falcon-epdb Documentation

Josh Wilson

Feb 07, 2019

Contents:

A Falcon middleware that wraps the excellent epdb tool and allows one to connect to a running Falcon app and use interactive debugging to step through the code.

Better documentation can be found at readthedocs.

Source code can be found on GitHub at jcwilson/falcon-epdb.

CHAPTER 1

Installation

If you are only planning on debugging in a development environment where access to your service is restricted to you or trusted partners, you may find the *Base64* backend sufficient to your purposes. You can just install the library as you would any Python library.

requirements.txt

```
falcon-epdb
```

pip

```
pip install falcon-epdb
```

poetry

```
poetry add falcon-epdb
```

However, if you need a little more security, you can use one of the other authenticated backends (*Fernet*, *JWT*). Choose the one that best fits your use case and install it as a Python extra.

requirements.txt

```
falcon-epdb[fernet]
```

pip

```
pip install falcon-epdb[fernet, jwt]
```

poetry

```
poetry add falcon-epdb[jwt]
```

CHAPTER 2

Usage

This library adds a middleware to your Falcon API stack, and as such will run for all requests, save those excluded by `exempt_methods` provided to the `EPDBServer` constructor. If it detects a well-formed (and possibly authenticated) `X-EPDB` header on the request it will start the epdb server on the configured port and block until it establishes a connection from an epdb client, at which point processing continues but under the control of the remote debugging session.

Subsequent requests with an acceptable header will reuse the client connection and automatically drop into the remote debugging session again.

2.1 Configuring the middleware

The `EPDBServe<falcon_epdb.EPDBServe>` middleware accepts a handful of parameters. The most important are the `backend` and `serve_options` parameters. The `backend` determines how a request is examined for the "secret knock" to start the remote debugging server. The included implementations assume a well-formed `X-EPDB` header, but nothing precludes you from sub-classing `EPDBBackend<falcon_epdb.EPDBBackend>` and implementing your own.

The `serve_options` are options that are passed through to the `epdb.serve()` call. See *Backends* for details on how to add this middleware to your API.

2.2 Constructing the `X-EPDB` header

The content of the header is as follows:

```
{
  "epdb": {}
}
```

Depending on the backend in use, one should encode this content into the appropriate header-safe value. Then append this value to the name of the backend.

```
X-EPDB: Base64 eyJlcGRiIjoge319
```

2.3 Connecting the client

Example code for connecting to the waiting port:

```
import epdb

edpb.connect(host=<host>, port=9000)
```

2.4 Backends

2.4.1 Base64

Server side configuration

```
epdb_middleware = EPDBServe(
    backend=Base64Backend(),
    serve_options={'port': 9000})
api = falcon.API(middleware=[epdb_middleware])
```

Crafting an appropriate header

```
import base64
import json

header_content = base64.b64encode(json.dumps({'epdb': {}}).encode()).decode()
header_value = 'Base64 {}'.format(header_content)
```

2.4.2 Fernet

Server side configuration

```
fernet_key = Fernet.generate_key()  # The shared key
epdb_middleware = EPDBServe(
    backend=FernetBackend(key=fernet_key),
    serve_options={'port': 9000})
api = falcon.API(middleware=[epdb_middleware])
```

Crafting an appropriate header

```
import json
from cryptography.fernet import Fernet

f = Fernet(<fernet_key>)  # Key configured on the server
header_content = f.encrypt(json.dumps({'epdb': {}}).encode()).decode()
header_value = 'Fernet {}'.format(header_content)
```

2.4.3 JWT

Server side configuration

```
jwt_key = uuid.uuid4().hex  # The shared key
epdb_middleware = EPDBServe(
    backend=JWTBackend(key=jwt_key),
    serve_options={'port': 9000})
api = falcon.API(middleware=[epdb_middleware])
```

Crafting an appropriate header

```
import jwt

header_content = jwt.encode({'epdb': {}}, <jwt_key>, algorithm='HS256').decode()
header_value = 'JWT {}'.format(header_content)
```

CHAPTER 3

Troubleshooting

You must be sure to allow access to the configured port on your host. Be sure to check your security groups and firewall rules.

Configure your web app to only run one worker process. If you have multiple workers, only the first one will be able to serve on the configured port. If this is not possible you will have to take steps to ensure that all requests that wish to use the remote debugging port are routed to the same worker. This will depend heavily on your HTTP stack and is beyond the scope of this documentation.

Be sure to up your request timeout limit to something on the order of minutes so that the HTTP server doesn't close your request connection or kill your worker process while you're debugging.

You may need to provide the `HTTP-` prefix on your `X-EPDB` header for it to be handled correctly. So instead of sending `X-EPDB`, you would send `HTTP-X-EPDB`.

