Developer Guide

- Getting Started
  - Running a local copy of the client
  - Find issues to work on
  - Testing
    - Running automated unit tests
    - Running automated integration tests
    - Running manual integration tests
    - Running tests in CI
- Code components and layout
  - Plugin-architecture
  - Authenticators
  - Installer
  - Installer Development
  - Writing your own plugin
    - Writing your own plugin snap
- Coding style
- Use certbot.compat.os instead of os
- Mypy type annotations
- Submitting a pull request
- Asking for help
- Building the Certbot and DNS plugin snaps
- Updating the documentation
- Certbot's dependencies
  - Updating dependency versions
  - Choosing dependency versions
- macOS suggestions

Table of Contents

Getting Started

Certbot has the same `system requirements <system_requirements>` when set up for development. While the section below will help you install Certbot and its dependencies, Certbot needs to be run on a UNIX-like OS so if you're using Windows, you'll need to set up a (virtual) machine running an OS such as Linux and continue with these instructions on that UNIX-like OS.

If you're using macOS, it is recommended to first check out the macOS suggestions section before continuing with the installation instructions below.

Running a local copy of the client

Running the client in developer mode from your local tree is a little different than running Certbot as a user. To get set up, clone our git repository by running:

```
git clone https://github.com/certbot/certbot
```

If you're running on a UNIX-like OS, you can run the following commands to install dependencies and set up a virtual environment where you can run Certbot.

Install and configure the OS system dependencies required to run Certbot.

```
# For APT-based distributions (e.g. Debian, Ubuntu ...)
sudo apt update
sudo apt install python3-dev python3-venv libaugeas-dev gcc
# For RPM-based distributions (e.g. Fedora, CentOS ...)
# NB1: old distributions will use yum instead of dnf
# NB2: RHEL-based distributions use python3X instead of python3 (e.g. python38)
sudo dnf install python3 python3-devel augeas-devel gcc
# For macOS installations with Homebrew already installed and configured
# NB: CFLAGS are needed to compile and link to Augeas installed through
#     Homebrew and some of our developer scripts expect GNU coreutils be first in
#     your PATH. The commands below set this up for bash and zsh, but your
#     instructions may be slightly different if you use an alternate shell.
brew install augeas coreutils gnu-sed
BREW_PREFIX=$(brew --prefix)
RC_LINES="export CFLAGS=\"\$CFLAGS -I$BREW_PREFIX/include -L$BREW_PREFIX/lib\"\n"
RC_LINES+="export PATH=\"$BREW_PREFIX/opt/coreutils/libexec/gnubin:"
RC_LINES+="$BREW_PREFIX/opt/gnu-sed/libexec/gnubin:\$PATH\"\n"
printf "$RC_LINES" >> ~/.bashrc  # for bash
```

```
printf "$RC_LINES" >> ~/.zshrc  # for zsh
```

> [!NOTE] If you have trouble creating the virtual environment below, you may need to install additional dependencies. See the cryptography project's site for more information.

Set up the Python virtual environment that will host your Certbot local instance.

```
cd certbot
python3 tools/venv.py
```

> [!NOTE] You may need to repeat this when Certbot's dependencies change or when a new plugin is introduced.

You can now run the copy of Certbot from git either by executing `venv/bin/certbot`, or by activating the virtual environment. You can do the latter by running:

```
source venv/bin/activate
```

After running this command, `certbot` and development tools like `ipdb3`, `ipython`, `pytest`, and `tox` are available in the shell where you ran the command. These tools are installed in the virtual environment and are kept separate from your global Python installation. This works by setting environment variables so the right executables are found and Python can pull in the versions of various packages needed by Certbot. More information can be found in the virtualenv docs.

Find issues to work on

You can find the open issues in the github issue tracker. If you're starting work on something, post a comment to let others know and seek feedback on your plan where appropriate.

Once you've got a working branch, you can open a pull request. All changes in your pull request must have thorough unit test coverage, pass our tests, and be compliant with the `coding style <coding-style>`.

