chepy

Nov 27, 2019

Contents:

Solving a CTF with Chepy

Chepy is a python library with a handy cli that is aimed to mirror some of the capabilities of CyberChef. A reasonable amount of effort was put behind Chepy to make it compatible to the various functionalities that CyberChef offers, all in a pure Pythonic manner. There are some key advantages and disadvantages that Chepy has over Cyberchef. The Cyberchef concept of *stacking* different modules is kept alive in Chepy.

There is still a long way to go for Chepy as it does not offer every single ability of Cyberchef.

CHAPTER 1

Docs

Refer to the docs for full usage information

CHAPTER 2

Example

For all usage and examples, see the docs.

Chepy has a stacking mechanism similar to Cyberchef. For example, this in Cyberchef:

This is equivalent to

```
from chepy import Chepy

file_path = "/tmp/demo/encoding"

print(
    Chepy(file_path)
    .load_file()
    .reverse()
    .rot_13()
    .base64_decode()
    .base32_decode()
    .hexdump_to_str()
    .o
```

(continues on next page)

(continued from previous page)

```
)
```

CHAPTER 3

Installation

Chepy can be installed in a few ways.

3.1 Pypi

```
pip3 install chepy
```

3.2 Git

```
git clone https://github.com/securisec/chepy.git
cd chepy
pip3 install -e .
# I use -e here so that if I update later with git pull, I dont have it install it␣
↪again (unless dependencies have changed)
```

3.2.1 Pipenv

```
git clone https://github.com/securisec/chepy.git
cd chepy
pipenv install
```

3.2.2 Docker

```
docker run --rm -ti -v $PWD:/data securisec/chepy "some string" [somefile, "another␣
↪string"]
```

CHAPTER 4

Chepy vs Cyberchef

4.1 Advantages

- Chepy is pure python with a supporting and accessible python api
- Chepy has a CLI
- Chepy CLI has full autocompletion.
- Extendable via plugins
- Infinitely scalable as it can leverage the full Python library.
- Chepy can interface with the full Cyberchef web app to a certain degree. It is easy to move from Chepy to Cyberchef if need be.
- The Chepy python library is significantly faster than the Cyberchef Node library.
- Works with HTTP/S requests without CORS issues.

4.2 Disadvantages

- Chepy is not a web app (at least for now).
- Chepy does not offer every single thing that Cyberchef does
- Chepy does not have the `magic` method (at the moment)

4.2.1 How to use Chepy

Concept

Chepy shares the same concept of stacking as Cyberchef. This means different methods can be chained together or stack together like Cyberchef. This concept applies to both the CLI and the python library. Chepy does offer some

extras which are not part of the overall Chepy library, but offers handy functionalities. These functionalities are often data generators which cannot really be chained with other methods.

Just like Cyberchef, Chepy is only capable of working with the data that it is provided. What this means is that we cannot give Chepy a bytearray and then try to convert that to upper case. It will throw an error.

Library vs CLI

It is important to keep in mind that Chevy was developed first to be a Python library, and then the CLI was added. There are some functions that are different between the two. It employees a very unique and dynamic CLI implementation. The CLI is self generating and offers autocompletion and built in help. The CLI itself instantiates the Chepy class as a user would when they are using it like a library.

Lets take a look at a quick example. Lets get the HMAC hash of a string. The `hmac_hash` method in Chepy takes one required argument `key`. In python code, we would do:

```
>>> from chepy import Chepy
>>> print(Chepy("A string").hmac_hash(key="secret").o)
9eac8f0d2a45ad4d142227e29327d0f2241071dd
```

In the CLI, we would execute the same thing as:

```
❯ chepy "A string"
>>> hmac_hash --key secret
9eac8f0d2a45ad4d142227e29327d0f2241071dd
```

Notice how both the implementation are very similar. At anytime in the cli, we can type `--` to see if the preceding option takes any required or optional arguments.

See more [examples here(‘./examples.md)

Verbs

Throughout the documentation, and while using the libraries and cli, you will come across certain words. This section will describe what those words mean as they are important.

- **state** Every time a method is called on Chepy, the data has to be stored somewhere for the following methods to be able to access it and use it. This is why almost all methods in Chepy return *self*. **state** is the attribute where this data is stored. When developing a plugin for Chepy, all methods should save the final to state before returning self.
- **states** Because Chepy library and cli both works with multiple inputs, the main Chepy class is instantiated with **args*. Each arg is its own state. You can think of states as tabs in Cyberchef or a browser. Each one is independent of each other.
- **buffers** Because the `state` attribute is always overwritten, we can save the current state in **buffers**. This is similar to vi buffers. We can always load data from a buffer into the current state.

Input methods

The Chepy class in the library, and `nargs` in the cli both take ***args**. This means anything that is used to instantiate the main Chepy class are considered strings. A file path, a url, all are treated as strings.

