workflow.

For secrets stored at the organization-level, you can use access policies to control which repositories can use organization secrets. Organization-level secrets let you share secrets between multiple repositories, which reduces the need for creating duplicate secrets. Updating an organization secret in one location also ensures that the change takes effect in all repository workflows that use that secret.

For secrets stored at the environment level, you can enable required reviewers to control access to the secrets. A workflow job cannot access environment secrets until approval is granted by required approvers.

Naming your secrets 🔗

The following rules apply to secret names:

- Names can only contain alphanumeric characters (`[a-z]` , `[A-Z]` , `[0-9]`) or

underscores (_). Spaces are not allowed.

- Names must not start with the `GITHUB_` prefix.

- Names must not start with a number.

- Names are not case-sensitive.

- Names must be unique at the level they are created at.

  For example, a secret created at the environment level must have a unique name in that environment, a secret created at the repository level must have a unique name in that repository, and a secret created at the organization level must have a unique name at that level.

  If a secret with the same name exists at multiple levels, the secret at the lowest level takes precedence. For example, if an organization-level secret has the same name as a repository-level secret, then the repository-level secret takes precedence. Similarly, if an organization, repository, and environment all have a secret with the same name, the environment-level secret takes precedence.

To help ensure that GitHub redacts your secret in logs, avoid using structured data as the values of secrets. For example, avoid creating secrets that contain JSON or encoded Git blobs.

Accessing your secrets 🔗

To make a secret available to an action, you must set the secret as an input or environment variable in the workflow file. Review the action's README file to learn about which inputs and environment variables the action expects. For more information, see "Workflow syntax for GitHub Actions."

You can use and read encrypted secrets in a workflow file if you have access to edit the file. For more information, see "Access permissions on GitHub."

> **Warning:** GitHub automatically redacts secrets printed to the log, but you should avoid printing secrets to the log intentionally.

Organization and repository secrets are read when a workflow run is queued, and environment secrets are read when a job referencing the environment starts.

You can also manage secrets using the REST API. For more information, see "Actions."

Limiting credential permissions 🔗

When generating credentials, we recommend that you grant the minimum permissions possible. For example, instead of using personal credentials, use deploy keys or a service account. Consider granting read-only permissions if that's all that is needed, and limit access as much as possible.

When generating a personal access token, select the fewest scopes necessary.

Instead of using a personal access token, consider using a GitHub App, which uses fine-grained permissions and short lived tokens. Unlike a personal access token, a GitHub App is not tied to a user, so the workflow will continue to work even if the user who installed the app leaves your organization. For more information, see "Making authenticated API requests with a GitHub App in a GitHub Actions workflow."

> **Note:** You can use the REST API to manage secrets. For more information, see "Actions."

Creating encrypted secrets for a repository 🔗

To create secrets for a personal account repository, you must be the repository owner. To create secrets for an organization repository, you must have `admin` access.

1. On your GitHub Enterprise Server instance, navigate to the main page of the repository.

2. Under your repository name, click ⚙ **Settings**. If you cannot see the "Settings" tab, select the ··· dropdown menu, then click **Settings**.
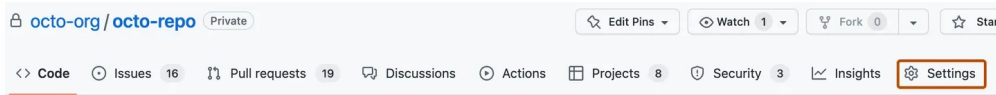
3. In the left sidebar, click **Secrets**.

4. Click **New repository secret**.

5. In the **Name** field, type a name for your secret.

6. In the **Secret** field, enter the value for your secret.

7. Click **Add secret**.

If your repository has environment secrets or can access secrets from the parent organization, then those secrets are also listed on this page.

> To learn more about GitHub CLI, see "[About GitHub CLI](#)."

To add a repository secret, use the `gh secret set` subcommand. Replace `secret-name` with the name of your secret.

```
gh secret set SECRET_NAME
```

The CLI will prompt you to enter a secret value. Alternatively, you can read the value of the secret from a file.
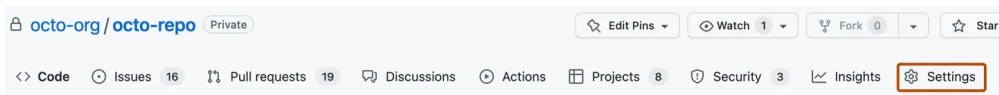
```
gh secret set SECRET_NAME < secret.txt
```

To list all secrets for the repository, use the `gh secret list` subcommand.

Creating encrypted secrets for an environment 🔗

To create secrets for an environment in a personal account repository, you must be the repository owner. To create secrets for an environment in an organization repository, you must have `admin` access. For more information on environments, see "[Using environments for deployment](#)."

1. On your GitHub Enterprise Server instance, navigate to the main page of the repository.

2. Under your repository name, click ⚙ **Settings**. If you cannot see the "Settings" tab, select the ··· dropdown menu, then click **Settings**.

