Pyarmor Documentation

Release 8.2.0

Jondy Zhao

May 09, 2023

Contents

1 How the documentation is organized **3**

2 Getting help **5**

3 Table of Contents **7**

3.1 Tutorials . . . 7
3.1.1 Getting Started . . . 7
3.1.2 Installation . . . 13
3.1.3 Basic Tutorial . . . 15
3.1.4 Advanced Tutorial . . . 21
3.1.5 Customization and Extension . . . 28
3.2 How To . . . 33
3.2.1 Highest security and performance . . . 33
3.2.2 Protecting Runtime Memory Data . . . 35
3.2.3 Packing with outer key . . . 37
3.2.4 Building obfuscated wheel . . . 38
3.2.5 Protecting system packages . . . 41
3.2.6 Fix encoding error . . . 41
3.2.7 Removing docstring . . . 41
3.2.8 Work with Third-Party Libraries . . . 41
3.2.9 Using Pyarmor License . . . 44
3.3 References . . . 47
3.3.1 Concepts . . . 47
3.3.2 Man Page . . . 49
3.3.3 Building Environments . . . 63
3.3.4 Target Environments . . . 66
3.3.5 Error Messages . . . 69
3.4 Topics . . . 71
3.4.1 Insight Into Obfuscation . . . 71
3.4.2 Understanding Obfuscated Script . . . 72
3.4.3 Insight Into Pack Command . . . 74
3.4.4 Insight Into RFT Mode . . . 76
3.4.5 Insight Into BCC Mode . . . 80
3.4.6 Security and Performance . . . 82
3.4.7 Localization and Internationalization . . . 86
3.5 License Types . . . 87
3.5.1 Introduction . . . 87

3.5.2 License types . . . 88
3.5.3 Purchasing license . . . 89
3.5.4 Upgrading old license . . . 90
3.6 FAQ . . . 91
3.6.1 Asking questions in GitHub . . . 91
3.6.2 Packing . . . 92
3.6.3 License . . . 92
3.6.4 Purchasing . . . 93

4 Indices and tables **95**

Python Module Index **97**

Index **99**

Version 8.2.0

Homepage https://pyarmor.dashingsoft.com/

Contact pyarmor@163.com

Authors Jondy

CHAPTER 1

How the documentation is organized

Pyarmor has a lot of documentation. A high-level overview of how it's organized will help you know where to look for certain things:

- *Part 1: Tutorials* takes you by the hand through a series of steps to obfuscate *Python* scripts and packages. Start here if you're new to *Pyarmor*. Also look at the *Getting Started*
- *Part 2: How To* guides are recipes. They guide you through the steps involved in addressing key problems and use-cases. They are more advanced than tutorials and assume some knowledge of how *Python* works.
- *Part 3: References* guides contain key concepts, man page, configurations and other aspects of *Pyarmor* machinery.
- *Part 4: Topics* guides insight into key topics and provide useful background information and explanation. They describe how it works and how to use it but assume that you have a basic understanding of key concepts.
- *Part 5: Licenses* describes EULA of *Pyarmor*, the different *Pyarmor* licenses and how to purchase *Pyarmor* license.

CHAPTER 2

Getting help

Having trouble?

Try the *FAQ* – it's got answers to many common questions.

Looking for specific information? Try the genindex, or *the detailed table of contents*.

Not found anything? See *asking questions in github*.

Report bugs with Pyarmor in issues

CHAPTER 3

Table of Contents

3.1 Tutorials

3.1.1 Getting Started

Content

- *What's Pyarmor*
- *Installation from PyPI*
- *Obfuscating one script*
  - *Distributing the obfuscated script*
- *Obfuscating one package*
  - *Distributing the obfuscated package*
- *Expiring obfuscated scripts*
- *Binding obfuscated scripts to device*
- *Packaging obfuscated scripts*
- *Something need to know*
- *What to read next*
- *How the documentation is organized*

New to *Pyarmor*? Well, you came to the right place: read this material to quickly get up and running.

What's Pyarmor

Pyarmor is a command-line tool designed for obfuscating Python scripts, binding obfuscated scripts to specific machines, and setting expiration dates for obfuscated scripts.

Key Features:

- **Seamless Replacement**: Obfuscated scripts remain as standard *.py* files, allowing them to seamlessly replace the original Python scripts in most cases.
- **Balanced Obfuscation**: Offers multiple ways to obfuscate scripts to balance security and performance.
- **Irreversible Obfuscation**: Renames functions, methods, classes, variables, and arguments.
- **C Function Conversion**: Converts some Python functions to C functions and compiles them into machine instructions using high optimization options for irreversible obfuscation.
- **Script Binding**: Binds obfuscated scripts to specific machines or sets expiration dates for obfuscated scripts.
- **Themida Protection**: Protects obfuscated scripts using Themida (Windows only).

Installation from PyPI

Pyarmor packages are published on the PyPI. The preferred tool for installing packages from PyPI is **`pip`**. This tool is provided with all modern versions of Python.

On Linux or MacOS, you should open your terminal and run the following command:

```
$ pip install -U pyarmor
```

On Windows, you should open Command Prompt (`Win-r` and type **`cmd`**) and run the same command:

```
C:\> pip install -U pyarmor
```

After installation, type **`pyarmor --version`** on the command prompt. If everything worked fine, you will see the version number for the Pyarmor package you just installed.

Not all the platforms are supported, more information check *Building Environments*

Obfuscating one script

Here it's the simplest command to obfuscate one script `foo.py`:

```
$ pyarmor gen foo.py
```

The command `gen` could be replaced with `g` or `generate`:

```
$ pyarmor g foo.py
$ pyarmor generate foo.py
```

This command generates an obfuscated script `dist/foo.py`, which is a valid Python script, run it by Python interpreter:

```
$ python dist/foo.py
```

Check all generated files in the default output path:

```
$ ls dist/
...    foo.py
...    pyarmor_runtime_000000
```

There is an extra Python package `pyarmor_runtime_000000`, which is required to run the obfuscated script.

Distributing the obfuscated script

Only copy `dist/foo.py` to another machine doesn't work, instead copy all the files in the `dist/`.

Why? It's clear after checking the content of `dist/foo.py`:

```
from pyarmor_runtime_000000 import __pyarmor__
__pyarmor__(__name__, __file__, ...)
```

Actually the obfuscated script can be taken as normal Python script with dependent package `pyarmor_runtime_000000`, use it as it's not obfuscated.

Important: Please run this obfuscated in the machine with same Python version and same platform, otherwise it doesn't work. Because `pyarmor_runtime_000000` has an *extension module*, it's platform-dependent and bind to Python version.

Note: DO NOT install Pyarmor in the *Target Device*, Python interpreter could run the obfuscated scripts without Pyarmor.

Obfuscating one package

Now let's do a package. *-O* is used to set output path `dist2` different from the default:

```
$ pyarmor gen -O dist2 src/mypkg
```

Check the output:

```
$ ls dist2/
...    mypkg
...    pyarmor_runtime_000000

$ ls dist2/mypkg/
...          __init__.py
```

All the obfuscated scripts in the `dist2/mypkg`, test it:

```
$ cd dist2/
$ python -C 'import mypkg'
```

If there are sub-packages, using *-r* to enable recursive mode:

```
$ pyarmor gen -O dist2 -r src/mypkg
```

Distributing the obfuscated package

Also it works to copy the whole path `dist2` to another machine. But it's not convenience, the better way is using `-i` to generate all the required files inside package path:

```
$ pyarmor gen -O dist3 -r -i src/mypkg
```

Check the output:

```
$ ls dist3/
...    mypkg

$ ls dist3/mypkg/
...          __init__.py
...          pyarmor_runtime_000000
```

Now everything is in the package path `dist3/mypkg`, just copy the whole path to any target machine.

Note: Comparing current `dist3/mypkg/__init__.py` with above section `dist2/mypkg/__init__.py` to understand more about obfuscated scripts

---

Expiring obfuscated scripts

It's easy to set expire date for obfuscated scripts by `-e`. For example, generate obfuscated script with the expire date to 30 days:

```
$ pyarmor gen -O dist4 -e 30 foo.py
```

Run the obfuscated scripts `dist4/foo.py` to verify it:

```
$ python dist4/foo.py
```

It checks network time, make sure your machine is connected to internet.

Let's use another form to set past date `2020-12-31`:

```
$ pyarmor gen -O dist4 -e 2020-12-31 foo.py
```

Now `dist4/foo.py` should not work:

```
$ python dist4/foo.py
```

If expire date has a leading `.`, it will check local time other than NTP server. For examples:

```
$ pyarmor gen -O dist4 -e .30 foo.py
$ pyarmor gen -O dist4 -e .2020-12-31 foo.py
```

For this form internet connection is not required in target machine.

Distributing the expired script is same as above, copy the whole directory `dist4/` to target machine.

Binding obfuscated scripts to device

Suppose got target machine hardware information:

```
IPv4:                           128.16.4.10
Ethernet Addr:                  00:16:3e:35:19:3d
Hard Disk Serial Number:        HXS2000CN2A
```

Using *-b* to bind hardware information to obfuscated scripts. For example, bind `dist5/foo.py` to Ethernet address:

```
$ pyarmor gen -O dist5 -b 00:16:3e:35:19:3d foo.py
```

So `dist5/foo.py` only could run in target machine.

It's same to bind IPv4 and serial number of hard disk:

```
$ pyarmor gen -O dist5 -b 128.16.4.10 foo.py
$ pyarmor gen -O dist5 -b HXS2000CN2A foo.py
```

It's possible to combine some of them. For example:

```
$ pyarmor gen -O dist5 -b "00:16:3e:35:19:3d HXS2000CN2A" foo.py
```

Only both Ethernet address and hard disk are matched machine could run this obfuscated script.

Distributing scripts bind to device is same as above, copy the whole directory `dist5/` to target machine.

Packaging obfuscated scripts

Remember again, the obfuscated script is normal Python script, use it as it's not obfuscated.

Suppose package `mypkg` structure like this:

```
projects/
└── src/
    └── mypkg/
        ├── __init__.py
        ├── utils.py
        └── config.json
```

First make output path `projects/dist6` for obfuscated package:

```
$ cd projects
$ mkdir dist6
```

Then copy package data files to output path:

```
$ cp -a src/mypkg dist6/
```

Next obfuscate scripts to overwrite all the `.py` files in `dist6/mypkg`:

```
$ pyarmor gen -O dist6 -i src/mypkg
```

The final output:

```
projects/
├── README.md
└── src/
    └── mypkg/
        ├── __init__.py
        ├── utils.py
```

(continues on next page)

(continued from previous page)

```
        └── config.json
└── dist6/
    └── mypkg/
        ├── __init__.py
        ├── utils.py
        ├── config.json
        └── pyarmor_runtime_000000/__init__.py
```

Comparing with `src/mypkg`, the only difference is `dist6/mypkg` has an extra sub-package `pyarmor_runtime_000000`. The last thing is packaging `dist6/mypkg` as your prefer way.

New to Python packaging? Refer to Python Packaging User Guide

Something need to know

There is binary extension module `pyarmor_runtime` in extra sub-package `pyarmor_runtime_000000`, here it's package content:

```
$ ls dist6/mypkg/pyarmor_runtime_000000
...    __init__.py
...    pyarmor_runtime.so
```

Generally using binary extensions means the obfuscated scripts require `pyarmor_runtime` be created for different platforms, so they

- only works for platforms which provides pre-built binaries, refer to *Building Environments*
- may not be compatible with different builds of CPython interpreter. For example, when obfuscating scripts by Python 3.8, they can't be run by Python 3.7, 3.9 etc.
- often will not work correctly with alternative interpreters such as PyPy, IronPython or Jython

Another disadvantage of relying on binary extensions is that alternative import mechanisms (such as the ability to import modules directly from zipfiles) often won't work for extension modules (as the dynamic loading mechanisms on most platforms can only load libraries from disk).

What to read next

There is a complete *installation* guide that covers all the possibilities:

- install pyarmor by source
- call pyarmor from Python script
- clean uninstallation

Next is *Basic Tutorial*. It covers

- using more option to obfuscate script and package
- using outer file to store runtime key
- localizing runtime error messages
- packing obfuscated scripts and protect system packages

And then *Advanced Tutorial*, some of them are not available in trial pyarmor

- 2 irreversible obfuscation: RFT mode, BCC mode [pro]

- Customization error handler
- runtime error internationalization
- cross platform, multiple platforms and multiple Python version

Also you may be interesting in this guide *Highest security and performance*

How the documentation is organized

Pyarmor has a lot of documentation. A high-level overview of how it's organized will help you know where to look for certain things:

- *Part 1: Tutorials* now you're reading.
- *Part 2: How To* guides are recipes. They guide you through the steps involved in addressing key problems and use-cases. They are more advanced than tutorials and assume some knowledge of how *Python* works.
- *Part 3: References* guides contain key concepts, man page, configurations and other aspects of *Pyarmor* machinery.
- *Part 4: Topics* guides insight into key topics and provide useful background information and explanation. They describe how it works and how to use it but assume that you have a basic understanding of key concepts.
- *Part 5: Licenses* describes EULA of *Pyarmor*, the different *Pyarmor* licenses and how to purchase *Pyarmor* license.

Looking for specific information? Try the genindex, or *the detailed table of contents*.

3.1.2 Installation

Contents

- *Installation from PyPI*
  - *Installed command*
  - *Start Pyarmor by Python interpreter*
- *Using virtual environments*
- *Installation from source*
- *Run Pyarmor from Python script*
- *Clean uninstallation*

Installation from PyPI

Pyarmor packages are published on the PyPI. The preferred tool for installing packages from PyPI is **pip**. This tool is provided with all modern versions of Python.

On Linux or MacOS, you should open your terminal and run the following command:

```
$ pip install -U pyarmor
```

On Windows, you should open Command Prompt (`Win-r` and type **cmd**) and run the same command:

```
C:\> pip install -U pyarmor
```

After installation, type **pyarmor --version** on the command prompt. If everything worked fine, you will see the version number for the Pyarmor package you just installed.

Installation from PyPI also allows you to install the latest development release. You will not generally need (or want) to do this, but it can be useful if you see a possible bug in the latest stable release. To do this, use the `--pre` flag:

```
$ pip install -U --pre pyarmor
```

If you need generate obfuscated scripts to run in other platforms, install *pyarmor.cli.runtime*:

```
$ pip install pyarmor.cli.runtime
```

Not all the platforms are supported, more information check *Building Environments*

Installed command

- **pyarmor** is the main command to do everything. See *Man Page*.
- **pyarmor-7** is used to call old commands, it equals bug fixed Pyarmor 7.x

Start Pyarmor by Python interpreter

pyarmor is same as the following command:

```
$ python -m pyarmor.cli
```

Using virtual environments

When installing Pyarmor using **pip**, use *virtual environments* which could isolate the installed packages from the system packages, thus removing the need to use administrator privileges. To create a virtual environment in the `.venv` directory, use the following command:

```
$ python -m venv .venv
```

You can read more about them in the Python Packaging User Guide.