Example:

```
from chepy import Chepy

c = Chepy("some string", "https://google.com")
```

In this example, both `some string` and `https://google.com` are treated as strings. In this case, Chepy has created two states. State index 0 holds `some string`, and state index 1 holds `https://google.com`. There are some helpful methods that we can use to manipulate our data.

read_file / load_file

Either of these methods can be used to load a file info chepy. These methods load the content of a file (works with binary files also) and saves them to the buffer.

```
c = Chepy("/path/to/some/file.txt").read_file()
```

load_dir

The `load_dir` method is used to load the entire contents of a directory into Chepy. This method optionally takes a pattern argument to specify which files to load. A state is created for each file that matches the pattern in the directory. To load recursively, the pattern `**/*` can be used.

```
c = Chepy("/path/to/dir").load_dir("*.txt")
```

Now we can combine read_file and change_state independently to load whichever file we want into Chepy.

http_request

The last way to load data into Chepy is by making an http request. This is a binder to the requests library, and supports most http request methods, with the option to set headers, cookies, body payload etc. A dictionary of the responses body, headers and cookies are saved in the state by default.

```
c = Chepy("https://google.com").http_request().get_by_key("body")
```

Library

The main class for for Chepy is `Chepy`.

```
from chepy import Chepy
```

The *Chepy* class offers all the chainable / stackable methods in one class. Refer to the docs and *examples* for more use cases.

Individual modules can also be imported from Chepy. They provided slightly limited functionality, but are tied together. For example, if we only want to use the image processing methods from Chepy, we can do

```
from chepy.modules.multimedia import Multimedia
```

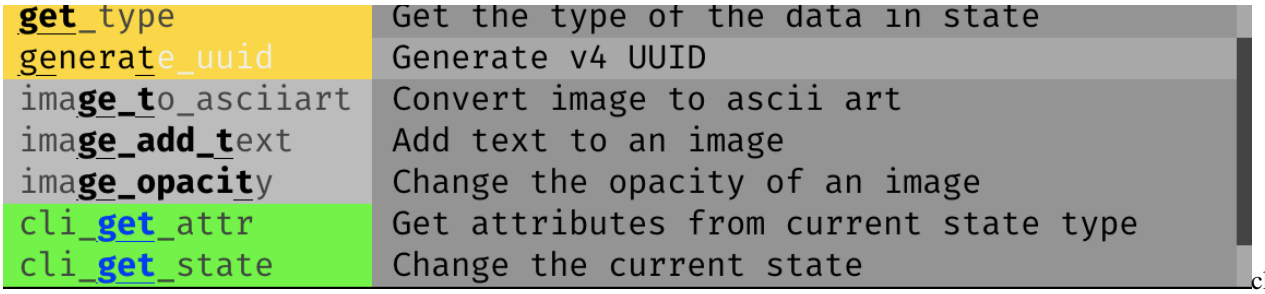
The library does provide some extra functionality also which are not accessed by the Chepy class. These *extras* are documented.

CLI

The Chepy cli is accessed by the `chepy` command which is set to path during the setup process. The cli has all the methods available in the Chepy class, but has some cli specific methods also. The cli autocompletion is color coded

to indicate their functionality. Some methods take either required or optional arguments in the cli. These are also auto populated, and this can be seen by using `--`.

- **default** Default methods does not have any background color
- **green** Green methods are only available as part of the cli. These methods always start with `cli_`. These methods also do not change the state.
- **yellow** These methods will return the current value of the cli, and can no longer be chained.



cli colors

Output methods

Because most methods in Chepy returns self, if you run `type` on an output, you will see it is a Chepy object. To get the value out of the state, we can use a few many methods and attributes.

- **o** This is an attribute
- **output** This is the same as o

```
>>> Chepy("A").to_hex().o
"41"
>>> Chepy("A").to_hex().output
"41"
```

- **out** This is a method that will get the current value from state. Same as `o` and `output`.

```
>>> Chepy("A").to_hex().out()
"41"
```

- **copy** This method will copy the current value of state into the clipboard. For linux, it will require either `xsel` or `xclip`

```
>>> Chepy("A").to_hex().copy()
```

- **web** Open the current Chepy state in Cyberchef itself. This will launch the default browser and put the current state in it.

```
>>> Chepy("A").to_hex().web()
```

- **write** The `write` and the `write_to_file` methods will write the state data to a file. These two methods take an optional argument `as_binary` which can be set to `True` to write as a binary file. A complimentary method `write_binary` is also available which will write directly as binary.

States and Buffers

States

Think of states as tabs in your browser. Best way to understand states is by following this simple example code.

```
>>> c = Chepy("AA", "BB")
"AA"
```

Currently we have two states, one containing AA and the other containing BB. By default, the 0 index state is loaded which contains AA

```
>>> c.to_hex()
"4141"
```

We call the `to_hex` method on the current state. Now state index 0 is `4141`.

```
c.change_state(1)
"BB"
```

Now we are switching to the state and index 1 which contains BB

```
c.to_hex()
"4242"
```

Buffers

Buffers are very similar to states with some key differences.

