Multiple Vulnerabilities in Flower and Downstream Attacks on Airflow

May 26, 2022

This post discloses two vulnerabilities in Flower, a popular open-source web application deployed along with Celery (a task queue for Python):

- Lack of CSRF or SSRF protections in Flower, allowing remote API invocations on internally-hosted instances
- An OAuth authentication bypass allowing anyone to login and access all Flower functionality

These issues are tracked under CVE-2022-30034.

I created a PR which fixes the OAuth bypass and disables the API unless authentication is configured. The maintainer did not respond to attempts at contact so it is unclear if this will be merged: https://github.com/mher/flower/pull/1216.

The vulnerabilities themselves are moderately interesting, but a more interesting question to ask is "what could an attacker *do* with Flower?":

- Invoke arbitrary tasks registered with Celery (basically, RPC; but the tasks would be highly dependent on application context)
- Apache Airflow is probably the most popular software which uses Celery, and registers a Celery task in a default configuration. Through Flower, an attacker could invoke `airflow tasks run` with arbitrary parameters, which could (dependent on configuration or other vulnerabilities) be used to achieve RCE

Background

Flower is a web app which allows monitoring and managing Celery, which is a task queue for Python.

- Flower has more than 5000 stars on Github and is included in the default deployments of other extremely popular software such as Apache Airflow.
- Flower is (or was) included but not enabled by default in some managed Airflow deployments such as Google Cloud Composer.

- Shodan shows more than 2000 publicly-exposed instances, although most instances appear to require authentication with basic authentication, and thus would not be vulnerable.
- There are likely many more instances which are only accessible on internal/corporate networks.

Lack of CSRF and SSRF Protections

Flower lacks CSRF protection, meaning that if a user with access to a Flower instance visits a malicious website, that site could run JavaScript which calls Flower's APIs. This would primarily be used to exploit other vulnerabilities, such as through the arbitrary task invocation issue. The lack of CSRF protections apply to all web routes, APIs, and the `api/task/events` websocket endpoint. In particular, websocket connections can be used to solidly confirm the presence of the vulnerable listener on internal networks and also leak some minor information.

Lack of authentication by default also means that SSRF attacks would be straightforward on any instance not configured for authentication.

Flower also has a strange (i.e. broken) CORS policy, but it doesn't appear to lead to any additional impact. *Note:* a recent PR opened the CORS headers up with a wildcard, which would increase the exploitability of CSRF.

Overall, I think the likelihood of CSRF exploitation through Flower is relatively low, so my PR didn't attempt to add CSRF protections, because doing so would likely require breaking all existing Flower API clients (e.g. requiring a custom HTTP header). Rather, the PR disables the API if authentication is disabled.

OAuth Bypass Vulnerability

Flower supports OAuth login via Google, Github, Okta, and Gitlab. When configuring OAuth according to the documentation, Flower requires the user to provide a regular expression which is checked against the user email that Flower receives after the user logs in:

```
flower --auth_provider=flower.views.auth.GitLabLoginHandler --auth=.*@exam
```

The above command line is intended to configure Flower to use Gitlab OAuth and only allow users whose emails are registered with `example.com`. However, the regular expression check *lacks regex anchors*, thus allowing anyone to authenticate as long as they can sign up to the OAuth identity provider with a user-controlled email.

Example scenarios:

- For `.*@example.com`, a valid user might be `alice@example.com`, but `attacker@example.com.attacker.com` can also login.
- Specifying `me@gmail.com` would only be intended to allow one user to login, but `me@gmail.com.attacker.com` also works.
- Because the regular expression is provided on the command line, the period may not be correctly escaped. For a literal period provided in a shell command, a double backslash is generally needed to get a literal backslash in the argument. The documentation examples use either zero or one backslashes. As a result, `.*@corp.example.com` or `.*@corp\.example\.com` could be bypassed with `attacker@corpZexample.com` even if anchors were applied.