Testing

You can test your code in several ways:

- running the automated unit tests,
- running the automated integration tests
- running an *ad hoc* manual integration test

Running automated unit tests

To run all unittests, mypy, and lint:

```
tox
```

If you're working on a specific test and would like to run just that one:

```
pytest acme/src/acme/_internal/tests/messages_test.py # Use the test file you're working on
```

To run a specific test case within a file:

```
pytest acme/src/acme/_internal/tests/messages_test.py -k test_to_partial_json
```

For debugging, we recommend putting `import ipdb; ipdb.set_trace()` statements inside the source code, which will require adding the -s flag to pytest invocations.

> [!WARNING] The full test suite may attempt to modify your system's Apache config if your user has sudo permissions, so it should not be run on a production Apache server.

Running automated integration tests

Generally it is sufficient to open a pull request and let GitHub run integration tests for you. However, you may want to run them locally before submitting your pull request. You need Docker installed and working.

The tox environment integration will setup Pebble, the Let's Encrypt ACME CA server for integration testing, then launch the Certbot integration tests.

With a user allowed to access your local Docker daemon, run:

```
tox run -e integration
```

Tests will be run using pytest. A test report and a code coverage report will be displayed at the end of the integration tests execution.

Running manual integration tests

You can also manually execute Certbot against a local instance of the Pebble ACME server. This is useful to verify that the modifications done to the code makes Certbot behave as expected.

To do so you need:

- Docker installed, and a user with access to the Docker client,
- an available local copy of Certbot.

The virtual environment set up with python3 tools/venv.py contains two CLI tools that can be used once the virtual environment is activated:

```
run_acme_server
```

- Starts a local instance of Pebble and runs in the foreground printing its logs.
- Press CTRL+C to stop this instance.
- This instance is configured to validate challenges against certbot executed locally.

> [!NOTE] Some options are available to tweak the local ACME server. You can execute `run_acme_server --help` to see the inline help of the `run_acme_server` tool.

```
certbot_test [ARGS...]
```

- Execute certbot with the provided arguments and other arguments useful for testing purposes, such as: verbose output, full tracebacks in case Certbot crashes, *etc.*
- Execution is preconfigured to interact with the Pebble CA started with `run_acme_server`.
- Any arguments can be passed as they would be to Certbot (eg. `certbot_test certonly -d test.example.com`).

Here is a typical workflow to verify that Certbot successfully issued a certificate using an HTTP-01 challenge on a machine with Python 3:

```
python3 tools/venv.py
source venv/bin/activate
run_acme_server &
certbot_test certonly --standalone -d test.example.com
# To stop Pebble, launch `fg` to get back the background job, then press CTRL+C
```

Running tests in CI

Certbot uses GitHub Actions to run continuous integration tests. If you are using our GitHub Actions setup, a branch whose name starts with test- will run all tests on that branch.

Code components and layout

The following components of the Certbot repository are distributed to users:

acme
contains all protocol specific code

certbot
main client code

certbot-apache and certbot-nginx
client code to configure specific web servers

certbot-dns-*
client code to configure DNS providers

Plugin-architecture

Certbot has a plugin architecture to facilitate support for different webservers, other TLS servers, and operating systems. The interfaces available for plugins to implement are defined in interfaces.py and plugins/common.py.

The main two plugin interfaces are ~certbot.interfaces.Authenticator, which implements various ways of proving domain control to a certificate authority, and ~certbot.interfaces.Installer, which configures a server to use a certificate once it is issued. Some plugins, like the built-in Apache and Nginx plugins, implement both interfaces and perform both tasks. Others, like the built-in Standalone authenticator, implement just one interface.

Authenticators

Authenticators are plugins that prove control of a domain name by solving a challenge provided by the ACME server. ACME currently defines two types of challenges: HTTP and DNS, represented by classes in acme.challenges. An authenticator plugin should implement support for at least one challenge type.

An Authenticator indicates which challenges it supports by implementing get_chall_pref(identifier) to return a sorted list of challenge types in preference order.

An Authenticator must also implement perform(achalls), which "performs" a list of challenges by, for instance, provisioning a file on an HTTP server, or setting a TXT record in DNS. Once all challenges have succeeded or failed, Certbot will call the plugin's cleanup(achalls) method to remove any files or DNS records that were needed only during authentication.