- States change every time a method is called, but a buffer never changes.
- States are automatically created, but buffers are not.
- Data can be saved in a buffer, and loaded into a state from a buffer.

Lets see an example:

```
c = Chepy("A").save_buffer()
```

Now the state contains an `A` and the buffer contains an `A`

```
c.str_to_hex()
```

Now the state contains `41` while the buffer still contains `A`

4.2.2 Examples

Solving a CTF channel

We are given a file called encoding which contains the following string.

```
=0GDAqREMS0EEWHGOEHJWWRAZqGDHAyHDSxDWyHImRHEEcIGOAyZMqHDPyHFScIDUSyDASREMS0EYS0AWEyJMyRARyHDQWGGUSxJ
```

Script

We can script the solution using the following python script:

```
from chepy import Chepy

c = (
    Chepy("/tmp/demo/encoding")
    .load_file()
    .reverse()
    .rot_13()
    .base64_decode()
    .base32_decode()
    .hexdump_to_str()
)
print(c.o)
StormCTF{Spot3:DcEC6181F48e3B9D3dF77Dd827BF34e0}
```

Cli

asciicast

TAMUCTF challenge

The provided challenge string is:

```
dah-dah-dah-dah-dah dah-di-di-dah di-di-di-di-dit dah-dah-di-di-dit dah-dah-di-di-dit␣
↪dah-dah-dah-dah-dah di-di-dah-dah-dah di-dah dah-di-di-di-dit dah-di-dah-dit di-di-
↪di-di-dit dah-dah-dah-di-dit dah-dah-di-di-dit di-di-di-di-dah di-di-di-di-dah dah-
↪dah-di-di-dit di-di-di-di-dit di-dah-dah-dah-dah di-di-di-dah-dah dah-dah-dah-di-
↪dit dah-di-di-di-dit di-di-di-di-dit di-di-di-dah-dah dah-dah-dah-di-dit dah-dah-di-
↪di-dit di-dah-dah-dah-dah dah-di-di-di-dit dit dah-di-di-di-dit dah-di-dit di-di-di-
↪di-dah dah-di-dit di-di-di-di-dit dah-dah-dah-dah-dit di-di-di-di-dit di-di-di-di-
↪dit di-di-dah-dah-dah di-dah dah-dah-di-di-dit di-di-di-dah-dah dah-dah-di-di-dit␣
↪dah-di-di-di-dit di-di-di-di-dah dah-di-di-di-dit di-di-di-di-dah dah-dah-dah-di-
↪dit dah-di-di-di-dit dah-di-di-dit dah-di-di-di-dit di-dah di-di-di-di-dah dah-dah-
↪dah-dah-dit dah-dah-di-di-dit di-di-di-di-dah di-di-dah-dah-dah di-dah di-di-di-di-
↪dit di-di-dah-dah-dah di-di-di-di-dit di-dah-dah-dah-dah di-di-dah-dah-dah dah-di-
↪di-di-dit di-di-di-di-dah di-dah dah-dah-di-di-dit dah-dah-dah-dah-dah di-di-di-di-
↪dit di-dah dah-dah-di-di-dit dah-di-di-di-dit dah-di-di-di-dit di-dah dah-di-di-di-
↪dit dah-di-dit di-di-dah-dah-dah di-dah-dah-dah-dah di-di-dah-dah-dah di-di-di-di-
↪dit di-di-dah-dah-dah di-di-di-di-dit di-di-di-di-dah dah-di-di-dit di-di-di-di-dah␣
↪di-di-di-di-dah dah-di-di-di-dit dah-di-di-dit dah-di-di-di-dit dah-di-di-di-dit␣
↪dah-dah-di-di-dit dah-dah-dah-dah-dah di-di-dah-dah-dah di-di-di-dah-dah di-di-di-
↪di-dit dit di-di-di-di-dah dit di-di-di-dah-dah dah-dah-dah-dah-dit dah-di-di-di-
↪dit dah-di-di-di-dit dah-di-di-di-dit dah-di-di-dit di-di-di-dah-dah di-di-di-di-
↪dah dah-di-di-di-dit di-di-di-di-dah di-di-di-di-dit di-di-di-di-dit di-di-di-dah-
↪dah di-di-di-di-dah dah-di-di-di-dit dah-di-dah-dit di-di-di-di-dah di-di-dah-dah-
↪dah di-di-di-dah-dah di-di-di-dah-dah dah-dah-di-di-dit di-di-dah-dah-dah di-di-di-
↪di-dit di-di-di-di-dah dah-di-di-di-dit di-di-dah-dit di-di-di-di-dit di-di-di-di-
↪dah di-di-di-dah-dah dah-dah-dah-dah-dah di-di-di-di-dit dah-dah-dah-dah-dah di-di-
↪di-di-dit di-dah di-di-di-di-dit di-dah-dah-dah-dah dah-di-di-di-dit dah-di-dit di-
↪di-di-di-dah di-di-di-dah-dah di-di-di-di-dit di-dah-dah-dah-dah di-di-di-di-dah di-
↪di-di-di-dit di-di-di-di-dah dah-di-di-dit di-di-di-di-dit dah-dah-dah-dah-dit di-
↪di-di-di-dah di-di-dah-dah-dah di-di-di-dah-dah di-di-di-di-dah di-di-di-di-dit di-
↪dah di-di-di-di-dah dah-di-dit dah-dah-di-di-dit dah-di-di-di-dit di-di-dah-dah-dah␣
↪di-dah di-di-dah-dah-dah di-dah-dah-dah-dah di-di-di-di-dah dah-di-di-di-dit dah-di-
↪di-di-dit dah-di-di-dit di-di-di-dah-dah dah-dah-dah-di-dit dah-di-di-di-dit dah-di-
↪dah-dit di-di-dah-dah-dah di-di-di-di-dit dah-di-di-di-dit di-di-dah-dah-dah di-di-
↪di-di-dit di-dah dah-dah-di-di-dit di-dah-dah-dah-dah dah-di-di-di-dit dah-di-dah-
↪dit di-di-di-di-dit dah-dah-dah-dah-dah di-di-di-di-dah dah-di-dit dah-di-di-di-dit␣
↪dah-di-di-di-dit di-di-di-di-dah dah-dah-dah-dah-dit di-di-di-di-dah dah-dah-di-di-
```