Installation from source

You can install Pyarmor directly from a clone of the Git repository. This can be done either by cloning the repo and installing from the local clone, on simply installing directly via **git**:

```
$ git clone https://github.com/dashingsoft/pyarmor
$ cd pyarmor
$ pip install .
```

You can also download a snapshot of the Git repo in either tar.gz or zip format. Once downloaded and extracted, these can be installed with **pip** as above.

Run Pyarmor from Python script

Create a script `tool.py`, pass arguments by yourself

```
from pyarmor.cli.__main__ import main_entry

args = ['gen', 'foo.py']
main(args)
```

Run it by Python interpreter:

```
$ python tool.py
```

Clean uninstallation

Run the following commands to make a clean uninstallation:

```
$ pip uninstall pyarmor
$ pip uninstall pyarmor.cli.core
$ pip uninstall pyarmor.cli.runtime
$ rm -rf ~/.pyarmor
$ rm -rf ./.pyarmor
```

Note: The path ~ may be different when logging by different user. `$HOME` is home path of current logon user, check the environment variable `HOME` to get the real path.

3.1.3 Basic Tutorial

Contents

- *Debug mode and trace log*
- *More options to protect script*
- *More options to protect package*
- *Copying package data files*
- *Checking runtime key periodically*
- *Binding to many machines*
- *Using outer file to store runtime key*
- *Localization runtime error*
- *Packing obfuscated scripts*
  - *Packing to one file*
  - *Packing to one folder*

We'll assume you have Pyarmor 8.0+ installed already. You can tell Pyarmor is installed and which version by running the following command in a shell prompt (indicated by the $ prefix):

```
$ pyarmor --version
```

If Pyarmor is installed, you should see the version of your installation. If it isn't, you'll get an error.

This tutorial is written for Pyarmor 8.0+, which supports Python 3.7 and later. If the Pyarmor version doesn't match, you can refer to the tutorial for your version of Pyarmor by using the version switcher at the bottom right corner of this page, or update Pyarmor to the newest version.

Throughout this tutorial, assume run **`pyarmor`** in project path which includes:

```
project/
     ├── foo.py
     ├── queens.py
     └── joker/
         ├── __init__.py
         ├── queens.py
         └── config.json
```

Pyarmor uses *pyarmor gen* with rich options to obfuscate scripts to meet the needs of different applications.

Here only introduces common options in a short, using any combination of them as needed. About usage of each option in details please refer to *pyarmor gen*

Debug mode and trace log

When something is wrong, check console log to find what Pyarmor does, and use *-d* to enable debug mode to print more information:

```
$ pyarmor -d gen foo.py
```

Trace log is useful to check whatever protected by Pyarmor, enable it by this command:

```
$ pyarmor cfg enable_trace=1
```

After that, *pyarmor gen* will generate a logfile `.pyarmor/pyarmor.trace.log`. For example:

```
$ pyarmor gen foo.py
$ cat .pyarmor/pyarmor.trace.log

trace.co              foo:1:<module>
trace.co              foo:5:hello
trace.co              foo:9:sum2
trace.co              foo:12:main
```

Each line starts with `trace.co` is reported by code object protector. The first log says `foo.py` module level code is obfuscated, second says function `hello` at line 5 is obfuscated, and so on.

Enable both debug and trace mode could show much more information:

```
$ pyarmor -d gen foo.py
```

Disable trace log by this command:

```
$ pyarmor cfg enable_trace=0
```

More options to protect script

For scripts, use these options to get more security:

```
$ pyarmor gen --enable-jit --mix-str --assert-call --private foo.py
```

Using *--enable-jit* tells Pyarmor processes some sensitive data by `c` function generated in runtime.

Using *--mix-str*[1] could mix the string constant (length > 8) in the scripts.

Using *--assert-call* makes sure function is obfuscated, to prevent called function from being replaced by special ways

Using *--private* makes the script could not be imported by plain scripts

For example,

```
data = "abcefgxyz"

def fib(n):
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b

if __name__ == '__main__':
    fib(n)
```

String constant `abcefgxyz` and function `fib` will be protected like this

```
data = __mix_str__(b"******")

def fib(n):
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b

if __name__ == '__main__':
    __assert_call__(fib)(n)
```

If function `fib` is obfuscated, `__assert_call__(fib)` returns original function `fib`. Otherwise it will raise protection exception.

To check which function or which string are protected, enable trace log and check trace logfile:

```
$ pyarmor cfg enable_trace=1
$ pyarmor gen --mix-str --assert-call fib.py
$ cat .pyarmor/pyarmor.trace.log

trace.assert.call    fib:10:'fib'
trace.mix.str        fib:1:'abcxyz'
trace.mix.str        fib:9:'__main__'
trace.co             fib:1:<module>
trace.co             fib:3:fib
```

[1] *--mix-str* is not available in trial version

More options to protect package

For package, remove *--private* and append 2 extra options:

```
$ pyarmor gen --enable-jit --mix-str --assert-call --assert-import --restrict joker/
```

Using *--assert-import* prevents obfuscated modules from being replaced with plain script. It checks each import statement to make sure the modules are obfuscated.

Using *--restrict* makes sure the obfuscated module is only available inside package. It couldn't be imported from any plain script, also not be run by Python interpreter.

By default `__init__.py` is not restricted, all the other modules are invisible from outside. Let's check this, first create a script `dist/a.py`

```
import joker
print('import joker OK')
from joker import queens
print('import joker.queens OK')
```

Then run it:

```
$ cd dist
$ python a.py
... import joker OK
... RuntimeError: unauthorized use of script
```

In order to export `joker.queens`, either removing option *--restrict*, or config only this module is not restrict like this:

```
$ pyarmor cfg -p joker.queens restrict_module=0
```

Then obfuscate this package with restrict mode:

```
$ pyarmor gen --restrict joker/
```

Now do above test again, it should work:

```
$ cd dist/
$ python a.py
... import joker OK
... import joker.queens
```

Copying package data files

Many packages have data files, but they're not copied to output path.

There are 2 ways to solve this problem:

1. Before generating the obfuscated scripts, copy the whole package to output path, then run *pyarmor gen* to overwrite all the `.py` files:

   ```
   $ mkdir dist/joker
   $ cp -a joker/* dist/joker
   $ pyarmor gen -O dist -r joker/
   ```

2. Changing default configuration let Pyarmor copy data files:

```
$ pyarmor cfg data_files=*
$ pyarmor gen -O dist -r joker/
```

Checking runtime key periodically

Checking runtime key every hour:

```
$ pyarmor gen --period 1 foo.py
```

Binding to many machines

Using *-b* many times to bind obfuscated scripts to many machines.

For example, machine A and B, the ethernet addresses are `66:77:88:9a:cc:fa` and `f8:ff:c2:27:00:7f` respectively. The obfuscated script could run in both of machine A and B by this command

```
$ pyarmor gen -b "66:77:88:9a:cc:fa" -b "f8:ff:c2:27:00:7f" foo.py
```

Using outer file to store runtime key

First obfuscating script with *--outer*:

```
$ pyarmor gen --outer foo.py
```

In this case, it could not be run at this time:

```
$ python dist/foo.py
```

Let generate an outer runtime key valid for 3 days by this command:

```
$ pyarmor gen key -e 3
```

It generates a file `dist/pyarmor.rkey`, copy it to runtime package:

```
$ cp dist/pyarmor.rkey dist/pyarmor_runtime_000000/
```

Now run `dist/foo.py` again:

```
$ python dist/foo.py
```

Let's generate another license valid for 10 days:

```
$ pyarmor gen key -O dist/key2 -e 10

$ ls dist/key2/pyarmor.rkey
```

Copy it to runtime package to replace the original one:

```
$ cp dist/key2/pyarmor.rkey dist/pyarmor_runtime_000000/
```

The outer runtime key file also could be saved to other paths, refer to *outer key*.

Localization runtime error

Some of runtime error messages could be customized. When something is wrong with the obfuscated scripts, it prints your own messages.

First create `messages.cfg` in the path `.pyarmor`:

```
$ mkdir .pyarmor
$ vi .pyarmor/message.cfg
```

Then edit it. It's a `.ini` format file, change the error messages as needed

```
[runtime.message]

  error_1 = this license key is expired
  error_2 = this license key is not for this machine
  error_3 = missing license key to run the script
  error_4 = unauthorized use of script
```

Now obfuscate the script in the current path to use customized messages:

```
$ pyarmor gen foo.py
```

If we want to show same message for all of license errors, edit it like this

```
[runtime.message]

  error_1 = invalid license key
  error_2 = invalid license key
  error_3 = invalid license key
```

Here no `error_4`, it means this error uses the default message.

And then obfuscate the scripts again.

Packing obfuscated scripts

Pyarmor need PyInstaller to pack scripts first, then replace plain scripts with obfuscated ones in bundle.

Packing to one file

First packing script to one file by PyInstaller with option `-F`:

```
$ pyinstaller -F foo.py
```

It generates one bundle file `dist/foo`, pass this to pyarmor:

```
$ pyarmor gen -O obfdist --pack dist/foo foo.py
```

This command will obfuscate `foo.py` first, then repack `dist/foo`, replace the original `foo.py` with `obfdist/foo.py`, and append all the runtime files to bundle.

The final output is still `dist/foo`:

```
$ dist/foo
```

Packing to one folder

First packing script to one folder by PyInstaller:

```
$ pyinstaller foo.py
```

It generates one bundle folder `dist/foo`, and an executable file `dist/foo/foo`, pass this executable to pyarmor:

```
$ pyarmor gen -O obfdist --pack dist/foo/foo foo.py
```

Like above section, `dist/foo/foo` will be repacked with obfuscated scripts.

Now run it:

```
$ dist/foo/foo
```

More information about pack feature, refer to *Insight Into Pack Command*

3.1.4 Advanced Tutorial

Contents

- *Using rftmode* [pro]
- *Using bccmode* [pro]
- *Customization error handler*
- *Filter mix string*
- *Filter assert function and import*
- *Patching source by inline marker*
- *Internationalization runtime error message*
- *Generating cross platform scripts*
- *Obfuscating scripts for multiple Pythons*

Using rftmode [pro]

RFT mode could rename most of builtins, functions, classes, local variables. It equals rewriting scripts in source level.

Using `--enable-rft` to enable RTF mode[1]:

```
$ pyarmor gen --enable-rft foo.py
```

For example, this script

```
1 import sys
2
3 def sum2(a, b):
4     return a + b
```

(continues on next page)

[1] This feature is only available for *Pyarmor Pro*.

(continued from previous page)

```

def main(msg):
    a = 2
    b = 6
    c = sum2(a, b)
    print('%s + %s = %d' % (a, b, c))

if __name__ == '__main__':
    main('pass: %s' % data)
```

transform to

```
pyarmor__17 = __assert_armored__(b'\x83\xda\x03sys')

def pyarmor__22(a, b):
    return a + b

def pyarmor__16(msg):
    pyarmor__23 = 2
    pyarmor__24 = 6
    pyarmor__25 = pyarmor__22(pyarmor__23, pyarmor__24)
    pyarmor__14('%s + %s = %d' % (pyarmor__23, pyarmor__24, pyarmor__25))

if __name__ == '__main__':
    pyarmor__16('pass: %s' % pyarmor__20)
```

By default if RFT mode doesn't make sure this name could be changed, it will leave this name as it is.

RFT mode doesn't change names in the module attribute `__all__`, it also doesn't change function arguments.

For example, this script

```
import re

__all__ = ['make_scanner']

def py_make_scanner(context):
    parse_obj = context.parse_object
    parse_arr = context.parse_array

make_scanner = py_make_scanner
```

transform to

```
pyarmor__3 = __assert_armored__(b'\x83e\x9d')

__all__ = ['make_scanner']

def pyarmor__1(context):
    pyarmor__4 = context.parse_object
    pyarmor__5 = context.parse_array

make_scanner = pyarmor__1
```

If want to know what're refactored exactly, enable trace rft to generate transformed script[2]:

[2] This feature only works for Python 3.9+

```
$ pyarmor cfg trace_rft=1
$ pyarmor gen --enable-rft foo.py
```

The transformed script will be stored in the path `.pyarmor/rft`:

```
$ cat .pyarmor/rft/foo.py
```

Now run the obfuscated script:

```
$ python dist/foo.py
```

If something is wrong, try to obfuscate it again, it may make senses:

```
$ pyarmor gen --enable-rft foo.py
$ python dist/foo.py
```

If it still doesn't work, or you need transform more names, refer to *Insight Into RFT Mode* to learn more usage.

Using bccmode pro

BCC mode could convert most of functions and methods in the scripts to equivalent C functions, those c functions will be compiled to machine instructions directly, then called by obfuscated scripts.

It requires `c` compiler. In Linux and Darwin, `gcc` and `clang` is OK. In Windows, only `clang.exe` works. It could be configured by one of these ways:

- If there is any `clang.exe`, it's OK if it could be run in other path.
- Download and install Windows version of LLVM
- Download *https://pyarmor.dashingsoft.com/downloads/tools/clang-9.0.zip*, it's about 26M bytes, there is only one file in it. Unzip it and save `clang.exe` to `$HOME/.pyarmor/`. `$HOME` is home path of current logon user, check the environment variable `HOME` to get the real path.

After compiler works, using *`--enable-bcc`* to enable BCC mode[3]:

```
$ pyarmor gen --enable-bcc foo.py
```

All the source in module level is not converted to C function.

To check which functions are converted to C function, enable trace mode before obfuscate the script:

```
$ pyarmor cfg enable_trace=1
$ pyarmor gen --enable-bcc foo.py
```

Then check the trace log:

```
$ ls .pyarmor/pyarmor.trace.log
$ grep trace.bcc .pyarmor/pyarmor.trace.log

trace.bcc            foo:5:hello
trace.bcc            foo:9:sum2
trace.bcc            foo:12:main
```

The first log means `foo.py` line 5 function `hello` is protected by bcc. The second log means `foo.py` line 9 function `sum2` is protected by bcc.

[3] This feature is only available for *Pyarmor Pro*.

When something is wrong, enable debug mode by common option `-d`:

```
$ pyarmor -d gen --enable-bcc foo.py
```

Check console log and trace log, most of cases there is modname and line no in console or trace log. Assume the problem function is `sum2`, then tell BCC mode does not deal with it by this way:

```
$ pyarmor cfg -p foo bcc:excludes "sum2"
```

Use `-p` to specify mod-name, and option `bcc:excludes` for function name.

