Developing CTFd Plugins

Introduction

CTFd features a plugin interface allowing for the modification of CTFd behavior without modifying the core CTFd code. This has a number of benefits over forking and modifying CTFd:

- Your modifications and plugins can be shared more easily
- CTFd can be updated without losing any custom behavior

The CTFd developers will do their best to not introduce breaking changes but keep in mind that the plugin interface is still under development and could change.

TIP

Official CTFd plugins are available at https://ctfd.io/store. Contact us regarding custom plugins and special projects.

TIP

Community plugins are available at https://github.com/CTFd/plugins.

Architecture

CTFd plugins are implemented as Python modules with some CTFd specific files.

```
CTFd
└── plugins
    └── CTFd-plugin
        ├── README.md          # README file
```

```
        ├── __init__.py        # Main code file loaded by CTFd
        ├── requirements.txt   # Any requirements that need to be installed
        └── config.json        # Plugin configuration file
```

Effectively CTFd will look at every folder in the `CTFd/plugins` folder for the `load()` function.

If the `load()` function is found, CTFd will call that function with itself (as a Flask app) as a parameter (i.e. `load(app)`). This is done after CTFd has added all of its internal routes but before CTFd has fully instantiated itself. This allows plugins to modify many aspects of CTFd without having to modify CTFd itself.

config.json

`config.json` exists to give plugin developers a way to define attributes about their plugin. It's primary usage within CTFd is to give users a way to access a Configuration or Settings page for the plugin.

This is an example `config.json` file:

```
{
    "name": "CTFd Plugin",
    "route": "/admin/custom_plugin_route"
}
```

This is ultimately rendered to the user with the following template snippet:

```
{% if plugins %}
<li>
    <a href="#" class="dropdown-toggle" data-toggle="dropdown" role="button" aria-haspo
    <ul class="dropdown-menu">
            {% for plugin in plugins %}
                    <li><a href="{{ request.script_root }}{{ plugin.route }}">{{ plugin
            {% endfor %}
    </ul>
</li>
{% endif %}
```

config.html

In the past CTFd used a static file known as `config.html` which existed to give plugin developers a page that is loaded by the CTFd admin panel. This has been superceded in favor of config.json but is still supported for backwards compatability.

The `config.html` file for a plugin is available by CTFd admins at `/admin/plugins/<plugin-folder-name>`. Thus if `config.html` is stored in `CTFd-S3-plugin`, it would be available at `/admin/plugins/CTFd-S3-plugin`.

`config.html` is loaded as a Jinja template so it has access to all of the same functions and abilities that CTFd exposes to Jinja. Jinja templates are technically also capable of running arbitrary Python code but this is ancillary.

Adding New Routes

Adding new routes in CTFd is effectively just an exercise in writing new Flask routes. Since the plugin itself is passed the entire app, the plugin can leverage the `app.route` decorator to add new routes.

A simple example is as follows:

```
from flask import render_template

def load(app):
    @app.route('/faq', methods=['GET'])
    def view_faq():
        return render_template('page.html', content="<h1>FAQ Page</h1>")
```

Modifying Existing Routes

It is slightly more complicated to override existing routes in CTFd/Flask because it is not strictly supported by Flask. The approach currently used is to modify the `app.view_functions` dictionary which contains the mapping of routes to the functions used to handle them.

```
from flask import render_template

def load(app):
    def view_challenges():
```

```
    def view_challenges():
        return render_template('page.html', content="<h1>Challenges are currently close

    # The format used by the view_functions dictionary is blueprint.view_function_name
    app.view_functions['challenges.challenges_view'] = view_challenges
```

If for some reason you wish to add a new method to an existing route you can modify the `url_map` as follows:

```
from werkzeug.routing import Rule

app.url_map.add(Rule('/challenges', endpoint='challenges.challenges_view', methods=['GE
```

Adding Database Tables

Sometimes CTFd doesn't have enough database tables or columns to let you do what you need. In this case you can use a plugin to create a new table and then use the information in the previous two sections to create routes or modify existing routes to access your new table.

```
from CTFd.models import db


class Avatars(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    team = db.Column(db.Integer, db.ForeignKey('teams.id'))
    location = db.Column(db.Text)

    def __init__(self, team, location):
        self.target = team
        self.location = location


def load(app):
    app.db.create_all()
    @app.route('/profile/avatar', methods=['GET', 'POST'])
    def profile_avatars():
        raise NotImplementedError
```

> **TIP**
>
> For information on how to perform migrations, see Performing Migrations.

To modify your migration for a plugin:

1. Move the generated migration file to the appropriate `CTFd/plugins/<your-plugin>/migrations/` folder
2. Adjust the `down_revision` hash to match the prior revision file's `revision` hash
3. Add `op` to the parameters of the migration functions. For example, change `upgrade()` to `upgrade(op)`.

Replacing Templates

In some situations it might make sense for your plugin to replace the logic for a single page template instead of creating an entire theme.

The `override_template()` function allows a plugin to replace the content of a single template within CTFd such that CTFd will use the new content instead of the content in the original file.

```
from pathlib import Path

from CTFd.utils.plugins import override_template

def load(app):
    dir_path = Path(__file__).parent.resolve()
    template_path = dir_path / 'templates' / 'new-scoreboard.html'
    override_template('scoreboard.html', open(template_path).read())
```

With this code CTFd will use `new-scoreboard.html` instead of the `scoreboard.html` file it normally would have used.

Registering Assets

Very often you will want to provide users with static assets (e.g. JS, CSS). Instead of registering handlers for them on your own, you can use the CTFd built in plugin utilities `register_plugin_assets_directory`

for them on your own, you can use the CTFd built in plugin utilities, `register_plugin_assets_directory` and `register_plugin_asset`.

For example to register an entire assets directory as available to the user:

```
from CTFd.plugins import register_plugin_assets_directory

def load(app):
    # Available at http://ctfd/plugins/test_plugin/assets/
    register_plugin_assets_directory(app, base_path='/plugins/test_plugin/assets/')
```

Or to only provide a single file:

```
from CTFd.plugins import register_plugin_asset

def load(app):
    # Available at http://ctfd/plugins/test_plugin/assets/file.js
    register_plugin_asset(app, asset_path='/plugins/test_plugin/assets/file.js')
```

Previous
« CSV Exporting

Next
Challenge Type Plugins »