(continues on next page)

(continued from previous page)

Script

```
from chepy import Chepy

data = "dah-dah-dah-dah-dah dah-di-di-dah di-di-di-di-dit dah-dah-di-di-dit dah-dah-
↪di-di-dit dah-dah-dah-dah-dah di-di-dah-dah-dah di-dah dah-di-di-di-dit dah-di-dah-
↪dit di-di-di-di-dit dah-dah-dah-di-dit dah-dah-di-di-dit di-di-di-di-dah di-di-di-
↪di-dah dah-dah-di-di-dit di-di-di-di-dit di-dah-dah-dah-dah di-di-di-dah-dah dah-
↪dah-dah-di-dit dah-di-di-di-dit di-di-di-di-dit di-di-di-dah-dah dah-dah-dah-di-dit␣
↪dah-dah-di-di-dit di-dah-dah-dah-dah dah-di-di-di-dit dit dah-di-di-di-dit dah-di-
↪dit di-di-di-di-dah dah-di-dit di-di-di-di-dit dah-dah-dah-dah-dit di-di-di-di-dit␣
↪di-di-di-di-dit di-di-dah-dah-dah di-dah dah-dah-di-di-dit di-di-di-dah-dah dah-dah-
↪di-di-dit dah-di-di-di-dit di-di-di-di-dah dah-di-di-di-dit di-di-di-di-dah dah-dah-
↪dah-di-dit dah-di-di-di-dit dah-di-di-dit dah-di-di-di-dit di-dah di-di-di-di-dah␣
↪dah-dah-dah-dah-dit dah-dah-di-di-dit di-di-di-di-dah di-di-dah-dah-dah di-dah di-
↪di-di-di-dit di-di-dah-dah-dah di-di-di-di-dit di-dah-dah-dah-dah di-di-dah-dah-dah␣
↪dah-di-di-di-dit di-di-di-di-dah di-dah dah-dah-di-di-dit dah-dah-dah-dah-dah di-di-
↪di-di-dit di-dah dah-dah-di-di-dit dah-di-di-di-dit dah-di-di-di-dit di-dah dah-di-
↪di-di-dit dah-di-dit di-di-dah-dah-dah di-dah-dah-dah-dah di-di-dah-dah-dah di-di-
↪di-di-dit di-di-dah-dah-dah di-di-di-di-dit di-di-di-di-dah dah-di-di-dit di-di-di-
↪di-dah di-di-di-di-dah dah-di-di-di-dit dah-di-di-dit dah-di-di-di-dit dah-di-di-di-
↪dit dah-dah-di-di-dit dah-dah-dah-dah-dah di-di-dah-dah-dah di-di-di-dah-dah di-di-
↪di-di-dit dit di-di-di-di-dah dit di-di-di-dah-dah dah-dah-dah-dah-dit dah-di-di-di-
↪dit dah-di-di-di-dit dah-di-di-di-dit dah-di-di-dit di-di-di-dah-dah di-di-di-di-
↪dah dah-di-di-di-dit di-di-di-di-dah di-di-di-di-dit di-di-di-di-dit di-di-di-dah-
↪dah di-di-di-di-dah dah-di-di-di-dit dah-di-dah-dit di-di-di-di-dah di-di-dah-dah-
↪dah di-di-di-dah-dah di-di-di-dah-dah dah-dah-di-di-dit di-di-dah-dah-dah di-di-di-
↪di-dit di-di-di-di-dah dah-di-di-di-dit di-di-dah-dit di-di-di-di-dit di-di-di-di-
↪dah di-di-di-dah-dah dah-dah-dah-dah-dah di-di-di-di-dit dah-dah-dah-dah-dah di-di-
↪di-di-dit di-dah di-di-di-di-dit di-dah-dah-dah-dah dah-di-di-di-dit dah-di-dit di-
↪di-di-di-dah di-di-di-dah-dah di-di-di-di-dit di-dah-dah-dah-dah di-di-di-di-dah di-
↪di-di-di-dit di-di-di-di-dah dah-di-di-dit di-di-di-di-dit dah-dah-dah-dah-dit di-
↪di-di-di-dah di-di-dah-dah-dah di-di-di-dah-dah di-di-di-di-dah di-di-di-di-dit di-
↪dah di-di-di-di-dah dah-di-dit dah-dah-di-di-dit dah-di-di-di-dit dah-di-di-dah-dah␣
↪di-dah di-di-dah-dah-dah di-dah-dah-dah-dah dah-di-di-di-dit dah-di-di-di-dit dah-di-