③ In the left sidebar, click **Environments**.

④ Click on the environment that you want to add a secret to.

⑤ Under **Environment secrets**, click **Add secret**.

⑥ Type a name for your secret in the **Name** input box.

⑦ Enter the value for your secret.

⑧ Click **Add secret**.

To add a secret for an environment, use the `gh secret set` subcommand with the `--env` or `-e` flag followed by the environment name.

```
gh secret set --env ENV_NAME SECRET_NAME
```

To list all secrets for an environment, use the `gh secret list` subcommand with the `--env` or `-e` flag followed by the environment name.
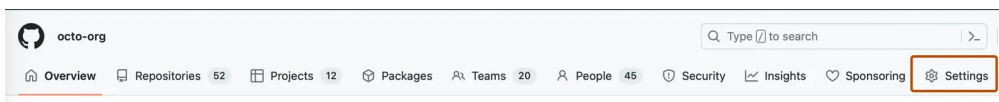
```
gh secret list --env ENV_NAME
```

Creating encrypted secrets for an organization 🔗

When creating a secret or variable in an organization, you can use a policy to limit access by repository. For example, you can grant access to all repositories, or limit access to only private repositories or a specified list of repositories.

To create secrets at the organization level, you must have `admin` access.

① On your GitHub Enterprise Server instance, navigate to the main page of the organization.

② Under your organization name, click ⚙ **Settings**. If you cannot see the "Settings" tab, select the ••• dropdown menu, then click **Settings**.

◯ octo-org   Q Type / to search   >_

⌂ Overview   ⊟ Repositories 52   ⊞ Projects 12   ⊙ Packages   ⊗ Teams 20   ⊗ People 45   ⊙ Security   ⌖ Insights   ♡ Sponsoring   ⚙ Settings

③ In the left sidebar, click **Secrets**.

④ Click **New organization secret**.

⑤ Type a name for your secret in the **Name** input box.

⑥ Enter the **Value** for your secret.

⑦ From the **Repository access** dropdown list, choose an access policy.

⑧ Click **Add secret**.

To add a secret for an organization, use the `gh secret set` subcommand with the `--org` or `-o` flag followed by the organization name.

```
gh secret set --org ORG_NAME SECRET_NAME
```

By default, the secret is only available to private repositories. To specify that the secret should be available to all repositories within the organization, use the `--visibility` or `-v` flag.

```
gh secret set --org ORG_NAME SECRET_NAME --visibility all
```

To specify that the secret should be available to selected repositories within the organization, use the `--repos` or `-r` flag.

```
gh secret set --org ORG_NAME SECRET_NAME --repos REPO-NAME-1, REPO-NAME-2"
```

To list all secrets for an organization, use the `gh secret list` subcommand with the `--org` or `-o` flag followed by the organization name.

```
gh secret list --org ORG_NAME
```

Reviewing access to organization-level secrets 🔗

You can check which access policies are being applied to a secret in your organization.

1. On your GitHub Enterprise Server instance, navigate to the main page of the organization.

2. Under your organization name, click ⚙ **Settings**. If you cannot see the "Settings" tab, select the ··· dropdown menu, then click **Settings**.
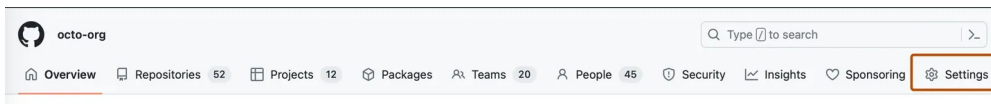
3. In the left sidebar, click **Secrets**.

4. The list of secrets includes any configured permissions and policies. For more details about the configured permissions for each secret, click **Update**.

Using encrypted secrets in a workflow 🔗

To provide an action with a secret as an input or environment variable, you can use the `secrets` context to access secrets you've created in your repository. For more information, see "Contexts" and "Workflow syntax for GitHub Actions."

```
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 }}
```

Secrets cannot be directly referenced in `if:` conditionals. Instead, consider setting secrets as job-level environment variables, then referencing the environment variables to conditionally run steps in the job. For more information, see "Contexts" and `jobs.<job_id>.steps[*].if`.

If a secret has not been set, the return value of an expression referencing the secret (such as `${{ secrets.SuperSecret }}` in the example) will be an empty string.

Avoid passing secrets between processes from the command line, whenever possible. Command-line processes may be visible to other users (using the `ps` command) or captured by security audit events. To help protect secrets, consider using environment variables, `STDIN`, or other mechanisms supported by the target process.

If you must pass secrets within a command line, then enclose them within the proper quoting rules. Secrets often contain special characters that may unintentionally affect your shell. To escape these special characters, use quoting with your environment variables. For example:

Example using Bash 🔗

```
steps:
  - shell: bash
    env:
      SUPER_SECRET: ${{ secrets.SuperSecret }}
    run: |
      example-command "$SUPER_SECRET"
```

Example using PowerShell 🔗

```
steps:
  - shell: pwsh
    env:
      SUPER_SECRET: ${{ secrets.SuperSecret }}
    run: |
      example-command "$env:SUPER_SECRET"
```

Example using Cmd.exe 🔗

```
steps:
  - shell: cmd
    env:
      SUPER_SECRET: ${{ secrets.SuperSecret }}
    run: |
      example-command "%SUPER_SECRET%"
```

Limits for secrets 🔗

You can store up to 1,000 organization secrets, 100 repository secrets, and 100 environment secrets.