Installer

Installers plugins exist to actually setup the certificate in a server, possibly tweak the security configuration to make it more correct and secure (Fix some mixed content problems, turn on HSTS, redirect to HTTPS, etc). Installer plugins tell the main client about their abilities to do the latter via the `~.Installer.supported_enhancements` call. We currently have two Installers in the tree, the ~~.ApacheConfigurator~~. and the .NginxConfigurator. External projects have made some progress toward support for IIS, Icecast and Plesk.

Installers and Authenticators will oftentimes be the same class/object (because for instance both tasks can be performed by a webserver like nginx) though this is not always the case (the standalone plugin is an authenticator that listens on port 80, but it cannot install certificates; a postfix plugin would be an installer but not an authenticator).

Installers and Authenticators are kept separate because it should be possible to use the ~.StandaloneAuthenticator (it sets up its own Python server to perform challenges) with a program that cannot solve challenges itself (Such as MTA installers).

Installer Development

There are a few existing classes that may be beneficial while developing a new ~~certbot.interfaces.Installer~~. Installers aimed to reconfigure UNIX servers may use Augeas for configuration parsing and can inherit from .AugeasConfigurator class to handle much of the interface. Installers that are unable to use Augeas may still find the ~.Reverter class helpful in handling configuration checkpoints and rollback.

Writing your own plugin

> [!NOTE] The Certbot team is not currently accepting any new plugins because we want to rethink our approach to the challenge and resolve some issues like #6464, #6503, and #6504 first.
>
> In the meantime, you're welcome to release it as a third-party plugin. See certbot-dns-ispconfig for one example of that.

Certbot client supports dynamic discovery of plugins through the importlib.metadata entry points using the certbot.plugins group. This way you can, for example, create a custom implementation of ~certbot.interfaces.Authenticator or the ~certbot.interfaces.Installer without having to merge it with the core upstream source code. An example is provided in `examples/plugins/` directory.

While developing, you can install your plugin into a Certbot development virtualenv like this:

```
. venv/bin/activate
pip install -e examples/plugins/
certbot_test plugins
```

Your plugin should show up in the output of the last command. If not, it was not installed properly.

Once you've finished your plugin and published it, you can have your users install it system-wide with pip install. Note that this will only work for users who have Certbot installed from OS packages or via pip.

Writing your own plugin snap

If you'd like your plugin to be used alongside the Certbot snap, you will also have to publish your plugin as a snap. Plugin snaps are regular confined snaps, but normally do not provide any "apps" themselves. Plugin snaps export loadable Python modules to the Certbot snap.

When the Certbot snap runs, it will use its version of Python and prefer Python modules contained in its own snap over modules contained in external snaps. This means that your snap doesn't have to contain things like an extra copy of Python, Certbot, or their dependencies, but also that if you need a different version of a dependency than is already installed in the Certbot snap, the Certbot snap will have to be updated.

Certbot plugin snaps expose their Python modules to the Certbot snap via a snap content interface where `certbot-1` is the value for the `content` attribute. The Certbot snap only uses this to find the names of connected plugin snaps and it expects to find the Python modules to be loaded under `lib/python3.12/site-packages/` in the plugin snap. This location is the default when using the `core24` base snap and the python snapcraft plugin.

The Certbot snap also provides a separate content interface which you can use to get metadata about the Certbot snap using the `content` identifier `metadata-1`.

The script used to generate the snapcraft.yaml files for our own externally snapped plugins can be found at https://github.com/certbot/certbot/blob/main/tools/snap/generate_dnsplugins_snapcraft.sh .

For more information on building externally snapped plugins, see the section on `Building snaps`.

Once you have created your own snap, if you have the snap file locally, it can be installed for use with Certbot by running:

```
snap install --classic certbot
snap set certbot trust-plugin-with-root=ok
snap install --dangerous your-snap-filename.snap
sudo snap connect certbot:plugin your-snap-name
sudo /snap/bin/certbot plugins
```

If everything worked, the last command should list your plugin in the list of plugins found by Certbot. Once your snap is published to the snap store, it will be installable through the name of the snap on the snap store without the `--dangerous` flag. If you are also using Certbot's metadata interface, you can run `sudo snap connect your-snap-name:your-plug-name-for-metadata certbot:certbot-metadata` to connect your snap to it.