↪di-di-dit di-dah dah-dah-di-di-dit di-dah-dah-dah-dah dah-di-di-di-dit dah-di-dah-
↪dit di-di-di-di-dit dah-dah-dah-dah-dah di-di-di-dah-dah dah-di-dit dah-di-di-di-dit␣
↪dah-di-di-di-dit di-di-di-di-dah dah-dah-dah-dah-dit di-di-di-di-dah dah-dah-di-di-
↪dit dah-di-di-di-dit dah-di-dit dah-di-di-di-dit di-dah-dah-dah-dah di-di-dah-dah-
↪dah di-di-di-di-dit di-di-dah-dah-dah di-di-di-di-dit di-di-di-di-dah dah-di-di-di-
↪dit dah-dah-di-di-dit di-dah di-di-di-di-dah dah-dah-di-di-dit di-di-dah-dah-dah␣
↪dah-dah-dah-dah-dah dah-di-di-di-dit dah-dah-di-di-dit dah-di-di-di-dit dah-dah-dah-
↪dah-dit dah-di-di-di-dit dah-dah-di-di-dit dah-di-di-di-dit di-di-di-di-dit dah-di-
↪di-di-dit dah-di-dit dah-dah-di-di-dit dah-di-di-dit di-di-di-di-dah di-di-di-dah-
↪dah di-di-di-dah-dah di-dah-dah-dah-dah dah-di-di-di-dit dah-dah-dah-dah-dit dah-di-
↪di-di-dit di-di-di-dah-dah di-di-di-di-dah dah-di-di-dit di-di-di-dah-dah di-di-dah-
↪dit dah-di-di-di-dit di-di-di-dah-dah dah-di-di-di-dah-dah di-di-di-di-dah dah-di-di-
↪dah di-dah-dah-dah-dah di-di-di-di-dah di-di-di-dah-dah di-di-di-di-dah dah-di-di-
↪dit di-di-dah-dah-dah dah-di-dit dah-dah-di-di-dit dah-dah-dah-dah-dit di-di-di-dah-
↪dah dah-dah-dah-dah-dah dah-dah-di-di-dit di-di-di-di-dit di-di-di-di-dit di-di-dah-
↪dit dah-di-di-di-dit dah-dah-dah-di-dit di-di-di-dah-dah di-di-di-di-dah dah-dah-di-
↪di-dit dah-di-di-di-dit di-di-di-dah-dah di-di-di-dah-dah di-di-di-di-dah dah-di-di-
↪dit dah-di-di-di-dit dah-di-dit di-di-di-dah-dah di-di-di-di-dah di-di-di-di-dah␣
↪dah-dah-dah-dah-dit di-di-di-dah-dah di-dah-dah-dah-dah dah-dah-di-di-dit dah-di-
↪dah dah-di-di-di-dit dah-di-di-di-dah di-di-di-di-dit dah-dah-di-di-dit dah-di-dah-
↪di-di-dit dah-di-di-di-dit di-di-di-dah-dah di-di-di-di-dah dah-dah-di-di-dit dah-
↪di-di-dit dah-di-di-di-dah di-di-di-dah-dah di-di-di-di-dah dah-dah-di-di-dit dah-
↪di-di-di-dit dah-dah-di-di-dit di-dah di-di-di-di-dah dah-di-di-dit dah-di-di-di-dit␣
↪di-dah dah-dah-di-di-dit di-di-di-di-dah di-di-di-dah-dah di-di-di-di-dah dah-dah-
↪di-di-dit dah-di-di-di-dit di-di-di-dah-dah di-di-di-di-dah di-di-di-di-dah dah-di-
```

(continues on next page)

(continued from previous page)

```
c = (
    Chepy(data)
    .find_replace("\-", "")
    .find_replace("(dit?)", ".")
    .find_replace("dah", "-")
    .from_morse_code()
    .slice(2)
    .hex_to_str()
    .regex_search("gigem.+}")
)

print(c.o)
>>> ['gigem{C1icK_cl1CK-y0u_h4v3_m4I1}']
```

Cli

asciicast

CyberChef solution

CyberChef

TAMUCTF xor bruteforce

In this example, we will combine the `Chepy` class, with a function from chepy extras called `xor_bruteforce_multi`. The main Chepy class can do single byte xor, but this function can bruteforce any length key.