Reproduction

I demonstrated the vulnerability in a local deployment of Flower with the following configuration:

1. Create a public Gitlab user `user1@tannerprynn.com` (the intended user of the application)
    - Configure an application under this user to login to Flower
2. Create a second public Gitlab user `user1@tannerprynn.com.pry.nz`
3. Flower configuration:
    - Command-line options `--auth_provider=flower.views.auth.GitLabLoginHandler --auth=.*@tannerprynn.com`
    - Environment variables `FLOWER_OAUTH2_KEY`, `FLOWER_OAUTH2_SECRET`, `FLOWER_OAUTH2_REDIRECT_URI`

Recommendations for Flower

1. Disable the API unless authentication is configured for Flower
2. Implement CSRF protections (primarily relevant to the API); the most straightforward would be to require a custom non-CORS-exempt header for all API requests
3. Fixing the OAuth bypass will likely require either a breaking change for users, or using custom logic to match the user-provided string without interpreting a single period as a wildcard character

Post-Exploitation

With the ability to send requests to Flower, an attacker would primarily use its APIs - in particular, `POST /api/task/apply/{taskname}` which allows invoking any Celery task. Assuming the attacker is not operating through blind CSRF, they can use `GET /api/task/types` and other APIs to list which tasks can be called and the parameters, and attempt to exploit those specific tasks. Given the expectation that these calls would only come from internal, trusted systems, it is likely that any custom-developed tasks have vulnerabilities or cause some dangerous actions. However, those vulns would be highly application/context-specific.

I was interested in one specific deployment of Flower, which is with Airflow, a very popular open-source app. Flower is deployed without authentication in the default Airflow configuration (when deployed with the recommended docker-compose configuration and Helm chart). Airflow registers a single task `airflow.executors.celery_executor.execute_command` which allows invoking an Airflow task via the equivalent of a command-line invocation of `airflow tasks run`.

```
POST /api/task/apply/airflow.executors.celery_executor.execute_command HTT
Host: flower:5555
Content-Type: text/plain
Content-Length: 84

{"args":[["airflow", "tasks", "run", "example_bash_operator", "runme_1", "
```

Initially, I was relatively confident that this would lead directly to RCE, but it doesn't appear straightforward:

- Although the arguments appear to specify a command-line invocation, they don't actually pass through a shell at any point. Celery passes these arguments through a message queue (e.g. Redis) and they invoke the `execute_command` Python method after being picked up by an Airflow worker, which then parses the arguments using its own command-line parser.
- The first three arguments (`"airflow", "tasks", "run"`) are validated and cannot be changed to invoke other `airflow` CLI commands.
- The last argument is `execution_date_or_run_id` which is used unsafely in Airflow's `example_bash_operator`, but this argument appears to be validated.

That's not to say that this is safe. An attacker can still specify arbitrary arguments after `airflow tasks run`, which would include parameters such as `--cfg-path` and `--subdir`. If combined with the ability to write a file somewhere on the filesystem, either of these configuration options allows picking up and running arbitrary python code. Without that ability, the impact would be limited to calling the DAGs/tasks that are already set up on the system.

Recommendations for Airflow

1. If possible, strip all parameters from the arguments provided to this invocation path so that an attacker does not have the ability to change the configuration in a way which might allow unexpected code execution
2. Although `run_id` seems to be validated, preventing injection in `example_bash_operator`, that doesn't seem to be a documented expectation in general, so you should handle it safely in your example code
3. If Flower isn't a necessary component of Airflow, consider removing it from the default docker-compose/Helm setup
4. Alternatively, set up docker-compose and Helm to use basic auth with randomly-generated passwords for Flower (even a weak static password would actually help limit CSRF/SSRF attacks)

From discussions with the Airflow maintainer, Flower is seen as a development-only tool that should not be used in any production deployment. However, given that Flower itself does not appear to be actively maintained, the Airflow maintainer decided that they would disable Flower by default in their docker-compose and Helm chart (starting in version 2.3.1).

Disclosure Timeline

- 05 April 2022: Initial contacts to Apache Security team
- 06 April 2022: Partial disclosure to Airflow maintainer of impact and recommendations for Airflow
- 06 April 2022: Initial attempt to contact the Flower maintainer (mher)
- 06-21 April 2022: Follow up emails to the Airflow maintainer
- 12-25 April 2022: Discussion with Airflow maintainer
- 23-26 April 2022: Attempts to reach a maintainer for Flower via the associated Celery project's team members, but no team member was able to give me a working contact
- 26 April 2022: Opened an issue on the public Flower repo asking for a security contact
- 03 May 2022: Followed up on the public issue to state that disclosure would occur in 14 days (17 May) due to lack of response
- 16 May 2022: Airflow maintainer requested a short delay to allow a new release of Airflow which disables Flower by default
- 25 May 2022: Airflow updated to 2.3.1
- 26 May 2022: Public disclosure

Tanner Prynn

Tanner Prynn
blog@tannerprynn.com

tprynn
tannerprynn

Blog posts by @tannerprynn