Append more functions to exclude by this way:

```
$ pyarmor cfg -p foo bcc:excludes + "hello"
```

When obfuscating package, it also could exclude one script separately. For example, the following commands tell BCC mode doesn't handle `joker/card.py`, but all the other scripts in package `joker` are still handled by BCC mode:

```
$ pyarmor cfg -p joker.card bcc:disabled=1
$ pyarmor gen --enable-bcc /path/to/pkg/joker
```

It's possible that BCC mode could not support some Python features, in this case, use `bcc:excludes` and `bcc:disabled` to ignore them, and make all the others work.

If it still doesn't work, or you want to know more about BCC mode, goto *Insight Into BCC Mode*.

Customization error handler

By default when something is wrong with obfuscated scripts, RuntimeError with traceback is printed:

```
$ pyarmor gen -e 2020-05-05 foo.py
$ python dist/foo.py

Traceback (most recent call last):
  File "dist/foo.py", line 2, in <module>
    from pyarmor_runtime_000000 import __pyarmor__
  File "dist/pyarmor_runtime_000000/__init__.py", line 2, in <module>
    from .pyarmor_runtime import __pyarmor__
RuntimeError: this license key is expired (1:10937)
```

If prefer to show error message only:

```
$ pyarmor cfg on_error=1

$ pyarmor gen -e 2020-05-05 foo.py
$ python dist/foo.py

this license key is expired (1:10937)
```

If prefer to quit directly without any message:

```
$ pyarmor cfg on_error=2

$ pyarmor gen -e 2020-05-05 foo.py
$ python dist/foo.py

$
```

Restore the default handler:

```
$ pyarmor cfg on_error=0
```

Or reset this option:

```
$ pyarmor cfg --reset on_error
```

Note: This only works for execute the obfuscated scripts by Python interpreter directly. If *--pack* is used, the script is loaded by PyInstaller loader, it may not work as expected.

Filter mix string

By default *--mix-str* encrypts all the string length > 8.

But it can be configured to filter any string to meet various needs.

Exclude short strings by length < 10:

```
$ pyarmor cfg mix.str:threshold 10
```

Exclude any string by regular expression with format `/pattern/`, the pattern syntax is same as module `re`. For example, exclude all strings length > 1000:

```
$ pyarmor cfg mix.str:excludes "/.{1000,}/"
```

Append new ruler to exclude 2 words `__main__` and `xyz`:

```
$ pyarmor cfg mix.str:excludes ^ "__main__ xyz"
```

Reset exclude ruler:

```
$ pyarmor cfg mix.str:excludes = ""
```

Encrypt only string length between 8 and 32 by regular expression:

```
$ pyarmor cfg mix.str:includes = "/.{8,32}/"
```

Check trace log to find which strings are protected.

Filter assert function and import

--assert-call and *--assert-import* could protect function and module, but sometimes it may make mistakes.

One case is that pyarmor asserts a third-party function is obfuscated, thus the obfuscated scripts always raise protection error.

Adding an assert rule to fix this problem. For example, tell *--assert-import* ignore module `json` and `inspect` by word list:

```
$ pyarmor cfg assert.import:excludes = "json inspect"
```

Tell *--assert-call* ignore all the function starts with `wintype_` by regular expression:

```
$ pyarmor cfg assert.call:excludes "/wintype_.*/"
```

The other case is that some functions or modules are obfuscated, but pyarmor doesn't protect them. refer to next section *Patching source by inline marker* to fix this issue.

Patching source by inline marker

Before obfuscating a script, Pyarmor scans each line, remove inline marker plus the following one white space, leave the rest as it is.

The default inline marker is `# pyarmor:`, any comment line with this prefix will be as a inline marker.

For example, these lines

```
print('start ...')

# pyarmor: print('this script is obfuscated')
# pyarmor: check_something()
```

will be changed to

```
print('start ...')

print('this script is obfuscated')
check_something()
```

One real case: protecting hidden imported modules

By default *--assert-import* could only protect modules imported by statement `import`, it doesn't handle modules imported by other methods.

For example,

```
m = __import__('abc')
```

In obfuscated script, there is a builtin function *__assert_armored__()* could be used to check `m` is obfuscated. In order to make sure `m` could not be replaced by others, check it manually:

```
m = __import__('abc')
__assert_armored__(m)
```

But this results in a problem, The plain script could not be run because `__assert_armored__` is only available in the obfuscated script.

The inline marker is right solution for this case. Let's make a little change

```
m = __import__('abc')
# pyarmor: __assert_armored__(m)
```

By inline marker, both the plain script and the obfuscated script work as expected.

Sometimes *--assert-call* may miss some functions, in this case, using inline marker to protect them. Here is an example to protect extra function `self.foo.meth`:

```
# pyarmor: __assert_armored__(self.foo.meth)
self.foo.meth(x, y, z)
```

Internationalization runtime error message

Create `messages.cfg` in the path `.pyarmor`:

```
$ mkdir .pyarmor
$ vi .pyarmor/message.cfg
```

It's a `.ini` format file, add a section `runtime.message` with option `languages`. The language code is same as environment variable `LANG`, assume we plan to support 2 languages, and only customize 2 errors:

- error_1: license is expired
- error_2: license is not for this machine

```
[runtime.message]

languages = zh_CN zh_TW

error_1 = invalid license
error_2 = invalid license
```

`invalid license` is default message for any non-matched language.

Now add 2 extra sections `runtime.message.zh_CN` and `runtime.message.zh_TW`

```
[runtime.message]

languages = zh_CN zh_TW

error_1 = invalid license
error_2 = invalid license

[runtime.message.zh_CN]

error_1 =
error_2 =

[runtime.message.zh_TW]

error_1 =
error_2 =
```

Then obfuscate script again to make it works.

When obfuscated scripts start, it checks *LANG* to get current language code. If this language code is not `zh_CN` or `zh_TW`, default message is used.

PYARMOR_LANG could force the obfuscated scripts to use specified language. If it's set, the obfuscated scripts ignore *LANG*. For example, force the obfuscated script `dist/foo.py` to use lang `zh_TW` by this way:

```
export PYARMOR_LANG=zh_TW
python dist/foo.py
```

Generating cross platform scripts

New in version 8.1.

Here list all the standard *platform* names.

In order to generate scripts for other platform, use *--platform* specify target platform. For example, building scripts for windows.x86_64 in Darwin:

```
$ pyarmor gen --platform windows.x86_64 foo.py
```

pyarmor.cli.runtime provides prebuilt binaries for these platforms. If it's not installed, pyarmor may complain of `cross platform need pyarmor.cli.runtime, please run "pip install pyarmor.cli.runtime~=2.1.0" first`. Following the hint to install pyarmor.cli.runtime with the right version.

Using *--platform* multiple times to support multiple platforms. For example, generate the scripts to run in most of x86_64 platforms:

```
$ pyarmor gen --platform windows.x86_64
              --platform linux.x86_64 \
              --platform darwin.x86_64 \
              foo.py
```

Obfuscating scripts for multiple Pythons

New in version 8.x: This feature is still not implemented

3.1.5 Customization and Extension

Contents

- *Changing runtime package name*
- *Appending assert functions and modules*
- *Using plugin to fix loading issue in darwin*
- *Using hook to bind script to docker id*
- *Using hook to check network time by other service*
- *Protecting extension module pyarmor_runtime*
- *Comments within outer key*

Pyarmor provides the following ways to extend:

- Using *pyarmor cfg* to change default configurations
- Using *plugin script* to customize all generated files
- Using *hook script* to extend features in obfuscated scripts

Changing runtime package name

New in version 8.2:[1]

By default the runtime package name is `pyarmor_runtime_xxxxxx`

This name is variable with any valid package name. For example, set it to `my_runtime`:

[1] Pyarmor trial version could not change runtime package name

```
pyarmor cfg package_name_format "my_runtime"
```

Appending assert functions and modules

New in version 8.2.

Pyarmor 8.2 introduces configuration item `auto_mode` to protect more functions and modules. The default value is `and`, *`--assert-call`* and *`--assert-import`* only protect modules and functions which Pyarmor make sure they're obfuscated.

If set its value to `or`, then all the names in the configuration item `includes` are also protected. For example, appending function `foo koo` to assert list:

```
$ pyarmor cfg ast.call:auto_mode "or"
$ pyarmor cfg ast.call:includes "foo koo"

$ pyarmor gen --assert-call foo.py
```

For example, also protect hidden imported module `joker.card`:

```
$ pyarmor cfg ast.import:auto_mode "or"
$ pyarmor cfg ast.import:includes "joker.card"

$ pyarmor gen --assert-import joker/
```

Using plugin to fix loading issue in darwin

New in version 8.2.

In darwin, if Python is not installed in the standard path, the obfuscated scripts may not work because *extension module* `pyarmor_runtime` in the *runtime package* could not be loaded.

Let's check the dependencies of `pyarmor_runtime.so`:

```
$ otool -L dist/pyarmor_runtime_000000/pyarmor_runtime.so

dist/pyarmor_runtime_000000/pyarmor_runtime.so:

    pyarmor_runtime.so (compatibility version 0.0.0, current version 1.0.0)
    ...
    @rpath/lib/libpython3.9.dylib (compatibility version 3.9.0, current version 3.9.0)
    ...
```

Suppose *target device* has no `@rpath/lib/libpython3.9.dylib`, but `@rpath/lib/libpython3.9.so`, in this case `pyarmor_runtime.so` could not be loaded.

We can create a plugin script `.pyarmor/myplugin.py` to fix this problem

```
__all__ = ['CondaPlugin']

class CondaPlugin:

    def _fixup(self, target):
        from subprocess import check_call
        check_call('install_name_tool -change @rpath/lib/libpython3.9.dylib @rpath/
↪lib/libpython3.9.so %s' % target)
```

(continues on next page)

(continued from previous page)

```
        check_call('codesign -f -s - %s' % target)

    @staticmethod
    def post_runtime(ctx, source, target, platform):
        if platform.startswith('darwin.'):
            print('using install_name_tool to fix %s' % target)
            self._fixup(target)
```

Enable this plugin and generate the obfuscated script again:

```
$ pyarmor cfg plugins + "myplugin"
$ pyarmor gen foo.py
```

See also:

Plugins

Using hook to bind script to docker id

New in version 8.2.

Suppose we need bind script `app.py` to 2 dockers which id are `docker-a1` and `docker-b2`

First create hook script `.pyarmor/hooks/app.py`

```
def _pyarmor_check_docker():
    cid = None
    with open("/proc/self/cgroup") as f:
        for line in f:
            if line.split(':', 2)[1] == 'name=systemd':
                cid = line.strip().split('/')[-1]
                break

    docker_ids = __pyarmor__(0, None, b'keyinfo', 1).decode('utf-8')
    if cid is None or cid not in docker_ids.split(','):
        raise RuntimeError('license is not for this machine')

_pyarmor_check_docker()
```

Then generate the obfuscated script, store docker ids to *runtime key* as private data at the same time:

```
$ pyarmor gen --bind-data "docker-a1,docker-b2" app.py
```

Run the obfuscated script to check it, please add print statements in the hook script to debug it.

See also:

Hooks `__pyarmor__()`

Using hook to check network time by other service

New in version 8.2.

If NTP is not available in the *target device* and the obfuscated scripts has expired date, it may raise `RuntimeError: Resource temporarily unavailable`.

In this case, using hook script to verify expired data by other time service.

First create hook script in the `.pyarmor/hooks/foo.py`:

```
def _pyarmor_check_worldtime(host, path):
    from http.client import HTTPSConnection
    expired = __pyarmor__(1, None, b'keyinfo', 1)
    conn = HTTPSConnection(host)
    conn.request("GET", path)
    res = conn.getresponse()
    if res.code == 200:
        data = res.read()
        s = data.find(b'"unixtime":')
        n = data.find(b',', s)
        current = int(data[s+11:n])
        if current > expire:
            raise RuntimeError('license is expired')
     else:
         raise RuntimeError('got network time failed')
_pyarmor_check_worldtime('worldtimeapi.org', '/api/timezone/Europe/Paris')
```

Then generate script with local expired date:

```
$ pyarmor gen -e .30 foo.py
```

Thus the obfuscated script could verify network time by itself.

See also:

Hooks __pyarmor__()

Protecting extension module pyarmor_runtime

New in version 8.2.

This example shows how to check the file content of an extension module to make sure it's not changed by others.

First create a hook script `.pyarmor/hooks/foo.py`:

```
1 def check_pyarmor_runtime(value):
2     from pyarmor_runtime_000000 import pyarmor_runtime
3     with open(pyarmor_runtime.__file__, 'rb') as f:
4         if sum(bytearray(f.read())) != value:
5             raise RuntimeError('unexpected %s' % filename)
6
7 check_pyarmor_runtime(EXCEPTED_VALUE)
```

Line 7 `EXCEPTED_VALUE` need to be replaced with real value, but it doesn't work to get the sum value of `pyarmor_runtime.so` after building, because each build the sum value is different. We need use a post-runtime plugin to get the expected value and update the hook script automatically

```
# Plugin script: .pyarmor/myplugin.py

__all__ = ['CondaPlugin', 'RuntimePlugin']

class RuntimePlugin:

    @staticmethod
    def post_runtime(ctx, source, target, platform):
        with open(target, 'rb') as f:
```

(continues on next page)

(continued from previous page)

```
            value = sum(bytearray(f.read()))
        with open('.pyarmor/hooks/foo.py', 'r') as f:
            source = f.read()
        source = source.replace('EXPECTED_VALUE', str(value))
        with open('.pyarmor/hooks/foo.py', 'r') as f:
            f.write(source)

class CondaPlugin:
    ...
```

Then enable this plugin:

```
$ pyarmor cfg plugins + "myplugin"
```

Finally generate the obfuscated script, and verify it:

```
$ pyarmor gen foo.py
$ python dist/foo.py
```

This example is only guide how to do, it's not safe enough to use it directly. There is always a way to bypass open source check points, please write your private check code. There are many other methods to prevent binary file from hacking, please learn and search these methods by yourself.

See also:

Hooks

Comments within outer key

New in version 8.2.

The *outer key* ignores all the printable text at the header, so it's possible to insert some readable text in the *outer key* as comments.