Script

```
from chepy import Chepy
from chepy.extras.crypto_extras import xor_bruteforce_multi
import re

chall = "XUBdTFdScw5XCVRGTglJXEpMSFpOQE5AVVxJBRpLT10aYBpIVwlbCVZATl1WTBpaTkBOQFVcSQdH"
c = Chepy(chall).base64_decode()
for match in xor_bruteforce_multi(c.o, min=2, max=2):
    if re.search('gigem', match['out'], re.I):
        print(match)
```

The `xor_bruteforce_multi` is not available in the cli.

Convert a png to asciiart

Script

```
from chepy import Chepy

c = (
    Chepy("py.png")
    .load_file()
    .image_to_asciiart(art_width=60)
    .write('art.txt')
)
```

```
.::.:.....................................................:.::.
.::.........................................................::.
:.........................&SSSSSSSSS!.......................:
...................*SSSSSSSSSSSSSSSSSSS.....................
.................SSSS$@SSSSSSSSSSSSSSSSS...................
.................:SSS....SSSSSSSSSSSSSSSS...................
.................:SSS....SSSSSSSSSSSSSSSS...................
.................:SSSSSSSSSSSSSSSSSSSSSSS...................
.................SSSSSSSSSSSSSSSSSSSSSSSS...................
.............................SSSSSSSSSSSS.*****!...........
........SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS.*********.......
......SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS.**********......
.....SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS.***********.....
.....SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS..***********....
....SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS..***********....
....SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS...***********....
....SSSSSSSSSSSSSSSS.....................**********....
....SSSSSSSSSSSSSS....***********************************....
....SSSSSSSSSSSSS...************************************....
.....SSSSSSSSSSS..*************************************.....
.....SSSSSSSSSS..**************************************.....
......SSSSSSSSS..*************************************......
.......SSSSSSSS..*************************************!.......
..........SSSSSS..************.............................
..................************************.................
..................************************.................
..................*****************...****.................
..................*****************.....***.................
..................*****************:.****.................
..................**********************.................
:.....................!***********:.....................:
.::.........................................................::.
.::.:.....................................................:.::.
```

Cli

asciicast

Bruteforce xor

Script

```
from chepy import Chepy

```

(continues on next page)

(continued from previous page)

```
c = (
    Chepy("322422332435612c243232202624")
    .hex_to_str()
    .xor_bruteforce()
)
print(c.o)

{
    ...
    '3a': bytearray(b'\\x08\\x1e\\x18\\t\\x1e\\x0f[\\x16\\x1e\\x08\\x08\\x1a\\x1c\\x1e
↪'),
    '3b': bytearray(b'\\t\\x1f\\x19\\x08\\x1f\\x0eZ\\x17\\x1f\\t\\t\\x1b\\x1d\\x1f'),
    '3c': bytearray(b'\\x0e\\x18\\x1e\\x0f\\x18\\t]\\x10\\x18\\x0e\\x0e\\x1c\\x1a\\x18
↪'),
    '3d': bytearray(b
↪'\\x0f\\x19\\x1f\\x0e\\x19\\x08\\x1b\\x11\\x19\\x0f\\x0f\\x1d\\x1b\\x19'),
    '3e': bytearray(b'\\x0c\\x1a\\x1c\\r\\x1a\\x0b_\\x12\\x1a\\x0c\\x0c\\x1e\\x18\\x1a
↪'),
    '3f': bytearray(b'\\r\\x1b\\x1d\\x0c\\x1b\\n^\\x13\\x1b\\r\\r\\x1f\\x19\\x1b'),
    '40': bytearray(b'rdbsdu!ldrr`fd'),
    '41': bytearray(b'secret message'),
    '42': bytearray(b'pf`qfw#nfppbdf'),
    '43': bytearray(b'qgapgv"ogqqceg'),
    '44': bytearray(b'v`fw`q%h`vvdb`'),
    ...
}
```

m1con mobile CTF - Chepy fork

This example shows how to use the `fork` method in Chepy. The `fork` method allows one to call the same methods on all the states that are available. This avoids duplication. The `fork` method argument structure can be quite complex. Essentially, it is an array of tuples. Each tuple must have the name of the method to call at index 0, followed by a dict of all the arguments that method may require. We then stack these tuples in the order we want them to run.

Steps to take are:

- Load all 4 base64 encoded strings into chepy
- base64 decode them
- decrypt AES

The challenge gives 4 base64 encoded strings, which must be decrypted using AES.