Coding style

Please:

1. **Be consistent with the rest of the code**.
2. Read PEP 8 - Style Guide for Python Code.
3. Follow the Google Python Style Guide, with the exception that we use Sphinx-style documentation:

   ```
   def foo(arg):
       """Short description.

       :param int arg: Some number.

       :returns: Argument
       :rtype: int

       """
       return arg
   ```

4. Remember to use `pylint`.
5. You may consider installing a plugin for editorconfig in your editor to prevent some linting warnings.
6. Please avoid unittest.assertTrue or unittest.assertFalse when possible, and use assertEqual or more specific assert. They give better messages when it's failing, and are

generally more correct.

Use `certbot.compat.os` instead of `os`

Python's standard library `os` module lacks full support for several Windows security features about file permissions (eg. DACLs). However several files handled by Certbot (eg. private keys) need strongly restricted access on both Linux and Windows.

To help with this, the `certbot.compat.os` module wraps the standard `os` module, and forbids usage of methods that lack support for these Windows security features.

As a developer, when working on Certbot or its plugins, you must use `certbot.compat.os` in every place you would need `os` (eg. `from certbot.compat import os` instead of `import os`). Otherwise the tests will fail when your PR is submitted.

Mypy type annotations

Certbot uses the mypy static type checker. Python 3 natively supports official type annotations, which can then be tested for consistency using mypy. Mypy does some type checks even without type annotations; we can find bugs in Certbot even without a fully annotated codebase.

Zulip wrote a great guide to using mypy. It's useful, but you don't have to read the whole thing to start contributing to Certbot.

To run mypy on Certbot, use `tox run -e mypy` on a machine that has Python 3 installed.

Also note that OpenSSL, which we rely on, has type definitions for crypto but not SSL. We use both. Those imports should look like this:

```
from OpenSSL import crypto
from OpenSSL import SSL
```

Submitting a pull request

Steps:

0. We recommend you talk with us in a GitHub issue or `Mattermost <ask for help>` before writing a pull request to ensure the changes you're making is something we have the time and interest to review.
1. Write your code! When doing this, you should add `mypy type annotations <type annotations>` for any functions you add or modify. You can check that you've done this correctly by running `tox run -e mypy` on a machine that has Python 3 installed.
2. Make sure your environment is set up properly and that you're in your virtualenv. You can do this by following the instructions in the `Getting Started <getting_started>` section.
3. Run `tox run -e lint` to check for pylint errors. Fix any errors.
4. Run `tox --skip-missing-interpreters` to run all the tests we recommend developers run locally. The `--skip-missing-interpreters` argument ignores missing versions of Python needed for running the tests. Fix any errors.
5. If any documentation should be added or updated as part of the changes you have made, please include the documentation changes in your PR.
6. Submit the PR. Once your PR is open, please do not force push to the branch containing your pull request to squash or amend commits. We use squash merges on PRs and rewriting commits makes changes harder to track between reviews.
7. Did your tests pass on GitHub Actions? If they didn't, fix any errors.

Asking for help

If you have any questions while working on a Certbot issue, don't hesitate to ask for help! You can do this in the Certbot channel in EFF's Mattermost instance for its open source projects as described below.

You can get involved with several of EFF's software projects such as Certbot at the EFF Open Source Contributor Chat Platform. By signing up for the EFF Open Source Contributor Chat Platform, you consent to share your personal information with the Electronic Frontier Foundation, which is the operator and data controller for this platform. The channels will be available both to EFF, and to other users of EFFOSCCP, who may use or disclose information in these channels outside of EFFOSCCP. EFF will use your information, according to the Privacy Policy, to further the mission of EFF, including hosting and moderating the discussions on this platform.

Use of EFFOSCCP is subject to the EFF Code of Conduct. When investigating an alleged Code of Conduct violation, EFF may review discussion channels or direct messages.