Post-key plugin is designed to do this. The following example plugin will print all the key information in the console, and write expired date to outer key file:

```
# Plugin script: .pyarmor/myplugin.py

from datetime import datetime

__all__ = ['CommentPlugin']

class CommentPlugin:

    @staticmethod
    def post_key(ctx, keyfile, **keyinfo):
        expired = None
        for name, value in keyinfo.items():
            print(name, value)
            if name == 'expired':
               expired = datetime.fromtimestamp(value).isoformat()

        if expired:
            print('patching runtime key')
            comment = '# expired date: %s\n' % expired
```

(continues on next page)

(continued from previous page)

```
            with open(keyfile, 'rb') as f:
                keydata = f.read()
            with open(keyfile, 'wb') as f:
                f.write(comment.encode())
                f.write(keydata)
```

Enable this plugin and generate an outer key:

```
$ pyarmor cfg plugins + "myplugin"
$ pyarmor gen key -e 2023-05-06
```

Check comment:

```
$ head -n 1 dist/pyarmor.rkey
```

See also:

Plugins

3.2 How To

3.2.1 Highest security and performance

Contents

- *What's the most security pyarmor could do?*
- *What's the best performance pyarmor could do?*
- *Recommended options for different applications*
- *Reforming scripts to improve security*

What's the most security pyarmor could do?

The following options could improve security

- `--enable-rft` almost doesn't impact performance
- `--enable-bcc` even a little faster than plain script, but consume more memory to load binary code
- `--enable-jit` prevents from static decompilation
- `--enable-themida` prevents from most of debuggers, only available in Windows, and reduce performance remarkable
- `--mix-str` protects string constant in the script
- `pyarmor cfg mix_argnames=1` may broken annotations
- `--obf-code 2` could make more difficult to reverse byte code

The following options hide module attributes

- `--private` for script or `--restrict` for package

The following options prevent functions or modules from replaced by hack code

- *--assert-call*
- *--assert-import*

What's the best performance pyarmor could do?

Using default options and the following settings

- *--obf-code* 0
- *--obf-module* 0
- `pyarmor cfg restrict_module=0`

By these options, the security is almost same as *.pyc*

In order to improve security, and doesn't reduce performance, also enable RFT mode

- *--enable-rft*

If there are sensitive string, enable mix-str with filter

- `pyarmor cfg mix.str:includes "/regular expression/"`
- *--mix-str*

Without filter, all of string constants in the scripts are encrypted, it may reduce performance. Using filter only encrypt the sensitive string may balance security and performance.

Recommended options for different applications

For Django application or serving web request

If RFT mode is safe enough, you can check the transformed scripts to make decision, using these options

- *--enable-rft*
- *--obf-code* 0
- *--obf-module* 0
- *--mix-str* with filter

If RFT mode is not safe enough, using these options

- *--enable-rft*
- *--no-wrap*
- *--mix-str* with filter

For most of applications and packages

If RFT mode and BCC mode are available

- *--enable-rft*
- *--enable-bcc*
- *--mix-str* with filter
- *--assert-import*

If RFT mode and BCC mode are not available

- *--enable-jit*
- *--private* for scripts, or *--restrict* for packages
- *--mix-str* with filter
- *--assert-import*
- *--obf-code* 2

If care about monkey trick, also

- *--assert-call* with inline marker to make sure all the key functions are protected

If it's not performance sensitive, using *--enable-themida* prevent from debuggers

Reforming scripts to improve security

Move main script module level code to other module

Pyarmor will clear the module level code after the module is imported, the injected code could not get any module level code because it's gone.

But the main script module level code is never cleared, so moving un-necessary code here to other module could improve security.

3.2.2 Protecting Runtime Memory Data

Pyarmor focus on protecting Python scripts, by several irreversible obfuscation methods, now Pyarmor makes sure the obfuscated scripts can't be restored by any way.

But it isn't good at memory protection and anti-debug. If you care about runtime memory data, or runtime key verification, generally it need extra methods to prevent debugger from hacking dynamic libraries.

Pyarmor could prevent hacker from querying runtime data by valid Python C API and other Python ways, only if Python interpreter and extension module `pyarmor_runtime` are not hacked. This is what extra tools need to protect, the common methods include

- Signing the binary file to make sure they're not changed by others
- Using third-party binary protection tools to protect Python interpreter and extension module `pyarmor_runtime`
- Pyarmor provides some configuration options to check interps and debuggers.
- Pyarmor provides runtime patch feature to let expert users to write C functions or python scripts to improve security.

Basic steps

Above all, Python interpreter to run the obfuscated scripts can't be replaced, if the obfuscated scripts could be executed by patched Python interpreter, it's impossible to prevent others to read any Python runtime data.

At this time Pyarmor need combine *--pack* and restrict mode options to implement this.

First pack the script by PyInstaller[1]:

```
$ pyinstaller foo.py
```

[1] If pack to one file by PyInstaller, it's not enough to protect this file alone. You must make sure all the binary files extracted from this file are protected too.

Next configure and repack the bundle, the following options are necessary[2]:

```
$ pyarmor cfg check_debugger=1 check_interp=1
$ pyarmor gen --mix-str --assert-call --assert-import --restrict --pack dist/foo/foo␣
↪foo.py
```

Then protect all the binary files in the output path `dist/foo/` by external tools, make sure these binary files could not be replaced or modified in runtime.

Available external tools: codesign, VMProtect

Note

Hook Scripts

Expert users could write *hook script* to check PyInstaller bootstrap modules to improve security.

Here it's an example to show how to do, note that it may not work in different PyInstaller version, do not use it directly.

```
# Hook script ".pyarmor/hooks/foo.py"

def protect_self():
    from sys import modules

    def check_module(name, checklist):
        m = modules[name]
        for attr, value in checklist.items():
            if value != sum(getattr(m, attr).__code__.co_code):
                raise RuntimeError('unexpected %s' % m)

    checklist__frozen_importlib = {}
    checklist__frozen_importlib_external = {}
    checklist_pyimod03_importers = {}

    check_module('_frozen_importlib', checklist__frozen_importlib)
    check_module('_frozen_importlib_external', checklist__frozen_importlib_external)
    check_module('pyimod03_importers', checklist_pyimod03_importers)

protect_self()
```

The highlight lines need to be replaced with real check list. In order to get baseline, first replace function `check_module` with this fake function

```
def check_module(name, checklist):
    m = modules[name]
    refs = {}
    for attr in dir(m):
        value = getattr(m, attr)
        if hasattr(value, '__code__'):
            refs[attr] = sum(value.__code__.co_code)
    print('    checklist_%s = %s' % (name, refs))
```

Run the following command to get baseline:

```
$ pyinstaller foo.py
$ pyarmor gen --pack dist/foo/foo foo.py
```

(continues on next page)

[2] Do not use `check_interp` in 32-bit x86 platforms, it doesn't work

(continued from previous page)

```
...
checklist__frozen_importlib = {'__import__': 9800, ...}
checklist__frozen_importlib_external = {'_calc_mode': 2511, ...}
checklist_pyimod03_importers = {'imp_lock': 183, 'imp_unlock': 183, ...}
```

Edit hook script to restore `check_module` and replace empty check lists with real ones.

Using this real hook script to generate the final bundle:

```
$ pyinstaller foo.py
$ pyarmor gen --pack dist/foo/foo foo.py
```

Runtime Patch

New in version 8.x: It's not implemented.

Pyarmor provides runtime patch feature so that users could write one C or python script to do any anti-debug or other checks. It will be embedded into *runtime files*, and called on extension module `pyarmor_runtime` initialization.

The idea is to make a file `.pyarmor/hooks/pyarmor_runtime.py` or `.pyarmor/hooks/pyarmor_runtime.c`, it will be inserted into runtime files when building obfuscated scripts.

3.2.3 Packing with outer key

This example shows how to pack `src/myapp.py` with *outer key*

First pack it by PyInstaller:

```
$ pyinstaller myapp.py
```

Next obfuscate the script with outer key:

```
$ pyarmor gen --outer --pack dist/myapp/myapp myapp.py
```

Then generate an outer key:

```
$ pyarmor gen key -O keylist -e 30
```

For one-folder mode, generally save outer key in the runtime package. For example:

```
$ cp keylist/pyarmor.rkey dist/myapp/pyarmor_runtime_000000/
```

Thus it could run `dist/myapp/myapp` in any path. For example:

```
$ dist/myapp/myapp
```

For one-file mode, generally store outer key to the same path of executable, and rename it to `EXECUTABLE.KEYNAME`. For example:

```
$ pyinstaller --onefile myapp.py
$ pyarmor gen --outer --pack dist/myapp myapp.py
$ pyarmor gen key -O keylist -e 30
$ cp keylist/pyarmor.rkey dist/myapp.pyarmor.rkey
```

Thus it could run `dist/myapp` in any path. For example:

```
$ dist/myapp
```

The outer key also could be stored in a fixed path specified by *PYARMOR_RKEY*. For example:

```
$ export PYARMOR_RKEY=/opt/pyarmor/runtime_data
$ mkdir -p /opt/pyarmor/runtime_data
$ cp keylist/pyarmor.rkey /opt/pyarmor/runtime_data/
$ dist/foo
```

3.2.4 Building obfuscated wheel

The test-project hierarchy is as follows:

```
$ tree test-project

test-project
├── MANIFEST.in
├── pyproject.toml
├── setup.cfg
└── src
    └── parent
        ├── child
        │   └── __init__.py
        └── __init__.py
```

4 directories, 5 files

The content of `MANIFEST.in` is:

```
recursive-include dist/parent/pyarmor_runtime_00xxxx *.so
```

The content of `pyproject.toml` is:

```
[build-system]
    requires = [
        "setuptools>=66.1.1",
        "wheel"
    ]
    build-backend = "setuptools.build_meta"
```

The content of `setup.cfg` is:

```
[metadata]
    name = parent.child
    version = attr: parent.child.VERSION

[options]
    package_dir =
        =dist/

    packages =
        parent
        parent.child
        parent.pyarmor_runtime_00xxxx

    include_package_data = True
```

`src/parent/__init__.py` and `src/parent/child/__init__.py` are the same:

```
VERSION = '0.0.1'
```

First obfuscate the package:

```
$ cd test-project
$ pyarmor gen --recursive -i src/parent
```

After successful execution the output is the following directory:

```
$ tree dist

dist
└── parent
    ├── child
    │   ├── __init__.py
    │   └── __pycache__
    │       └── __init__.cpython-311.pyc
    ├── __init__.py
    └── pyarmor_runtime_00xxxx
        ├── __init__.py
        └── pyarmor_runtime.so
```

Next, build the wheel package:

```
$ python -m build --skip-dependency-check --no-isolation
```

Unfortunately it raises exception:

```
* Building sdist...
Traceback (most recent call last):
  File "/usr/lib/python3/dist-packages/setuptools/config/expand.py", line 81, in __
↪getattr__
    return next(
           ^^^^^
StopIteration

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "/usr/lib/python3/dist-packages/setuptools/config/expand.py", line 191, in␣
↪read_attr
    return getattr(StaticModule(module_name, spec), attr_name)
```

From traceback we found it uses `StaticModule`, then check the source `/usr/lib/python3/dist-packages/setuptools/config/expand.py` at line 191 to find class `StaticModule` definition. By the source code we know it uses `ast.parse` to parse source code directly to get locals. It's impossible for obfuscated scripts, in order to fix this problem, we need insert a line in the `dist/parent/child/__init__.py` like this:

```
from pyarmor_runtime_00xxxx import __pyarmor__
VERSION = '0.0.1'
...
```

But pyarmor doesn't allow to change obfuscated scripts by default, it need disable this restriction by this command:

```
$ pyarmor cfg -p parent.child.__init__ restrict_module = 0
$ pyarmor gen --recursive -i src/parent
```

The option *pyarmor cfg -p* `parent.child.__init__` lets pyarmor disable this restriction only for `parent/child/__init__.py`.

Now patch `dist/parent/child/__init__.py` and rebuild wheel:

```
$ python -m build --skip-dependency-check --no-isolation
```

Rename runtime package and store it in sub-package

If you would rather to rename runtime package to `libruntime` and store it in the sub-package `parent.child`, you need change the content of `MANIFEST.in` to:

```
recursive-include dist/parent/child/libruntime *.so
```

and change the content of `setup.cfg` to:

```
[options]
    ...
    packages =
        parent
        parent.child
        parent.child.libruntime
    ...
```

And obfuscate the scripts by these configurations:

```
$ pyarmor cfg package_name_format "libruntime"
$ pyarmor gen --recursive --prefix parent.child src/parent
```

Don't forget to patch `dist/parent/child/__init__.py`, then build wheel:

```
$ python -m build --skip-dependency-check --no-isolation
```

Further more

In order to patch `dist/parent/child/__init__.py` automatically, you can write a plugin script `.pyarmor/myplugin.py`:

```
__all__ = ['VersionPlugin']

class VersionPlugin:

    @staticmethod
    def post_build(ctx, inputs, outputs, pack):
        script = os.path.join(outputs[0], 'parent', 'child', '__init__.py')
        with open(script, 'a') as f:
            f.write("\nVERSION = '0.0.1'")
```

And enable this plugin:

```
$ pyarmor cfg plugins + "myplugin"
```

After that, each build only run the following commands:

```
$ pyarmor gen --recursive --prefix parent.child src/parent
$ python -m build --skip-dependency-check --no-isolation
```

3.2.5 Protecting system packages

New in version 8.2.

When packing the scripts, Pyarmor could also protect system packages in the bundle. These are necessary options to prevent system packages from be replaced by plain scripts:

```
$ pyinstaller foo.py
$ pyarmor gen --assert-call --assert-import --restrict --pack dist/foo/foo foo.py
```

See also:

Protecting Runtime Memory Data

3.2.6 Fix encoding error

The default encoding is `utf-8`, if encoding error occurs when obfuscating the scripts, set encoding to right one. For example, change default encoding to `gbk`:

```
$ pyarmor cfg encoding=gbk
```

When customizing runtime error message, it also could specify encoding for `messages.cfg`. For example, set encoding to `gbk` by this command:

```
$ pyarmor cfg messages=messages.cfg:gbk
```

3.2.7 Removing docstring

It's easy to remove docstring from obfuscated scripts:

```
$ pyarmor cfg optimize 2
```

The argument optimize specifies the optimization level of the compiler; the default value of -1 selects the optimization level of the interpreter as given by -O options. Explicit levels are 0 (no optimization; __debug__ is true), 1 (asserts are removed, __debug__ is false) or 2 (docstrings are removed too).

3.2.8 Work with Third-Party Libraries

Contents

- *Third party libraries*
  - *pandas*
  - *nuitka*

There are countless big packages in Python world, many packages I never use and even don't know at all. It's also not easy for me to research a complex package to find which line conflicts with pyarmor, and it's difficult for me to run all of these complex packages in my local machine.

Pyarmor provides rich options to meet various needs, for complex application, please spend some time to check *Man Page* to understand all of these options, one of them may be just for your problem. **I won't learn your application and tell you should use which options**

I'll improve pyarmor make it works with other libraries as far as possible, but some issues can't be fixed from Pyarmor side.