Script

```
from chepy import Chepy
from pprint import pprint

challs = [
    "VgF6Ndz6kbPdTodjKtleWQ==",
    "/gFXZh1UIMgjwgRt3jxIPb94pIKDmcbiW8AghzmWcFA=",
    "Qqb1yxdZYPpO7IkgcwgY8Viv4lmNw/MQlb128tpcC1n+05vNWKRZrypzDWE3rtuG",
    "5CJD6tajuiEnHEHhlSKBZDxlQ0DEGhbZeLC7hpyzaVo=",
]
```

(continues on next page)

(continued from previous page)

```
c = Chepy(*challs).fork(
    [
        ("base64_decode",),
        (
            "aes_decrypt",
            {"key": "kiwi037900000000", "iv": "itsasecret000000", "hex_iv": False},
        ),
    ]
)
pprint(c.states)
{0: b'handsomerob3709',
 1: b'DI{k3y_t0_ev3ryth!ng}',
 2: b'Congratulations on finishing the challenge! :)',
 3: b'DI{Th15_u53r_15_l0gg3d_1n}'}
```

Cli

Important: It is important to remember that the array of tuples being passed to the fork method in the cli does not contain any spaces. This is because of BASH will interpret it. Not tested in Windows.

asciicast

Derbycon CTF 2019

Script

```
from chepy import Chepy

c = (
    Chepy("eJyzMNdRMDQwBBEWOgqWlkCGIZhnCRU3NdBRMDODyZtD1RiAtZlARAwNzQDKrwzB")
    .base64_decode()
    .zlib_decompress()
    .split_by(", ")
    .from_decimal()
    .join_list("")
)
print(c.o)
>>> Welcome2Blackhat
```

Cli

asciicast

4.2.3 Chepy CLI

Chepy CLI is a fully dynamically generated cli that combines the power for python-fire and prompt_toolkit to create its cli. The cli is used very similar to how the main `Chepy` class is used as it allows for method chaining. We know

from the docs that some of the methods in the `Chepy` class takes optional or required arguments. In the cli, these are passed as flags. Refer to the *examples* for use cases.

Using builtins

One of the more advanced functions of the cli allows the user to use arbitrary builtin methods when the state does not contain a Chepy object.

Consider this example in code. We will parse a User agent string in this case:

```
>>> ua = "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like␣
↪Gecko) Chrome/70.0.3538.77 Safari/537.36"
>>> c = Chepy(ua).parse_user_agent()
{'user_agent': {'family': 'Other', 'major': None, 'minor': None, 'patch': None}, 'os
↪': {'family': 'Other', 'major': None, 'minor': None, 'patch': None, 'patch_minor':␣
↪None}, 'device': {'family': 'Other', 'brand': None, 'model': None}, 'string': 'ua'}
# The state type currently is Chepy
>>> c.o
# The state type now is a dict
>>> c.get("user_agent").get("family")
"Chrome"
# we are using the dict builtin method get to pull the values based on keys
```

This same behavior is replicated in the Chepy cli.

asciicast

Cli only methods

For completeness sake everything is document here, but the only functions that are callable from the CLI are functions that start with `cli_`.

Cli shell commands

It is possible to run shell commands from the cli py starting the command with a `!`.

```
>>> !ls -la | grep py
```

This will run the following command inside the Chepy cli

4.2.4 Chepy Class

The **Chepy** class in the main class for Chepy, and includes all the methods from all the different classes under modules. This class takes ***args** as its argument, and each argument that is passed to it becomes its own state.

```
>>> from chepy import Chepy
>>> c = Chepy("some data", "/some/path/file")
>>> c.states
{0: "some data", 1: "/some/path/file"}
```

4.2.5 ChepyCore class

The `ChepyCore` class for Chepy is primarily used as an interface for all the current modules/classes in Chepy, or for plugin development. The `ChepyCore` class is what provides the various attributes like **states**, **buffers**, etc and is required to use and extend Chepy.

The most important `ChepyCore` attributes and methods are:

- **state** The state is where all objects are always stored when modified by any methods.
- **_convert_to_*** methods These are helper methods that ensures data is being accessed and put in the state in the correct manner. For example, `binasii.unhexlify` requires a bytes like object. We can use

```
self.state = binasii.unhexlify(self._convert_to_bytes())
```

This will ensure that the correct data type is being used at all times.

4.2.6 Modules

AritmeticLogic

CodeTidy

Compression

DataFormat

DateTime

EncryptionEncoding

Extractors

Forensics

Hashing

Language

Multimedia

Networking

Other

Publickey

Utils

Chepy Internal

Colors

Exceptions

4.2.7 Extras

Bruteforce

Combinations

Crypto

4.2.8 Chepy Plugins

Chepy allows users to extend Chepy and add plugins to it. This documentation describes how to create or load plugins in Chepy.

chepy.conf file

The chepy.conf file is what controls various aspects of how chepy runs. This file can be located in the users home directory.