A workflow created in a repository can access the following number of secrets:

- All 100 repository secrets.
- If the repository is assigned access to more than 100 organization secrets, the workflow can only use the first 100 organization secrets (sorted alphabetically by secret name).
- All 100 environment secrets.

Secrets are limited to 48 KB in size. To store larger secrets, see the "[Storing large secrets](#)" workaround below.

Storing large secrets 🔗

To use secrets that are larger than 48 KB, you can use a workaround to store encrypted secrets in your repository and save the decryption passphrase as a secret on GitHub. For example, you can use `gpg` to encrypt a file containing your secret locally before checking the encrypted file in to your repository on GitHub. For more information, see the "[gpg manpage](#)."

> **Warning**: Be careful that your secrets do not get printed when your workflow runs. When using this workaround, GitHub does not redact secrets that are printed in logs.

1. Run the following command from your terminal to encrypt the file containing your secret using `gpg` and the AES256 cipher algorithm. In this example, `my_secret.json` is the file containing the secret.

   ```
   gpg --symmetric --cipher-algo AES256 my_secret.json
   ```

2. You will be prompted to enter a passphrase. Remember the passphrase, because you'll need to create a new secret on GitHub that uses the passphrase as the value.

3. Create a new secret that contains the passphrase. For example, create a new secret with the name `LARGE_SECRET_PASSPHRASE` and set the value of the secret to the passphrase you used in the step above.

4. Copy your encrypted file to a path in your repository and commit it. In this example, the encrypted file is `my_secret.json.gpg`.

   > **Warning**: Make sure to copy the encrypted `my_secret.json.gpg` file ending with the `.gpg` file extension, and **not** the unencrypted `my_secret.json` file.

   ```
   git add my_secret.json.gpg
   git commit -m "Add new encrypted secret JSON file"
   ```

5. Create a shell script in your repository to decrypt the secret file. In this example, the script is named `decrypt_secret.sh`.

   ```
   #!/bin/sh
   ```

```
# Decrypt the file
mkdir $HOME/secrets
# --batch to prevent interactive command
# --yes to assume "yes" for questions
gpg --quiet --batch --yes --decrypt --passphrase="$LARGE_SECRET_PASSPHRASE" \
--output $HOME/secrets/my_secret.json my_secret.json.gpg
```

6 Ensure your shell script is executable before checking it in to your repository.

```
chmod +x decrypt_secret.sh
git add decrypt_secret.sh
git commit -m "Add new decryption script"
git push
```

7 In your GitHub Actions workflow, use a `step` to call the shell script and decrypt the secret. To have a copy of your repository in the environment that your workflow runs in, you'll need to use the `actions/checkout` action. Reference your shell script using the `run` command relative to the root of your repository.

```
name: Workflows with large secrets

on: push

jobs:
  my-job:
    name: My Job
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Decrypt large secret
        run: ./decrypt_secret.sh
        env:
          LARGE_SECRET_PASSPHRASE: ${{ secrets.LARGE_SECRET_PASSPHRASE }}
      # This command is just an example to show your secret being printed
      # Ensure you remove any print statements of your secrets. GitHub does
      # not hide secrets that use this workaround.
      - name: Test printing your secret (Remove this step in production)
        run: cat $HOME/secrets/my_secret.json
```

Storing Base64 binary blobs as secrets 🔗

You can use Base64 encoding to store small binary blobs as secrets. You can then reference the secret in your workflow and decode it for use on the runner. For the size limits, see "Encrypted secrets."

> **Note**: Note that Base64 only converts binary to text, and is not a substitute for actual encryption.

1 Use `base64` to encode your file into a Base64 string. For example:

```
$ base64 -i cert.der -o cert.base64
```

2 Create a secret that contains the Base64 string. For example:

```
$ gh secret set CERTIFICATE_BASE64 < cert.base64
✓ Set secret CERTIFICATE_BASE64 for octocat/octorepo
```

3. To access the Base64 string from your runner, pipe the secret to `base64 --decode`. For example:

```yaml
name: Retrieve Base64 secret
on:
  push:
    branches: [ octo-branch ]
jobs:
  decode-secret:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Retrieve the secret and decode it to a file
        env:
          CERTIFICATE_BASE64: ${{ secrets.CERTIFICATE_BASE64 }}
        run: |
          echo $CERTIFICATE_BASE64 | base64 --decode > cert.der
      - name: Show certificate information
        run: |
          openssl x509 -in cert.der -inform DER -text -noout
```

Note: Using another shell might require different commands for decoding the secret to a file. On Windows runners, we recommend using a bash shell with `shell: bash` to use the commands in the `run` step above.

Legal