Generally most of problems for these third party libraries are

- they try to use low level object *frame* to get local variable or other runtime information of obfuscated scripts
- they try to visit code object directly to get something which is just pyarmor protected. The common case is using `inspect` to get source code.
- they pickle the obfuscated code object and pass it to other processes or threads.

Also check *The differences of obfuscated scripts*, if third party library use any feature changed by obfuscated scripts, it will not work with pyarmor. Especially for *BCC mode*, it changes more.

The common solutions to fix third-party libraries issue

- Use RFT mode with `--obf-code=0`

  RFT mode almost doesn't change internal structure of code object, it transforms the script in source level. *`--obf-code`* is also required to disable code object obfuscation. The recommended options are like this:

```
$ pyarmor gen --enable-rft --obf-code 0 /path/to/myapp
```

  First make sure it works, then try other options. For example:

```
$ pyarmor gen --enable-rft --obf-code 0 --mix-str /path/to/myapp
$ pyarmor gen --enable-rft --obf-code 0 --mix-str --assert-call /path/to/myapp
```

- Ignore problem scripts

  If only a few scripts are in trouble, try to obfuscate them with `--obf-code 0`. For example, only module `config.py` has problem, all the other are fine, then:

```
$ pyarmor cfg -p myapp.config obf_code=0
$ pyarmor gen [other options] /path/to/myapp
```

  Another way is to copy plain script to overwrite the obfuscated one roughly:

```
$ pyarmor gen [other options] /path/to/myapp
$ cp /path/to/myapp/config.py dist/myapp/config.py
```

- Patch third-party library

  Here is an example

```
@cherrypy.expose(alias='myapi')
   @cherrypy.tools.json_out()
   # pylint: disable=no-member
   @cherrypy.tools.authenticate()
   @cherrypy.tools.validateOptOut()
   @cherrypy.tools.validateHttpVerbs(allowedVerbs=['POST'])
   # pylint: enable=no-member
```

(continues on next page)

(continued from previous page)

```
    def abc_xyz(self, arg1, arg2):
        """
        This is the doc string
        """
```

If call this API with alias name "myapi" it throws me 404 Error and the API's which do not have any alias name works perfectly. Because `cherrypy.expose` decorator uses

```
parents = sys._getframe(1).f_locals
```

And `sys._getframe(1)` return unexpected frame in obfuscated scripts. But it could be fixed by patching this decorator to

```
parents = sys._getframe(2).f_locals
```

Note: If `cheerypy` is also used by others, clone private one.

Third party libraries

Here are list problem libraries and possible solutions. Welcome create pull request to append new libraries sort alphabetically case insensitivity.

Table 1: Table-1. Third party libraries

Package	Status	Remark
cherrypy	patch work[1]	use sys._getframe
pandas	patch work[1]	use sys._getframe
playwright	patch should work[2]	Not verify yet
nuitka	Should work with restrict_module = 0	Not verify yet

pandas

Another similar example is `pandas`

```
import pandas as pd

class Sample:
    def __init__(self):
        self.df = pd.DataFrame(
            data={'name': ['Alice', 'Bob', 'Dave'],
            'age': [11, 15, 8],
            'point': [0.9, 0.1, 0.4]}
        )

    def func(self, val: float = 0.5) -> None:
        print(self.df.query('point > @val'))

sampler = Sample()
sampler.func(0.3)
```

[1] the patched package could work with Pyarmor
[2] this package work with Pyarmor RFT mode

After obfuscated, it raises:

```
pandas.core.computation.ops.UndefinedVariableError: local variable 'val' is not␣
↪defined
```

It could be fixed by changing `sys._getframe(self.level)` to `sys._getframe(self.level+1)`, `sys._getframe(self.level+2)` or `sys._getframe(self.level+3)` in `scope.py` of pandas.

nuitka

Because the obfuscated scripts could be taken as normal scripts with an extra runtime package, they also could be translated to C program by Nuitka.

I haven't tested it, but it's easy to verify it.

First disable restrict mode:

```
$ pyarmor cfg restrict_module=0
```

No disable restrict_module, run the nuitka script may raise `RuntimeError: unauthorized use of script`

Next use default options to obfuscate the scripts:

```
$ pyarmor gen foo.py
```

Finally nuitka the obfuscated script `dist/foo.py`, check it works or not.

Try more options, but I think restrict options such as *`--private`*, *`--restrict`*, *`--assert-call`*, *`--assert-import`* may not work.

3.2.9 Using Pyarmor License

Contents

- *Prerequisite*
- *Using Pyarmor Basic or Pro*
  - *Initial registration*
  - *Product name is not decided*
  - *Registering in other machines*
  - *Registering in Docker or CI pipeline*
- *Using group license*
  - *Initial registration*
  - *Group device file*
  - *Generating offline device regfile*
  - *Registering Pyarmor in offline device*
- *Upgrading old Pyarmor license*

Prerequisite

First of all

1. An *activation file* of *Pyarmor License* like `pyarmor-regcode-xxxx.txt`, refer to *License Types* to purchase right one
2. Pyarmor 8.2+
3. Internet connection
4. Product name bind to this license, for non-commercial use, product name is `non-profits`

Using Pyarmor Basic or Pro

Basic use steps:

1. Using *activation file* to initial registration, set product name bind to this license
2. Once initial registration completed, a *registration file* is generated
3. Using *registration file* to register Pyarmor in other devices

Initial registration

Using *-p* to specify product name for this license, for non-commercial use, set product name to `non-profits`.

Assume this license is used to protect your product `XXX`, initial registration by this command:

```
$ pyarmor reg -p "XXX" pyarmor-regcode-xxxx.txt
```

Pyarmor will show registration information and ask for your confirmation. If everything is fine, type `yes` and `Enter` to continue. Any other input aborts registration.

If initial registration is successful, it prints final license information in the console. And a *registration file* like `pyarmor-regfile-xxxx.zip` is generated in the current path at the same time. This file is used for subsequent registration in other machines.

Once initial registration completed, activation file `pyarmor-regcode-xxxx.txt` is invalid, do not use it again.

Once initial registration completed, product name can't be changed.

Please backup registration file `pyarmor-regfile-xxxx.zip` carefully. If lost, Pyarmor is not responsible for keeping this license and no lost-found service.

Product name is not decided

When product is in developing, and product name is not decided. Set product name to `TBD` on initial registration. For example:

```
$ pyarmor reg -p "TBD" pyarmor-regcode-xxxx.txt
```

In 6 months real product name must be set by this command:

```
$ pyarmor reg -p "XXX" pyarmor-regcode-xxxx.txt
```

If it's not changed after 6 months, the product name will be set to `non-profits` automatically and can't be changed again.

Registering in other machines

Copy *registration file* `pyarmor-regfile-xxxx.zip` to other machines, run the following command:

```
$ pyarmor reg pyarmor-regfile-xxxx.zip
```

Check the registration information:

```
$ pyarmor -v
```

After successful registration, all obfuscations will automatically apply this license, and each obfuscation requires online license verification.

Registering in Docker or CI pipeline

It's no problem to run Pyarmor in Docker or CI pipeline to obfuscate user's application. Register pyarmor with `pyarmor-regfile-xxxx.zip` same as above. **But It's not allowed to distribute pyarmor self and any Pyarmor License to customer**

Don't run too many build dockers, maximum is 100.

Using group license

New in version 8.2.

Each *Pyarmor Group* could have 100 offline devices, each device has its own number, from 1 to 100.

Basic use steps:

1. Using activate file `pyarmor-regcode-xxxx.txt` to initial registration, set product name bind to this license, and generate *registration file*
2. Generating group device file separately on each offline device
3. Using *registration file* and group device file to generate device registration file.
4. Using device registration file to register Pyarmor on offline device[1]

Initial registration

After purchasing *Pyarmor Group*, an activate file `pyarmor-regcode-xxxx.txt` is sent to registration email.

Initial registration need internet connection and Pyarmor 8.2+. Suppose product name is `XXX`, then run this command:

```
$ pyarmor reg -p XXX pyarmor-regcode-xxxx.txt
```

After initial registration completed, a *registration file* `pyarmor-regfile-xxxx.zip` will be generated.

Group device file

On each offline device, install Pyarmor 8.2+, and generate group device file. For example, on device no. 1, run this command:

[1] The device registration file is bind to specified device, each device has its own device regfile

```
$ pyarmor reg -g 1
```

It will generate group device file `pyarmor-group-device.1`.

Generating offline device regfile

Generating offline device regfile need internet connection, Pyarmor 8.2+, group device file `pyarmor-group-device.1` and group license *registration file* `pyarmor-regfile-xxxx.zip`.

Copying group device file `pyarmor-group-device.1` to initial registration device which has internet connection, this file must be saved in the path `.pyarmor/group/`, then run this command to generate device regfile `pyarmor-device-regfile-xxxx.1.zip`:

```
$ mkdir -p .pyarmor/group
$ cp pyarmor-group-device.1 .pyarmor/group/

$ pyarmor reg -g 1 /path/to/pyarmor-regfile-xxxx.zip
```

Registering Pyarmor in offline device

Once device regfile is generated, copy it to the corresponding device, run this command to register Pyarmor:

```
$ pyarmor reg pyarmor-device-regfile-xxxx.1.zip
```

Check registration information:

```
$ pyarmor -v
```

After successful registration, all obfuscations will automatically apply this group license, and each obfuscation need not online license verification.

Upgrading old Pyarmor license

Refer to *upgrade old license*

3.3 References

3.3.1 Concepts

Activation File A text file used to initial registration *Pyarmor License*

> When purchasing any *Pyarmor License*, an activation file is be sent to registration email after payment is completed.

BCC Mode An obfuscation method of Pyarmor by converting Python functions to C functions

extension module A module written in C or C++, using Python's C API to interact with the core and with user code.

Build Machine The device in which to install pyarmor, and to run pyarmor to generate obfuscated scripts.

Global Path Store Pyarmor global configuration file, default is `~/.pyarmor/config/`

> It's always relative to *Home Path*

Home Path Store Pyarmor registration file, global configuration, other data file generated by **pyarmor**, the default path is in user home path `~/.pyarmor/`

Local Path Store Pyarmor local configuration file, default is in the current path `./.pyarmor/`

Hook script Hook script is a python script which locates in sub-path `hooks` of *local path* or *global path*.

When obfuscating the scripts, if there is any same name script exists, it's called module hook script, and will be inserted into the obfuscated scripts.

The hook script will be executed first when running the obfuscated scripts.

JIT Abbr. JUST-IN-TIME, just generating machine instructions in run time.

Outer Key A file generally named `pyarmor.rkey` to store *Runtime Key*

The outer key file must be located in one of path

- *Runtime package*
- *PYARMOR_RKEY*, no trailing slash or backslash, and no `..` in the path. Generally it's an absolute path, for example, `/var/data`
- Current path

Or a file `sys.executable` + `.pyarmor.rkey`. For example, `dist/myapp.exe.pyarmor.rkey`

Platform The standard platform name defined by Pyarmor. It's composed of os.arch.

Supported platforms list:

- **Windows**
  - windows.x86_64
  - windows.x86
- **Many Linuxes**
  - linux.x86_64
  - linux.x86
  - linux.aarch64
  - linux.armv7
- **Apple Intel and Silicon**
  - darwin.x86_64
  - darwin.aarch64 or darwin.arm64

Plugin script A python script will be called in building stage to do some customization work.

Pyarmor Pyarmor is product domain, the goal is to provide functions and services to obfuscate Python scripts in high security and high performance. The mission of Pyarmor is let Python use easily in commercial product.

Pyarmor is composed of

- *Pyarmor Home*
- *pyarmor package*

Pyarmor Basic A *Pyarmor License* type

Pyarmor Group A *Pyarmor License* type

Pyarmor Home Host in GitHub: https://github.com/dashingsoft/pyarmor/

It serves open source part of Pyarmor, issues and documentations.

Pyarmor License Issued by Pyarmor Team to unlock some limitations in Pyarmor trial version.

Refer to *Pyarmor License Types*

Pyarmor Package A *Python Package*, it includes

- `pyarmor`
- `pyarmor.cli`
- `pyarmor.cli.core`
- `pyarmor.cli.runtime`

Pyarmor Pro A *Pyarmor License* type

Pyarmor Users Developers or organizations who use Pyarmor to obfuscate their Python scripts

Python A program language.

Python Script A file that serves as an organizational unit of Python code.

Refer to https://docs.python.org/3.11/glossary.html#term-module

Python Package Refer to https://docs.python.org/3.11/glossary.html#term-package

Registration File A zip file generated after initial registration is successful. It's used to register *Pyarmor License* except initial registration.

RFT Mode An obfuscation method of Pyarmor by renaming function/class in the scripts

Runtime Files All the files required to run the obfuscated scripts.

Generally it equals *Runtime Package*. If *outer key* is used, plus this outer key file.

Runtime Key The settings of obfuscated scripts. It may include the expired date, device information of bind to obfuscated scripts. Also include all the flags to control the behaviors of obfuscated scripts.

Generally it's embedded into *Runtime Package*, but it also could be stored to an independent file *outer key*

Runtime Package A *Python Package* generally named `pyarmor_runtime_000000`.

When obfuscating the scripts, it's be generated at the same time.

It's required to run the obfuscated scripts.

Target Device In which run the obfuscated scripts distributed by *Pyarmor Users*, generally it's in customer side

3.3.2 Man Page

Contents

- *pyarmor*
- *pyarmor gen*
- *pyarmor gen key*
- *pyarmor cfg*
- *pyarmor reg*

- *Environment Variables*

Pyarmor is a powerful tool to obfuscate Python scripts with rich option set that provides both high-level operations and full access to internals.

pyarmor

Syntax

pyarmor [options] <command> ...

Options

-h, --help	show available command set then quit
-v, --version	show version information then quit
-q, --silent	suppress all normal output ...
-d, --debug	show more information in the console ...
--home PATH	set Pyarmor HOME path ...

These options can be used after **pyarmor** but before command, here are available commands:

gen	Obfuscate scripts
gen key	Generate outer runtime key
cfg	Show and configure environments
reg	Register Pyarmor

See **pyarmor <command> -h** for more information on a specific command.

Description

-q, --silent

Suppress all normal output.