The default Chepy conf file on setup looks like this:

```
[Plugin]
pluginpath = None
# this needs to be an absolute path. Plugins are loaded from this directory

[Cli]
historypath = /home/hapsida/.chepy/chepy_history
# controls where the Chepy cli history is saved. This path will be set to
# the users home dir automatically on setup.
```

Plugins folder location

The location of the plugins folder can be found in the `chepy.conf` file. To use custom plugins, set the value of `pluginpath` in this file.

```
[Plugin]
pluginpath = /some/dir/
```

Chepy will attempt to read the plugins folder (if one is set) to resolve any plugins from it.

Creating plugins

Naming plugins

Because Chepy utilizes name spaces to load its plugins, all plugin files needs to be named as **chepy_some_plugin.py**. This ensures that there are no namespace conflicts.

Plugin files should be placed in the directory that is specified by the `pluginpath` in chepy.conf https://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_google.html

Plugin module

All Chepy plugins have to follow a specific format for best results.

- ChepyCore needs to be inherited in the plugin class
- Methods must have google style docstrings.
- Methods should preferably be prefixed with something that distinguishes them. For example `myplugin_somemethod`. This avoids namespace conflicts.

Sample plugin

Plugin Asciinema

This is a bare bones example of how a Chepy plugin works. In this case, `myplugin_method` will be available in both Chepy cli (with auto completion) and the Chepy library.

The only thing this plugin at the moment will do is take whatever value is in the state, and multiply it with 20. All methods in Chepy plugins should set the value of `self.state` and should return `self`. This allows chaining with other methods that are available.

```
import chepy

class MyPlugin(chepy.core.ChepyCore):

    def myplugin_method(self):
        """another method

        Returns:
            Chepy: The chepy object
        """
        self.state = self.state * 20
        return self
```

Lets breakdown this sample plugin.

Importing ChepyCore

```
import chepy

class MyPlugin(chepy.core.ChepyCore):
```

All Chepy plugins needs to inherit the **ChepyCore** class. This ensures that all the core attributes and methods from ChepyCore are available to the plugin.

Docstrings

```
"""another method

Returns:
    Chepy: The chepy object
"""
```

This is an example of Google style docstrings in python. Chepy cli parses these doc strings to show the help message and command completion dynamically. Although this can be omitted, it is strong recommended to have them to leverage the best capabilities of Chepy.

Method body

This could be any code that the method is trying to accomplish

Returns

```
self.state = self.state * 20
return self
```

Important: These two lines are very important. Both Chepy cli and library allows the user to chain various methods with each other.

- `self.state = ...` This line ensures that the value being produced by the method can be accessed by other methods in Chepy.
- `return self` This line ensures that methods can be changed together. Example,

In the example gif and asciinema, we can see how we first load the hello string to chepy, then call our myplugin_method, and then modify the output with to_hex followed by base64_encode.

Using plugins in script

As all plugins found in the directory is loaded automatically by Chepy at init, using plugins in script is super simple.

This code is equivalent to what is happening in the gif and asciinema.

```
from chepy import Chepy

c = Chepy("hello").myplugin_method().to_hex().base64_encode()
print(c)
```

Tip: If you do create a plugin that is helpful, feel free to share it, or make a pull request!

4.2.9 Pull requests

Pull requests for Chepy are very welcome, but the following guidelines needs to be followed.

Code Style

Chepy uses python black for its code style and formatting. All pull requests should have proper formatting applied.

Commit messages

Commit messages should always have proper flair indicating the changes. The first line of the commit message should include the emojis of what was changed followed by multiline description of what was added.

Example commit message

```
added new ability
refactored something
updated something
fixed something
added a new method
added another new method
added new docs
```

The current flairs in use are:

- A new feature has been added. This could be tests files, new arguments etc.
- An update has been made to an existing feature
- A major refactor has taken place. This could be anything in the Cli or ChepyCore classes.
- A new python dependency has been added
- New method has been added
- Added new documentation

Tests

Chepy maintains a 100% Codecov coverage, and all pull requests are required to submit complimentary tests. The tests should include all paths, including coverage for optional arguments, if loops etc. Failing the 100% coverage will automatically fail the configured Github Actions.

Tests requires the following dev dependencies:

- pytest
- pytest-cov
- sphinx
- recommonmark
- bandit

To run tests for coverage and pytest, use:

```
pytest --disable-pytest-warnings --cov-report=xml --cov=chepy --cov-config=.
↪coveragerc tests/
```

For bandit tests, use:

```
bandit --recursive chepy/ --ignore-nosec --skip B101,B413,B303,B310,B112,B304,B320,
↪B410
```

Finally for docs tests, use:

```
make -C docs/ clean html
```

The most convenient way to run all the tests are via the handy `all_tests.sh` from the root directory.

4.2.10 Indices and tables

- genindex
- modindex
- search