Building the Certbot and DNS plugin snaps

Instructions for how to manually build and run the Certbot snap and the externally snapped DNS plugins that the Certbot project supplies are located in the README file at https://github.com/certbot/certbot/tree/main/tools/snap.

Updating the documentation

Many of the packages in the Certbot repository have documentation in a `docs/` directory. This directory is located under the top level directory for the package. For instance, Certbot's documentation is under `certbot/docs`.

To build the documentation of a package, make sure you have followed the instructions to set up a local copy of Certbot including activating the virtual environment. After that, `cd` to the docs directory you want to build and run the command:

```
make clean html
```

This would generate the HTML documentation in `_build/html` in your current `docs/` directory.

Certbot's dependencies

We attempt to pin all of Certbot's dependencies whenever we can for reliability and consistency. Some of the places we have Certbot's dependencies pinned include our snaps, Docker images, CI, and our development environments.

In most cases, the file where dependency versions are specified is `tools/requirements.txt`. The one exception to this is our "oldest" tests where `tools/oldest_constraints.txt` is used instead. The purpose of the "oldest" tests is to ensure Certbot continues to work with the oldest versions of our dependencies which we claim to support. The oldest versions of the dependencies we support should also be declared in our setup.py files to communicate this information to our users.

The choices of whether Certbot's dependencies are pinned and what file is used if they are should be automatically handled for you most of the time by Certbot's tooling. The way it works though is `tools/pip_install.py` (which many of our other tools build on) checks for the presence of environment variables. If `CERTBOT_OLDEST` is set to 1, `tools/oldest_constraints.txt` will be used as constraints for `pip`, otherwise, `tools/requirements.txt` is used as constraints.

Updating dependency versions

`tools/requirements.txt` and `tools/oldest_constraints.txt` can be updated using `tools/pinning/current/repin.sh` and `tools/pinning/oldest/repin.sh` respectively. This works by using `poetry` to generate pinnings based on a Poetry project defined by the `pyproject.toml` file in the same directory as the script. In many cases, you can just run the script to generate updated dependencies, however, if you need to pin back packages or unpin packages that were previously restricted to an older version, you will need to modify the `pyproject.toml` file. The syntax used by this file is described at https://python-poetry.org/docs/pyproject/ and how dependencies are specified in this file is further described at https://python-poetry.org/docs/dependency-specification/.

If you want to learn more about the design used here, see `tools/pinning/DESIGN.md` in the Certbot repo.

Choosing dependency versions

When choosing dependency versions, we should choose whatever minimum versions simplify development of Certbot and our own distribution methods such as snaps, pip, and docker. Since these approaches have full access to PyPI, it's OK if the required packages declared in `setup.py` are quite new.

If this approach to development creates significant trouble for some of our users, we can revisit this decision and weigh their trouble against the difficulties involved in maintaining support for a wider range of package versions. When doing this, we should also be sure to consider the feasibility of users getting access to these newer packages on their system rather than changing our own approach here. Their OS distribution may be able to package it, especially in an alternate repository and/or for a different version of Python to help avoid conflicts with other packages on their system.

macOS suggestions

If you're developing on macOS, before `setting up your Certbot development environment <local copy>`, it is recommended you perform the following steps. None of this is required, but it is the approach used by all/most of the current Certbot developers on macOS as of writing this:

0. Install Homebrew. It is the most popular package manager on macOS by a wide margin and works well enough.
1. Install pyenv, ideally through Homebrew by running `brew install pyenv`. Using Homebrew's Python for Certbot development is annoying because it regularly updates and every time it does it breaks your virtual environments. Using Python from `pyenv` avoids this problem and gives you easy access to all versions of Python.

2. If you're using `pyenv`, make sure you've set up your shell for it by following instructions like https://github.com/pyenv/pyenv?tab=readme-ov-file#set-up-your-shell-environment-for-pyenv.

□□□ #2
□ □□□ □□□ 2026-10-31 13:02:33 UTC
□ □□□ □□□ 2026-10-31 13:02:35 UTC