CHAPTER 4

API

4.1 Middleware

4.1.1 EPDBServe

class falcon_epdb.**EPDBServe**(*backend*, *exempt_methods=('OPTIONS',)*, *serve_options=None*)

A middleware to enable remote debuging via an epdb server.

Parameters

- **backend** (EPDBBackend) – An instance of the class that will validate and decode the X-EPDB header
- **exempt_methods** (*iterable of strings*) – HTTP methods which will be ignored by this middleware
- **serve_options** (*dictionary*) – Parameters passed-through to epdb.serve()

A client may include a special X-EPDB header containing an appropriately formed payload. If they do, the header will be passed to the configured backend for processing. If the payload passes authentication and meets the content requirements, the app will be begin listening for epdb client connections.

A well-formed header has content simply of the form:

```
{
    "epdb": {}
}
```

The encoding and encryption of this payload is determined by the *EPDBBackend* provided to the middleware.

process_request(*req*, *resp*)

Check for a well-formed X-EPDB header and if present activate the epdb server.

Parameters

- **req** – The Falcon request object

- **resp** – The Falcon response object (unused)

This will block, waiting for an epdb client connection, the first time a valid header is received. Once the client is connected, subsequent passes will simply activate the connected client and drop it into the epdb shell.

The header processing is delegated to the configured *EPDBBackend*.

4.2 Backends

4.2.1 Base64Backend

class falcon_epdb.**Base64Backend**

A simple unauthenticated backend for local development.

decode_header_value(*epdb_header*)

Pull the encrypted data out of the header, if present.

Parameters **epdb_header** (*string*) – The content of the X-EPDB header.

Returns The decoded header payload

Return type dictionary

Raises EPDBException

It expects epdb_header to have the Base64 prefix.

4.2.2 FernetBackend

class falcon_epdb.**FernetBackend**(*key*)

A Python cryptography-based backend that supports a pre-shared key (ie. password) protocol.

Parameters **key** (*bytes*) – The fernet key used to encrypt the header content

---

Note: To use this backend, one must install the cryptography package. The easiest way to do this is to specify the [fernet] extra when adding the falcon-epdb dependency to your project.

Listing 1: **requirements.txt**

```
falcon-epdb[fernet]
```

---

decode_header_value(*epdb_header*)

Pull the encrypted data out of the header, if present.

Parameters **epdb_header** (*string*) – The content of the X-EPDB header.

Returns The decoded and decrypted header payload

Return type dictionary

Raises EPDBException

It expects epdb_header to have the Fernet prefix.

4.2.3 JWTBackend

class falcon_epdb.**JWTBackend**(*key*)
A JWT-based backend that supports a pre-shared key (ie. password) protocol.

Parameters **key** (*bytes*) – The JWT key used to encrypt the header content

---

Note: To use this backend, one must install the `PyJWT` package. The easiest way to do this is to specify the `[jwt]` extra when adding the `falcon-epdb` dependency to your project.

Listing 2: **requirements.txt**

```
falcon-epdb[jwt]
```

---

decode_header_value(*epdb_header*)
Pull the encrypted data out of the header, if present.

Parameters **epdb_header** (*string*) – The content of the `X-EPDB` header.

Returns The decoded and decrypted header payload

Return type dictionary

Raises EPDBException

It expects `epdb_header` to have the `JWT` prefix.

4.2.4 EPDBBackend

class falcon_epdb.**EPDBBackend**
The abstract base class defining the header-processing backend interface.

An inheriting subclass must define *decode_header_value()*, but may define other methods if necessary. This class is structured to provide a balance of convenience and flexibility.

decode_header_value(*epdb_header*)
Process the `X-EPDB` header content.

Parameters **epdb_header** (*string*) – The content of the `X-EPDB` header

Returns The decoded and decrypted header payload

Return type dictionary

This does not need to do any content validation, as that is handled in *validate_header_content()*.

get_header_data(*req*)
Process a request and return the contents of a conforming payload.

Parameters **req** (*Request*) – The Falcon request object

Returns The paylod content or None

Return type dictionary or None

This implementation assumes that the payload is present on the `X-EPDB` header, but subclasses may override this method if their use-case demands it.

If the request does not appear to be attempting begin a debugging session, this will return `None`.

static validate_header_content(*header_content*)
Ensure that the decoded X-EPDB header content is well-formed.

Parameters **header_content** (*dictionary*) – The decoded X-EPDB header content

Returns The value of the epdb key

Return type dictionary

Raises EPDBException

header_content must be of the form:

```
{
    "epdb": {}
}
```

4.3 Exceptions

4.3.1 EPDBException

exception falcon_epdb.**EPDBException**
Raised when an error occurs during the processing of an X-EPDB header.