For example:

```
pyarmor -q gen foo.py
```

-d, --debug

Show more information in the console

When something is wrong, print more debug information in the console. For example:

```
pyarmor -d gen foo.py
```

--home PATH[,GLOBAL[,LOCAL[,REG]]]

Set Pyarmor *Home Path*, *Global Path*, *Local Path* and registration file path

The default paths

- *Home Path* is `~/.pyarmor/`
- *Global Path* is `~/.pyarmor/config/`
- *Local Path* is `./.pyarmor/`
- registration file path is same as *Home Path*

All of them could be changed by this option. For example, change home path to `~/.pyarmor2/`:

```
$ pyarmor --home ~/.pyarmor2 ...
```

Then

- *Global Path* is `~/.pyarmor2/config/`
- Registration files are stored in the `~/.pyarmor2/`
- *Local Path* still is `./.pyarmor/`

Another example, keep all others but only change global path to `~/.pyarmor/config2/`:

```
$ pyarmor --home ,config2 ...
```

Another, keep all others but only change local path to `/var/myproject`:

```
$ pyarmor --home ,,/var/myproject/ ...
```

Another, set registration file path to `/opt/pyarmor/`:

```
$ pyarmor --home ,,,/opt/pyarmor ...
```

It's useful when using **`sudo`** to run **`pyarmor`** occasionally. This makes sure the registration file could be found even switch to another user.

When there are many Pyarmor Licenses registered in one machine, set each license to different registration file path. For example:

```
$ pyarmor --home ~/.pyarmor1 reg pyarmor-regfile-2051.zip
$ pyarmor --home ~/.pyarmor1 gen project1/foo.py

$ pyarmor --home ~/.pyarmor2 reg pyarmor-regfile-2052.zip
$ pyarmor --home ~/.pyarmor2 gen project2/foo.py
```

Start pyarmor with clean configuration by setting *Global Path* and *Local Path* to any non-exists path `x`:

```
$ pyarmor --home ,x,x, gen foo.py
```

See also:

PYARMOR_HOME

pyarmor gen

Generate obfuscated scripts and all the required runtime files.

`Syntax`

pyarmor gen <options> <SCRIPT or PATH>

`Options`

-h, --help show option list and help information then quit

-O PATH, --output PATH output path . . .

-r, --recursive search scripts in recursive mode . . .

-e DATE, --expired DATE set expired date . . .

-b DEV, --bind-device DEV bind obfuscated scripts to device . . .

--bind-data DATA store private data to runtime key . . .

--period N	check runtime key periodically . . .
--outer	enable outer runtime key . . .
--platform NAME	cross platform obfuscation . . .
-i	store runtime files inside package . . .
--prefix PREFIX	import runtime package with PREFIX . . .
--obf-module <0,1>	obfuscate whole module (default is 1) . . .
--obf-code <0,1,2>	obfuscate each function (default is 1) . . .
--no-wrap	disable wrap mode . . .
--enable <jit,rft,bcc,themida>	enable different obfuscation features . . .
--mix-str	protect string constant . . .
--private	enable private mode for script . . .
--restrict	enable restrict mode for package . . .
--assert-import	assert module is obfuscated . . .
--assert-call	assert function is obfuscated . . .
--pack BUNDLE	repack bundle with obfuscated scripts . . .

Description

This command is designed to obfuscate all the scripts and packages in the command line. For example:

```
pyarmor gen foo.py
pyarmor gen foo.py goo.py koo.py
pyarmor gen src/mypkg
pyarmor gen src/pkg1 src/pkg2 libs/dbpkg
pyarmor gen -r src/mypkg
pyarmor gen -r main.py src/*.py libs/utils.py libs/dbpkg
```

All the files in the command line will be taken as Python script, because a few scripts has unknown extension but it's still Python script.

All the paths in the command line will be taken as Python Package, package name is set to path's basename, all the `.py` files in this path are package modules. If this package has any sub-package, use `-r` to search recursively.

Do not use `pyarmor gen src/*` to obfuscate a package, it will obfuscate any file in the `src`, even they're not python scripts.

-O `PATH`**, --output** `PATH`

Set the output path for all the generated files, default is `dist`

-r, --recursive

When obfuscating package, search all scripts recursively. No this option, only the scripts in package path are obfuscated.

-i

When obfuscating package, store the runtime files inside package. For example:

```
$ pyarmor gen -r -i mypkg
```

The *runtime package* will be stored inside package `dist/mypkg`:

```
$ ls dist/
...      mypkg/

$ ls dist/mypkg/
...            pyarmor_runtime_000000/
```

Without this option, the output path is like this:

```
$ ls dist/
...      mypkg/
...      pyarmor_runtime_000000/
```

This option can't be used to obfuscate script.

--prefix PREFIX

Only used when obfuscating many packages at the same time and still store the runtime package inside package.

In this case, use this option to specify which package is used to store runtime package. For example:

```
$ pyarmor gen --prefix mypkg src/mypkg mypkg1 mypkg2
```

This command tells pyarmor to store runtime package inside `dist/mypkg`, and make `dist/mypkg1` and `dist/mypkg2` to import runtime package from `mypkg`.

Checking the content of `.py` files in output path to make it clear.

As a comparison, obfuscating 3 packages without this option:

```
$ pyarmor gen -O dist2 src/mypkg mypkg1 mypkg2
```

And check `.py` files in the path `dist2`.

-e DATE, **--expired** DATE
: Expired date of obfuscated scripts.

It supports 4 forms:

- A number stands for valid days
- A date with ISO format `YYYY-MM-DD`
- A leading `.` with above 2 forms

Without leading dot, the obfuscated scripts checks NTP server time. For example:

```
$ pyarmor gen -e 30 foo.py
$ pyarmor gen -e 2022-12-31 foo.py
```

With leading dot, it checks local time. For example:

```
$ pyarmor gen -e .30 foo.py
$ pyarmor gen -e .2022-12-31 foo.py
```

-b DEV, **--bind-device** DEV
: Use this option multiple times to bind multiple machines

Bind obfuscated script to specified device. Now only hard disk serial number, Ethernet address and IPv4 address are available.

For example:

```
$ pyarmor gen -b 128.16.4.10 foo.py
$ pyarmor gen -b 52:38:6a:f2:c2:ff foo.py
$ pyarmor gen -b HXS2000CN2A foo.py
```

Also set 30 valid days for this device:

```
$ pyarmor gen -e 30 -b 128.16.4.10 foo.py
```

Check all of hardware information in this device:

```
$ pyarmor gen -b "128.16.4.10 52:38:6a:f2:c2:ff HXS2000CN2A" foo.py
```

Using this options multiple times means binding many machines. For example, the following command makes the obfuscated scripts could run 2 machines:

```
$ pyarmor gen -b "52:38:6a:f2:c2:ff" -b "f8:ff:c2:27:00:7f" foo.py
```

In case there are more network cards, binding anyone by this form:

```
$ pyarmor gen -b "<2a:33:50:46:8f>" foo.py
```

Bind all network cards by this form:

```
$ pyarmor gen -b "<2a:33:50:46:8f,f0:28:69:c0:24:3a>" foo.py
```

In Linux, it's possible to bind named Ethernet card:

```
$ pyarmor gen -b "eth1/fa:33:50:46:8f:3d" foo.py
```

If there are many hard disks. In Windows, binding anyone by sequence no:

```
$ pyarmor gen -b "/0:FV994730S6LLF07AY" foo.py
$ pyarmor gen -b "/1:KDX3298FS6P5AX380" foo.py
```

In Linux, binding to specify name:

```
$ pyarmor gen -b "/dev/vda2:KDX3298FS6P5AX380" foo.py
```

--bind-data DATA
: DATA may be `@FILENAME` or string

Store any private data to runtime key, then check it in the obfuscated scripts by yourself. It's mainly used with the *hook script* to extend runtime key verification method.

If DATA has a leading `@`, then the rest is a filename. Pyarmor reads the binary data from file, and store into runtime key.

For any other case, DATA is converted to bytes as private data.

--period N

Check *Runtime Key* periodically.

Support units:

- s
- m
- h

The default unit is hour, for example, the following examples are equivalent:

```
$ pyarmor gen --period 1 foo.py
$ pyarmor gen --period 3600s foo.py
$ pyarmor gen --period 60m foo.py
$ pyarmor gen --period 1h foo.py
```

Note: If the obfuscated script enters an infinite loop without call any obfuscated function, it doesn't trigger periodic check.

--outer
Enable *outer key*

It tells the obfuscated scripts find *runtime key* in outer file.

Once this option is specified, *pyarmor gen key* must be used to generate an outer key file and copy to the corresponding path in *target device*. Otherwise the obfuscated scripts will complain of `missing license key to run the script`

The default name of outer key is `pyarmor.rkey`, it can be changed by this command:

```
$ pyarmor cfg outer_keyname=".pyarmor.key"
```

By this command the name of outer key is set to `.pyarmor.key`.

--platform NAME
Specify target platform to run obfuscated scripts.

The name must be one of standard *platform* defined by Pyarmor.

It requires *pyarmor.cli.runtime* to get prebuilt binary libraries of other platforms.

--private
Enable private mode for scripts.

When private mode is enabled, the obfuscated scripts could not be imported by plain script or Python interpreter.

--restrict
Enable restrict mode for package, do not use it to obfuscate scripts.

This option implies *--private*.

When restrict mode is enabled, all the modules except `__init__.py` in the package could not be imported by plain scripts.

For example, obfuscate a restrict package to `dist/joker`:

```
$ pyarmor gen -i --restrict joker
$ ls dist/
...    joker/
```

Then create a plaint script `dist/foo.py`

```
import joker
print('import joker should be OK')
from joker import queens
print('import joker.queens should fail')
```

Run it to verify:

```
$ cd dist
$ python foo.py
... import joker should be OK
... RuntimeError: unauthorized use of script
```

If there are extra modules need to be exported, no restrict this module by private settings. For example, no restrict `joker/queens.py` by this command:

```
$ pyarmor cfg -p "joker.queens" restrict_module=0
```

Then obfuscate the package again.

--obf-module <0,1>
: Enable the whole module obfuscation (default is 1)

--obf-code <0,1,2>
: Enable each function obfuscation (default is 1)

Mode `2` is new in Pyarmor 8.2, more security than `1`.

--no-wrap
: Disable wrap mode

If wrap mode is enabled, when enter a function, it's restored. but when exit, this function will be obfuscated again.

If wrap mode is disabled, once the function is restored, it's never be obfuscated again.

If *--obf-code* is `0`, this option is meaningless.

--enable <jit,rft,bcc,themida>
: Enable different obfuscation features.

--enable-jit

Use *JIT* to process some sensitive data to improve security.

--enable-rft
: Enable *RFT Mode* to obfuscate the script [pro]

--enable-bcc
: Enable *BCC Mode* to obfuscate the script [pro]

--enable-themida
: Use Themida to protect extension module in *runtime package*

  Only works for Windows platform.

--mix-str
: Mix the string constant in scripts [basic]

--assert-call
: Assert function is obfuscated

If this option is enabled, Pyarmor scans each function call in the scripts. If the called function is in the obfuscated scripts, protect it as below, and leave others as it is. For example,

```
def fib(n):
    a, b = 0, 1
    return a, b

print('hello')
fib(n)
```

will be changed to

```
def fib(n):
    a, b = 0, 1
    return a, b

print('hello')
__assert_armored__(fib)(n)
```

The function *__assert_armored__()* is a builtin function in obfuscated script. It checks the argument, if it's an obfuscated function, then returns this function, otherwise raises protection exception.

In this example, `fib` is protected, `print` is not.

--assert-import
Assert module is obfuscated

If this option is enabled, Pyarmor scans each `import` statement in the scripts. If the imported module is obfuscated, protect it as below, and leave others as it is. For example,

```
import sys
import foo
```

will be changed to

```
import sys
import foo
__assert_armored__(foo)
```

The function *__assert_armored__()* is a builtin function in obfuscated script. It checks the argument, if it's an obfuscated module, then return this module, otherwise raises protection exception.

This option neither touches statement `from import`, nor the module imported by function `__import__`.

--pack `BUNDLE`
Repack bundle with obfuscated scripts

Here `BUNDLE` is an executable file generated by PyInstaller

Pyarmor just obfuscates the script first.

Then unpack the bundle.

Next replace all the `.pyc` in the bundle with obfuscated scripts, and append all the *runtime files* to the bundle.

Finally repack the bundle and overwrite the original `BUNDLE`.

pyarmor gen key

Generate *outer key* for obfuscated scripts.

Syntax
pyarmor gen key <options>

Options

-O PATH, --output PATH output path

-e DATE, --expired DATE set expired date

--period N check runtime key periodically

-b DEV, --bind-device DEV bind obfuscated scripts to device

--bind-data store private data to runtime key

Description

This command is used to generate *outer key*, the options in this command have same meaning as in the *pyarmor gen*.

There must be at least one of option `-e` or `-b` for *outer key*.

It's invalid that outer key is neither expired nor binding to a device. For this case, don't use outer key.

By default the outer key is saved to `dist/pyarmor.rkey`. For example:

```
$ pyarmor gen key -e 30
$ ls dist/pyarmor.rkey
```

Save outer key to other path by this way:

```
$ pyarmor gen key -O dist/mykey2 -e 10
$ ls dist/mykey2/pyarmor.rkey
```

By default the outer key name is `pyarmor.rkey`, use the following command to change outer key name to any others. For example, `sky.lic`:

```
$ pyarmor cfg outer_keyname=sky.lic
$ pyarmor gen key -e 30
$ ls dist/sky.lic
```

The outer key must be stored in one of the following paths, the obfuscated script will search it in turn:

1. First search runtime package
2. Next search path *PYARMOR_RKEY*, no trailing slash or backslash, and no `..` in the path. Generally it's an absolute path, for example, `/var/data`
3. Next search current path

If no found in these paths, check file `sys.executable` + `.pyarmor.rkey`. For example, `dist/myapp.exe.pyarmor.rkey`

Still not found raise runtime error and exits.

pyarmor cfg

Configure or show Pyarmor environments

Syntax

pyarmor cfg <options> [OPT[=VALUE]] ...

