Installing an Apple certificate on macOS runners for Xcode development

In this article

You can sign Xcode apps within your continuous integration (CI) workflow by installing an Apple code signing certificate on GitHub Actions runners.

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.

Introduction

This guide shows you how to add a step to your continuous integration (CI) workflow that installs an Apple code signing certificate and provisioning profile on GitHub Actions runners. This will allow you to sign your Xcode apps for publishing to the Apple App Store, or distributing it to test groups.

Prerequisites

You should be familiar with YAML and the syntax for GitHub Actions. For more information, see:

- "Learn GitHub Actions"
- "Workflow syntax for GitHub Actions"

You should have an understanding of Xcode app building and signing. For more information, see the Apple developer documentation.

Creating secrets for your certificate and provisioning profile

The signing process involves storing certificates and provisioning profiles, transferring them to the runner, importing them to the runner's keychain, and using them in your build.

To use your certificate and provisioning profile on a runner, we strongly recommend that you use GitHub secrets. For more information on creating secrets and using them in a workflow, see "Using secrets in GitHub Actions."

Create secrets in your repository or organization for the following items:

- Your Apple signing certificate.
  - This is your `p12` certificate file. For more information on exporting your signing certificate from Xcode, see the Xcode documentation.
  - You should convert your certificate to Base64 when saving it as a secret. In this example, the secret is named `BUILD_CERTIFICATE_BASE64`.
  - Use the following command to convert your certificate to Base64 and copy it to your clipboard:

    ```
    base64 -i BUILD_CERTIFICATE.p12 | pbcopy
    ```
- The password for your Apple signing certificate.
  - In this example, the secret is named `P12_PASSWORD`.
- Your Apple provisioning profile.
  - For more information on exporting your provisioning profile from Xcode, see the Xcode documentation.
  - You should convert your provisioning profile to Base64 when saving it as a secret. In this example, the secret is named `BUILD_PROVISION_PROFILE_BASE64`.
  - Use the following command to convert your provisioning profile to Base64 and copy it to your clipboard:

    ```
    base64 -i PROVISIONING_PROFILE.mobileprovision | pbcopy
    ```
- A keychain password.
  - A new keychain will be created on the runner, so the password for the new keychain can be any new random string. In this example, the secret is named `KEYCHAIN_PASSWORD`.

Add a step to your workflow

This example workflow includes a step that imports the Apple certificate and provisioning profile from the GitHub secrets, and installs them on the runner.

YAML

```yaml
name: App build
on: push

jobs:
  build_with_signing:
    runs-on: macos-latest

    steps:
      - name: Checkout repository
        uses: actions/checkout@v4
      - name: Install the Apple certificate and provisioning profile
        env:
          BUILD_CERTIFICATE_BASE64: ${{ secrets.BUILD_CERTIFICATE_BASE64 }}
          P12_PASSWORD: ${{ secrets.P12_PASSWORD }}
          BUILD_PROVISION_PROFILE_BASE64: ${{
secrets.BUILD_PROVISION_PROFILE_BASE64 }}
          KEYCHAIN_PASSWORD: ${{ secrets.KEYCHAIN_PASSWORD }}
        run: |
```

```
          # create variables
          CERTIFICATE_PATH=$RUNNER_TEMP/build_certificate.p12
          PP_PATH=$RUNNER_TEMP/build_pp.mobileprovision
          KEYCHAIN_PATH=$RUNNER_TEMP/app-signing.keychain-db

          # import certificate and provisioning profile from secrets
          echo -n "$BUILD_CERTIFICATE_BASE64" | base64 --decode -o
$CERTIFICATE_PATH
          echo -n "$BUILD_PROVISION_PROFILE_BASE64" | base64 --decode -o $PP_PATH

          # create temporary keychain
          security create-keychain -p "$KEYCHAIN_PASSWORD" $KEYCHAIN_PATH
          security set-keychain-settings -lut 21600 $KEYCHAIN_PATH
          security unlock-keychain -p "$KEYCHAIN_PASSWORD" $KEYCHAIN_PATH

          # import certificate to keychain
          security import $CERTIFICATE_PATH -P "$P12_PASSWORD" -A -t cert -f
pkcs12 -k $KEYCHAIN_PATH
          security list-keychain -d user -s $KEYCHAIN_PATH

          # apply provisioning profile
          mkdir -p ~/Library/MobileDevice/Provisioning\ Profiles
          cp $PP_PATH ~/Library/MobileDevice/Provisioning\ Profiles
      - name: Build app
        ...
```

Note: For iOS build targets, your provisioning profile should have the extension `.mobileprovision`. For macOS build targets, the extension should be `.provisionprofile`. The example workflow above should be updated to reflect your target platform.

Required clean-up on self-hosted runners

GitHub-hosted runners are isolated virtual machines that are automatically destroyed at the end of the job execution. This means that the certificates and provisioning profile used on the runner during the job will be destroyed with the runner when the job is completed.

On self-hosted runners, the `$RUNNER_TEMP` directory is cleaned up at the end of the job execution, but the keychain and provisioning profile might still exist on the runner.

If you use self-hosted runners, you should add a final step to your workflow to help ensure that these sensitive files are deleted at the end of the job. The workflow step shown below is an example of how to do this.

```
- name: Clean up keychain and provisioning profile
  if: ${{ always() }}
  run: |
    security delete-keychain $RUNNER_TEMP/app-signing.keychain-db
    rm ~/Library/MobileDevice/Provisioning\ Profiles/build_pp.mobileprovision
```