CHAPTER 5

Development

Issues and pull requests are welcome at GitHub. Please be sure to add or update the documentation appropriately along with your code changes.

5.1 Style checks and testing

All pull requests will be validated with Travis CI, but you may run the tests locally with tox and/or poetry. We use `tox` to wrap `poetry` commands in our Travis CI configuration.

Run the entire suite of tests:

Listing 1: **Using tox**

```
tox
```

Or just run one off tests:

Listing 2: **Using poetry**

```
# Install the project dependencies, including dev-dependencies into a
# poetry-managed virtual environment.
poetry install -E jwt -E fernet

# Run the individual style checks as needed in the virtual environment
poetry run black --check falcon_epdb
poetry run flake8 falcon_epdb tests
poetry run pylint falcon_epdb
poetry run pydocstyle falcon_epdb tests

# Run the unit tests in the virtual environment
poetry run pytest -v tests

# Build the docs and find them in docs/_build
poetry run sphinx-build -b html docs docs/_build
```

5.1.1 Style

5.1.1.1 Conventions

No new-lines in paragraphs in `*.rst` documents to manage line-length. It's too much trouble to add line breaks manually at some arbitrary cut-off point. Your editor should word wrap for you. However, doc-comments in the code should respect the Python file line length.

5.1.1.2 Tools

This project uses several tools to ensure quality and consistency.

black

This is an opinionated code formatter. This is the first thing we check against, as this potentially modifies the code and we wish that the new code remains compliant with the subsequent checks.

While we use it to verify compliant formatting, it is recommended that you install it as a global tool on your own system and apply the auto-formatting prior to commiting your code. It already has out-of-the-box integrations with several popular editors.

If you do not wish to install globally on your system, you can still install it in the `poetry`-managed virtual environment:

```
# Install black unmanaged by poetry in order to get around
# impossible version requirements.
poetry run pip install black

# Run the formatter; will modify files
poetry run black falcon_epdb tests
```

flake8

This is the popular PEP8 tool with a few more improvements.

pylint

The comprehensive, fairly opinionated code quality tool. It generates a score (on a scale of 0 to 10) based on a multitude of criteria. This project has a minimal list of disabled rules, which are disabled to support Python 2.7 support.

pydocstyle

Even documentation needs to set a high bar. Much of the inline doc-comments become part of the auto-generated API documents. This ensures consistency of form as well as of content.

5.2 Adding dependencies

Use the `poetry add` command to add dependencies to the `pyproject.toml` file.

Listing 3: **Using poetry add**

```
poetry add cryptography
poetry add --dev coveralls
```

Note: If you add a non-dev dependency, be sure to also add it to requirement-docs.txt.

5.3 Publishing a new release

The project is configured to publish a release anytime a tag is pushed to the GitHub repository and the build succeeds. The tagging convention is `v<Major>.<minor>.<patch>`, and it should follow semver conventions. One can bump the version using the poetry version command.

When creating a release, ensure the following:

- The documentation is up to date with the new changes.
- The changes have been noted in the CHANGELOG.rst.
- The build "badges" are all passing.
- The version has been incremented accordingly.

CHAPTER 6

Changelog

6.1 v1.1.2

6.1.1 Expand pylint and black scopes

- Apply pylint and black checks to the `tests/` directory
- Emit a more precise error message when the `epdb.serve()` command fails.
- Add pyversions badge to the README

6.2 v1.1.1

6.2.1 Fix PyPI documentation

- Removed some sphinx extension markup that broke the PyPI readme render
- Added a step to the documentation tests to ensure it doesn't happen in the future

6.3 v1.1.0

6.3.1 Cleanup and more tooling

- Removed the `__version__` attribute. It's unnecessary and adds fragile manual maintenance overhead.
  - This would normally be considered a breaking change, but I'm pretty sure no one's using this yet, much less depending on that attribute being present
- Added the `black` code formatter to the development stack
  - Applied it to both code and tests

  - Mostly just converted all strings to double-quotes
  - Removed `pylint-quotes` now that `black` has been added
- Added source code link and badge to `README.rst` for easier navigation from readthedocs.io
- Switched `pip_install` to `false` in `readthedocs.io`
- Added documentation around the style-enforcement tools and other conventions
- Cleaned up some documentation
- Added several project url attributes to `pyproject.toml` in the hopes that poetry and PyPI will display the relative links on the project page.

6.4 v1.0.0

6.4.1 Initial release

- Add support for Fernet backend
- Add support for JWT backend

Index