Options

-h, --help show this help message and exit

-p NAME private settings for special module or package

-g, --global do everything in global settings, otherwise local settings

-r, --reset reset option to default value

--encoding ENCODING specify encoding to read configuration file

Description

Run this command without arguments to show all available options:

```
$ pyarmor cfg
```

Show one exact option `obf_module`:

```
$ pyarmor cfg obf_module
```

Show all options which start with `obf`:

```
$ pyarmor cfg obf*
```

Set option to int value by any of these forms:

```
$ pyarmor cfg obf_module 0
$ pyarmor cfg obf_module=0
$ pyarmor cfg obf_module =0
$ pyarmor cfg obf_module = 0
```

Set option to boolean value:

```
$ pyarmor cfg wrap_mode 0
$ pyarmor cfg wrap_mode=1
```

Set option to string value:

```
$ pyarmor cfg outer_keyname "sky.lic"
$ pyarmor cfg outer_keyname = "sky.lic"
```

Append word to an option. For example, `pyexts` has 2 words `.py .pyw`, append new one to it:

```
$ pyarmor cfg pyexts + ".pym"

Current settings
    pyexts = .py .pyw .pym
```

Remove word from option:

```
$ pyarmor cfg pyexts - ".pym"

Current settings
    pyexts = .py .pyw
```

Append new line to option:

```
$ pyarmor cfg rft_excludes ^ "/win.*/"

Current settings
    rft_excludes = super
        /win.*/
```

Reset option to default:

```
$ pyarmor cfg rft_excludes ""
$ pyarmor cfg rft_excludes=""
$ pyarmor cfg -r rft_excludes
```

Change option `excludes` in the section `finder` by this form:

```
$ pyarmor cfg finder:excludes "ast"
```

If no prefix `finder`, for example:

```
$ pyarmor cfg excludes "ast"
```

Not only option `excludes` in section `finder`, but also in other sections `assert.call`, `mix.str` etc. are changed.

Sections

Section is group name of options, here are popular sections

- finder: how to search scripts
- builder: how to obfuscate scripts, main section
- runtime: how to generate runtime package and runtime key

These are not popular sections

- mix.str: how to filter mix string
- assert.call: how to filter assert function
- assert.import: how to filter assert module
- bcc: how to convert function to C code

-p NAME
: Private settings for special modules in the package

  These modules need different obfuscation options.

All the settings is only applied to specified module *NAME*.

For example, only no restrict modules `joker/__init__.py` and `joker/card.py`:

```
$ pyarmor cfg -p joker.__init__ restrict_module = 0
$ pyarmor cfg -p joker.card restrict_module = 0
$ pyarmor gen -r --restrict joker
```

-g, --global
: Do everything in global settings

Without this option, all the changed settings are stored in *Local Path*, generally it's `./.pyarmor/config`. By this option, everything is stored in *Global Path*, generally it's `~/.pyarmor/config/global`

-r, --reset
: Reset option to default value

pyarmor reg

Register Pyarmor or upgrade Pyarmor license

Syntax
: pyarmor reg [OPTIONS] [FILENAME]

Options

-h, --help	show this help message and exit
-p NAME, --product NAME	license to this product

-u, --upgrade upgrade Pyarmor license

-g ID, --device ID device no. in group license

Arguments

The `FILENAME` must be one of these forms:

- `pyarmor-regcode-xxxx.txt` got by purchasing Pyarmor license
- `pyarmor-regfile-xxxx.zip` got by initial registration with above file

Description

Check the registration information:

```
$ pyarmor -v
```

Initial registration

Initial registration by the following command, replace `NAME` with real product name or `non-profits`:

```
$ pyarmor reg -p NAME pyarmor-regcode-xxxx.txt
```

A *registration file* `pyarmor-regfile-xxxx.zip` will be generated after initial registration completed. Using this file for subsequent registration:

```
$ pyarmor reg pyarmor-regfile-xxxx.zip
```

Upgrading old license

Upgrading old license by the following command, if product name is not same as old license, it's ignored:

```
$ pyarmor reg -p NAME pyarmor-regcode-xxxx.txt
```

A *registration file* `pyarmor-regfile-xxxx.zip` will be generated after upgrade completed. Using this file for subsequent registration:

```
$ pyarmor reg pyarmor-regfile-xxxx.zip
```

Using group license

Pyarmor group also need internet connect to initial registration, and generate the corresponding *registration file*.

One group license could have 100 offline devices, each device has its own number, from 1 to 100.

For each device, first install Pyarmor 8.2+, and generate one device file. For example, run this command in device no. 1 to generate group device file `pyarmor-group-device.1`:

```
$ pyarmor reg -g 1
```

Next prepare to generate device regfile `pyarmor-device-regfile-xxxx.1.zip` for this device.

It requires internet connection, group device file `pyarmor-group-device.1`, group license *registration file*. For example, copy group device file to initial registration machine, save it to path `.pyarmor/group/`, run the following command to generate `pyarmor-device-regfile-xxxx.1.zip`:

```
$ mkdir -p .pyarmor/group
$ cp pyarmor-group-device.1 .pyarmor/group/

$ pyarmor reg -g 1 pyarmor-regfile-xxxx.zip
```

Copy device regfile to device no. 1, then run the following command:

```
$ pyarmor reg pyarmor-device-regfile-xxxx.1.zip
```

Repeat above steps for the rest device no. 2, no. 3 ...

-p NAME, **--product** NAME
: Set product name bind to license

  For non-commercial use, set product name to `non-profits`

When initial registration, use this option to set product name for this license.

It's meaningless to use this option after initial registration.

`TBD` is a special product name. If product name is `TBD` at initial registration, the product name can be changed once in 6 months. If it's still not set after 6 months, the product name will be set to `non-profits` automatically.

For any other product name, it can't be changed any more.

Only *Pyarmor basic* and *Pyarmor pro* could set product name to `TBD`

-u, --upgrade
: Upgrade old license to Pyarmor 8.0 License

Not all the old license could be upgrade to new license, check *License Types*

-g ID, **--device** ID
: specify device no. in group license

  Valid value is from 1 to 100

Environment Variables

The following environment variables only used in *Build Machine* when generating the obfuscated scripts, not in *Target Device*.

PYARMOR_HOME
: Same as *pyarmor --home*

It mainly used in the shell scripts to change Pyarmor settings. If *pyarmor --home* is set, this environment var is ignored.

PYARMOR_PLATFORM
: Set the right *Platform* to run **pyarmor**

It's mainly used in some platforms Pyarmor could not tell but still works.

PYARMOR_CC
: Specify C compiler for *BCC mode*

PYARMOR_CLI
: Only for compatible with Pyarmor 7.x, ignore this if you don't use old command prior to 8.0

If you do not use new commands in Pyarmor 8.0, and prefer to only use old commands, set it to `7`, for example:

```
# In Linux
export PYARMOR_CLI=7
pyarmor -h

# Or
PYARMOR_CLI=7 pyarmor -h

# In Windows
```

(continues on next page)

(continued from previous page)

```
set PYARMOR_CLI=7
pyarmor -h
```

It forces command **pyarmor** to use old cli directly.

Without it, **pyarmor** only recognizes new Pyarmor 8 commands.

This only works for command **pyarmor**.

3.3.3 Building Environments

Command **pyarmor** runs in *build machine* to generate obfuscated scripts and all the other required files.

Here list everything related to **pyarmor**.

Above all it only runs in the *supported platforms* by *supported Python versions*.

Command line options, *configuration options*, *plugins*, *hooks* and a few environment variables control how to generate obfuscated scripts and runtime files.

All the command line options and environment variables are described in *Man Page*

Supported Python versions

Table 2: Table-1. Supported Python Versions

Python Version	2.7	3.0~3.4	3.5~3.6	3.7~3.10	3.11	3.12+	Remark
pyarmor 8 RFT Mode	No	No	No	Y	Y	N/y	[1]
pyarmor 8 BCC Mode	No	No	No	Y	Y	N/y	
pyarmor 8 others	No	No	No	Y	Y	N/y	
pyarmor-7	Y	Y	Y	Y	No	No	

Supported platforms

Table 3: Table-2. Supported Platforms

OS	Windows	Apple		Linux			
Arch	x86/x86_64	x86_64	arm64	x86/x86_64	aarch64	armv7	armv6
Themida Protection	Y	No	No	No	No	No	No
pyarmor 8 RFT Mode	Y	Y	Y	Y	Y	Y	No
pyarmor 8 BCC Mode	Y	Y	Y	Y	Y	N/y	No
pyarmor 8 others	Y	Y	Y	Y	Y	Y	No
pyarmor-7[2]	Y	Y	Y	Y	Y	Y	Y

notes

[1] `N/y` means not yet now, but will be supported in future.

[2] pyarmor-7 also supports more linux arches, refer to Pyarmor 7.x platforms.

Important: pyarmor-7 is bug fixed Pyarmor 7.x version, it's same as Pyarmor 7.x, and only works with old license. Do not use it with new license, it may report `HTTP 401 error`.

Configuration options

There are 3 kinds of configuration files

- global: an ini file `~/.pyarmor/config/global`
- local: an ini file `./.pyarmor/config`
- private: each module may has one ini file in *Local Path*. For example, `./.pyarmor/foo.rules` is private configuration of module `foo`

Use command *pyarmor cfg* to change options in configuration files.

Plugins

New in version 8.2.

Plugin is a Python script used to do some post-build work when generating obfuscated scripts.

Plugin use cases:

- Additional processing in the output path
- Fix import statement in the obfuscated script for special cases
- Add comment to *outer key* file
- Rename binary extension `pyarmor_runtime` suffix to avoid name conflicts
- In Darwin use *install_name_tool* to fix *extension module* `pyarmor_runtime` couldn't be loaded if Python is not installed in the standard path
- In Darwin codesign pyarmor runtime extensions

Plugin script must define attribute `__all__` to export plugin name.

Plugin script could be any name.

Plugin script could define one or more plugin classes:

class PluginName

static post_build(*ctx*, *inputs*, *outputs*, *pack=None*)

This method is optional.

This method is called when all the obfuscated scripts and runtime files have been generated by *pyarmor gen*

Parameters

- **ctx** (*Context*) – building context
- **inputs** (*list*) – all the input paths
- **outputs** (*list*) – all the output paths
- **pack** (*str*) – if not None, it's an executable file specified by *--pack*

static post_key (*ctx*, *keyfile*, ***keyinfo*)
This method is optional.

This method is called when *outer key* has been generated by *pyarmor gen key*

Parameters

- **ctx** (*Context*) – building context
- **keyfile** (*str*) – path of generated key file
- **keyinfo** (*dict*) – runtime key information

The possible items in the `keyinfo`:

Key expired expired epoch or None

Key devices a list for binding device hardware information or None

Key data binding data (bytes) or None

Key period period in seconds or None

static post_runtime (*ctx*, *source*, *dest*, *platform*)
This method is optional.

This method is called when the runtime extension module `pyarmor_runtime.so` in the *runtime package* has been generated by *pyarmor gen*.

It may be called many times if many platforms are specified in the command line.

Parameters

- **ctx** (*Context*) – building context
- **source** (*str*) – source path of pyarmor extension
- **dest** (*str*) – output path of pyarmor extension
- **platform** (*str*) – standard *platform* name

To make plugin script work, configure it with script name without extension `.py` by this way:

```
$ pyarmor cfg plugins + "script name"
```

Pyarmor search plugin script in these paths in turn:

- Current path
- *local path*, generally `./.pyarmor/`
- *global path*, generally `~/.pyarmor/`

Here it's an example plugin script `fooplugin.py`

```
__all__ = ['EchoPlugin']

class EchoPlugin:

    @staticmethod
    def post_runtime(ctx, source, dest, platform):
        print('-------- test fooplugin ----------')
        print('ctx is', ctx)
        print('source is', source)
        print('dest is', dest)
        print('platform is', platform)
```

Store it to local path `.pyarmor/fooplugin.py`, and enable it:

```
$ pyarmor cfg plugins + "fooplugin"
```

Check it, this plugin information should be printed in the console:

```
$ pyarmor gen foo.py
```

Disable this plugin:

```
$ pyarmor cfg plugins - "fooplugin"
```

Hooks

New in version 8.2.

Hook is a Python script which is embedded into the obfuscated script, and executed when the obfuscated script is running.

When obfuscating the scripts, Pyarmor searches path `hooks` in the *local path* and *global path* in turn. If there is any same name script exists, it's called module hook script.

For example, `.pyarmor/hooks/foo.py` is hook script of `foo.py`, `.pyarmor/hooks/joker.card.py` is hook script of `joker/card.py`.

When generating obfuscate script by this command:

```
$ pyarmor gen foo.py
```

`.pyarmor/hooks/foo.py` will be inserted into the beginning of `foo.py`.

A hook script is a normal Python script, it could do everything Python could do. And it could use 2 special function *__pyarmor__()* and *__assert_armored__()* to do some interesting work.

Note that all the source lines in the hook script are inserted into module level of original script, be careful to avoid name conflicts.

See also:

__pyarmor__() *__assert_armored__()*

3.3.4 Target Environments

Obfuscated scripts run in *target device*.

Supported Python versions and platforms

Supported platforms, arches and Python versions are same as *Building Environments*

Environment variables

A few environment variables are used by obfuscated scripts.

LANG
: OS environment variable, used to select runtime error language.

PYARMOR_LANG
It's used to set language runtime error language.

If it's set, *LANG* is ignored.

PYARMOR_RKEY
Set search path for *outer key*

Supported Third-Party Interpreter

About third-party interpreter, for example Jython, and any embedded Python C/C++ code, only they could work with CPython *extension module*, they could work with Pyarmor. Check third-party interpreter documentation to make sure this.

A few known issues

- On Linux, *RTLD_GLOBAL* must be set as loading *libpythonXY.so* by *dlopen*, otherwise obfuscated scripts couldn't work.
- Boost::python does not load *libpythonXY.so* with *RTLD_GLOBAL* by default, so it will raise error "No PyCode_Type found" as running obfuscated scripts. To solve this problem, try to call the method *sys.setdlopenflags(os.RTLD_GLOBAL)* as initializing.
- *PyPy* could not work with pyarmor, it's total different from *CPython*
- WASM is not supported.

Specialized builtin functions

New in version 8.2.

There are 2 specialized builtin functions, both of them could be used without import in the obfuscated scripts.

Generally they're used with inline marker or in the hook scripts.

__pyarmor__(*arg*, *kwarg*, *name*, *flag*)

Parameters

- **name** (*bytes*) – must be `b'hdinfo'` or `b'keyinfo'`
- **flag** (*int*) – must be `1`

get hdinfo

When `name` is `b'hdinfo'`, call it to get hardware information.

Parameters

- **arg** (*int*) – query which kind of device
- **kwarg** (*str*) – None or device name

Returns arg == 0 return the serial number of first harddisk

Returns arg == 1 return mac address of first network card

Returns arg == 2 return ipv4 address of first network card

Returns arg == 3 return device name

Return type str

Raises **RuntimeError** – when something is wrong

For example,

```
__pyarmor__(0, None, b'hdinfo', 1)
__pyarmor__(1, None, b'hdinfo', 1)
```

In Linux, `kwarg` is used to get named network card or named hard disk. For example:

```
__pyarmor__(0, "/dev/vda2", b'hdinfo', 1)
__pyarmor__(1, "eth2", b'hdinfo', 1)
```

In Windows, `kwarg` is used to get all network cards and hard disks. For example:

```
__pyarmor__(0, "/0", b'hdinfo', 1)     # First disk
__pyarmor__(0, "/1", b'hdinfo', 1)     # Second disk

__pyarmor__(1, "*", b'hdinfo', 1)
__pyarmor__(1, "*", b'hdinfo', 1)
```

get keyinfo

When `name` is `b'keyinfo'`, call it to query user data in the runtime key.

Parameters

- **arg** (*int*) – what information to get from runtime key
- **kwarg** – always None

Returns arg == 0 return bind data, no bind data return empty bytes

Return type Bytes

Returns arg == 1 return expired epoch, -1 if there is no expired date

Return type Long

Returns None if something is wrong

For example:

```
print('bind data is', __pyarmor__(0, None, b'keyinfo', 1))
print('expired epoch is' __pyarmor__(1, None, b'keyinfo', 1))
```

__assert_armored__(*arg*)

Parameters **arg** (*object*) – arg is a module or callable object

Returns return `arg` if `arg` is obfuscated, otherwise, raise protection error.

For example

```
m = __import__('abc')
__assert_armored__(m)

def hello(msg):
    print(msg)

__assert_armored__(hello)
hello('abc')
```

3.3.5 Error Messages

Here list all the errors when running **pyarmor** or obfuscated scripts.

If something is wrong, search error message here to find the reason.

If no exact error message found, most likely it's not caused by Pyarmor, search it in google or any other search engine to find the solution.

Building Errors

Obfuscating Errors

Table 4: Table-1. Build Errors

Error	Reasons
out of license	Using not available features, for example, big script Purchasing license to unlock the limitations, refer to *License Types*
not machine id	This machine is not registered, or the hardware information is changed. Try to register Pyarmor again to fix it.
query machine id failed	Could not get hardware information in this machine Pyarmor need query hard disk serial number, mac address etc. If it could not get hardware information, it complains of this.
relative import "%s" overflow	Obfuscating *.py* script which uses relative import Solution: obfuscating the whole package (path), instead of one module (file) separately

Registering Errors

Table 5: Table-1.1 Register Errors

Error	Reasons
HTTP Error 400: Bad Request	Please upgrade Pyarmor to 8.2+ to get the exact error message
HTTP Error 401: Unauthorized	Using old pyarmor commands with new license Please using Pyarmor 8 commands to obfuscate the scripts
HTTP Error 503: Service Temporarily Unavailable	Invoking too many register command in 1 minute For security reason, the license server only allows 3 register request in 1 minute
unknown license type OLD	Using old license in Pyarmor 8, the old license only works for Pyarmor 7.x Here are *the latest licenses* Please use `pyarmor-7` or downgrade pyarmor to 7.7.4
This code has been used too many times	If this code is used in CI/Docker pipeline, please send **order information** by registration email of this code to pyarmor@163.com to unlock it. Do not send this code only, it doesn't make sense.

Runtime Errors

Error messages reported by pyarmor

If it has an error code, it could be customized.

Table 6: Table-2. Runtime Errors of Obfuscated Scripts

Error Code	Error Message	Reasons
	error code out of range	Internal error
error_1	this license key is expired	
error_2	this license key is not for this machine	
error_3	missing license key to run the script	
error_4	unauthorized use of script	
error_5	this Python version is not supported	
error_6	the script doesn't work in this system	
error_7	the format of obfuscated script is incorrect	1. the obfuscated script is made by other Pyarmor version 2. can not get runtime package path
error_8	the format of obfuscated function is incorrect	
	RuntimeError: Resource temporarily unavailable	When using option `-e` to obfuscate the script, the obfuscated script need connect to NTP server to check expire date. If network is not available, or something is wrong with network, it raises this error. Solutions: 1. use local time if device is not connected to internet. 2. try it again it may works.

Error messages reported by Python interpreter

Generally they are not pyarmor issues. Please consult Python documentation or google error message to fix them.

Table 7: Table-2.1 Other Errors of Obfuscated Scripts

Error Message	Reasons
ImportError: attempted relative import with no known parent package	1. `from .pyarmor_runtime_000000 import __pyarmor__` Do not use `-i` or `--prefix` if you don't know what they're doing. For all the other relative import issue, please check Python documentation to learn about relative import knowledge, then check Pyarmor *Man Page* to understand how to generate runtime packages in different locations.

Outer Errors

Here list some outer errors. Most of them are caused by missing some system libraries, or unexpected configuration. It need nothing to do by Pyarmor, just install necessary libraries or change system configurations to fix the problem.

By searching error message in google or any other search engine to find the solution.

- **Operation did not complete successfully because the file contains a virus or is potentially unwanted software question**

  It's caused by Windows Defender, not Pyarmor. I'm sure Pyarmor is safe, but it uses some techniques which let anti-virus tools make wrong decision. The solutions what I thought of

  1. Check documentation of Windows Defender
  2. Ask question in MSDN
  3. Google this error message

- **Library not loaded: '@rpath/Frameworks/Python.framework/Versions/3.9/Python'**

  When Python is not installed in the standard path, or this Python is not Framework, pyarmor reports this error. The solution is using `install_name_tool` to change `pytransform3.so`. For example, in *anaconda3* with Python 3.9, first search which CPython library is installed:

```
$ otool -L /Users/my_username/anaconda3/bin/python
```

  Find any line includes `Python.framework`, `libpython3.9.dylib`, or `libpython3.9.so`, the filename in this line is CPython library. Or find it in the path:

```
$ find /Users/my_username/anaconda3 -name "Python.framework/Versions/3.9/Python"
$ find /Users/my_username/anaconda3 -name "libpython3.9.dylib"
$ find /Users/my_username/anaconda3 -name "libpython3.9.so"
```

  Once find CPython library, using `install_name_tool` to change and codesign it again:

```
$ install_name_tool -change @rpath/Frameworks/Python.framework/Versions/3.9/
↪Python /Users/my_username/anaconda3/lib/libpython3.9.dylib /Users/my_username/
↪anaconda3/lib/python3.9/site-packages/pyarmor/cli/core/pytransform3.so
$ codesign -f -s - /Users/my_username/anaconda3/lib/python3.9/site-packages/
↪pyarmor/cli/core/pytransform3.so
```

3.4 Topics

3.4.1 Insight Into Obfuscation

Filter scripts by finder

Script ext is not .py, list it in command line. For example, `my.config` is a python script but not standard extension name:

```
pyarmor gen main.py my.config
```

To include special script in package. For example:

```
pyarmor cfg finder:includes="lib/my.config"
pyamor gen -r lib
```

To exclude "test" and all the path "test":

```
pyarmor cfg finder:excludes + "*/test"
```

To include data files, these data file will be copied to output:

```
pyarmor cfg finder:data_files="lib/readme.txt"
pyamor gen -r lib
```

For example, the test-project hierarchy is as follows:

```
$ tree test-project

test-project
├── MANIFEST.in
├── pyproject.toml
├── setup.cfg
└── src
    └── parent
        ├── child
        │   └── __init__.py
        └── __init__.py
```

There are 2 exclude rules `*__pycache__` and `*/test.py` to filter scripts:

```
$ cd test-project
$ pyarmor cfg finder:exclude + "*__pycache__ */test.py"
$ pyamor gen -r src/parent
```

It uses `fnmatch` to match pattern, the matched item is excluded. Here are check list:

```
fnmatch("src/parent/__init__.py", "*__pycache__")
fnmatch("src/parent/__init__.py", "*/test.py")

fnmatch("src/parent/child", "*__pycache__")
fnmatch("src/parent/child", "*/test.py")

fnmatch("src/parent/child/__init__.py", "*__pycache__")
fnmatch("src/parent/child/__init__.py", "*/test.py")
```

3.4.2 Understanding Obfuscated Script

Remain as standard '.py' files

The obfuscated scripts are normal Python scripts, it's clear by checking the content of `dist/foo.py`:

```
1 from pyarmor_runtime_000000 import __pyarmor__
2 __pyarmor__(__name__, __file__, b'\xa...')
```

It's a simple script, first imports function `__pyarmor__` from package `pyarmor_runtime_000000`, then call this function.

Runtime package

This package `pyarmor_runtime_000000` is generated by Pyarmor, it's also a normal Python package, here it's package content:

```
$ ls dist/pyarmor_runtime_000000
...    __init__.py
...    pyarmor_runtime.so
```

There is binary extension module `pyarmor_runtime`, this is a big difference from plain Python script. Generally using binary extensions means the obfuscated scripts

- may not be compatible with different builds of CPython interpreter.
- often will not work correctly with alternative interpreters such as PyPy, IronPython or Jython

For example, when obfuscating scripts by Python 3.8, they can be run by any Python 3.8.x, but can't be run by Python 3.7, 3.9 etc.

For example, packaging pure *.py* script is easy, but packaging binary extension need more work.

For example, in Android pure *.py* script can be run in any location, but binary extensions must be in special system paths.

The runtime package `pyarmor_runtime_000000` could be in any path, it can be taken as a third-party package, save it in any location, and import it following Python import system.

pyarmor provides several options *-i*, *--prefix* to help generating right code to import it.

Runtime key

The runtime key generally is embedded into extension module `pyarmor_runtime`, it also could be an outer file. It stores expire date, bind devices, and user private data etc.

Extension module `pyarmor_runtime` will not load the obfuscated script unless the runtime key exists and is valid.

User also could store any private data in the runtime key, then use *hook script* to check private data in the obfuscated scripts.

If runtime key is stored in an outer file, any readable text in the header will be ignored. User can add comment at the header of runtime key file, the rest part are bytes data, only in the obfuscated scripts they could be read.

Restrict modes

By default the obfuscated scripts can't be changed.

After using *--private*, the obfuscated scripts could not be imported by plain script or Python interpreter.

After using *--restrict*, other plain scripts could not call any method in the obfuscated modules.

Disable all the restrictions by this command:

```
$ pyarmor cfg restrict_module 0
```

Generally only disable all the restrictions for specified module. For example, only no restrictions for module `NAME`:

```
$ pyarmor cfg -p NAME restrict_module 0
```

The differences of obfuscated scripts

Although **use obfuscated scripts as they're normal Python scripts**, but the obfuscated scripts are still different from pure Python scripts, they changes a few Python features and results in some third party packages could not work.

Here are major changed features:

- The obfuscated scripts are bind to Python major/minor version. For example, if it's obfuscated by Python 3.6, it must run by Python 3.6. It doesn't work for Python 3.5
- The obfuscated scripts are platform-dependent, supported platforms and Python versions refer to *Building Environments*
- If Python interpreter is compiled with Py_TRACE_REFS or Py_DEBUG, it will crash to run obfuscated scripts.

- Any module may not work if it try to visit the byte code, or some attributes of code objects in the obfuscated scripts. For example most of `inspect` function are broken.
- Pass the obfuscated code object by `cPickle` or any third serialize tool may not work.
- `sys._getframe([n])` may get the different frame. Note that many third packages uses this feature to get local variable and broken. For example, `pandas`, `cherrypy`.
- The code object attribute `__file__` is `<frozen name>` other than real filename.

  Note that module attribute `__file__` is still filename. For example, obfuscate the script `foo.py` and run it:

```
def hello(msg):
    print(msg)

# The output will be 'foo.py'
print(__file__)

# The output will be '<frozen foo>'
print(hello.__file__)
```

A few options may also change something:

- `pyarmor cfg mix_argname=1` hides annotations.

See also:

Work with Third-Party Libraries

Supported Third-Party Interpreter

About third-party interpreter, for example Jython, and any embedded Python C/C++ code, only they could work with CPython *extension module*, they could work with Pyarmor. Check third-party interpreter documentation to make sure this.

A few known issues

- On Linux, *RTLD_GLOBAL* must be set as loading *libpythonXY.so* by *dlopen*, otherwise obfuscated scripts couldn't work.
- Boost::python does not load *libpythonXY.so* with *RTLD_GLOBAL* by default, so it will raise error "No PyCode_Type found" as running obfuscated scripts. To solve this problem, try to call the method *sys.setdlopenflags(os.RTLD_GLOBAL)* as initializing.
- *PyPy* could not work with pyarmor, it's total different from *CPython*
- WASM is not supported.

See also:

Target Environments

3.4.3 Insight Into Pack Command

Pyarmor 8.0 has no command pack, but *--pack*. It could specify an executable file generated by PyInstaller:

```
pyinstaller foo.py
pyarmor gen --pack dist/foo/foo foo.py
```

If no this option, pyarmor only obfuscates the scripts.

If this option is set, pyarmor first obfuscates the scripts, then does extra work:

- Unpacking this executable to a temporary folder
- Replacing the scripts in bundle with obfuscated ones
- Appending runtime files to the bundle in this temporary folder
- Repacking this temporary folder to an executable file and overwrite the old

Packing obfuscated scripts manually

If something is wrong with *--pack*, or the final bundle doesn't work, try to pack the obfuscated scripts manually.

You need know how to using PyInstaller and using spec file, if not, learn it by yourself.

Here is an example to pack script `foo.py` in the path `/path/to/src`

- First obfuscating the script by Pyarmor[1]:

```
cd /path/to/src
pyarmor gen -O obfdist -a foo.py
```

- Moving runtime package to current path[2]:

```
mv obfdist/pyarmor_runtime_000000 ./
```

- Already have `foo.spec`, appending runtime package to `hiddenimports`

```
a = Analysis(
    ...
    hiddenimports=['pyarmor_runtime_000000'],
    ...
)
```

- Otherwise generating `foo.spec` by PyInstaller[3]:

```
pyi-makespec --hidden-import pyarmor_runtime_000000 foo.py
```

- Patching `foo.spec` by inserting extra code after `a = Analysis`

```
a = Analysis(
    ...
)

# Patched by Pyarmor
_src = r'/path/to/src'
_obf = r'/path/to/src/obfdist'

_count = 0
for i in range(len(a.scripts)):
    if a.scripts[i][1].startswith(_src):
        x = a.scripts[i][1].replace(_src, _obf)
        if os.path.exists(x):
```

(continues on next page)

[1] Do not use *-i* and *--prefix* to obfuscate the scripts
[2] Just let PyInstaller could find runtime package without extra pypath
[3] Most of other PyInstaller options could be used here

(continued from previous page)

```
            a.scripts[i] = a.scripts[i][0], x, a.scripts[i][2]
            _count += 1
if _count == 0:
    raise RuntimeError('No obfuscated script found')

for i in range(len(a.pure)):
    if a.pure[i][1].startswith(_src):
        x = a.pure[i][1].replace(_src, _obf)
        if os.path.exists(x):
            if hasattr(a.pure, '_code_cache'):
                with open(x) as f:
                    a.pure._code_cache[a.pure[i][0]] = compile(f.read(), a.pure[i][1],
↪ 'exec')
            a.pure[i] = a.pure[i][0], x, a.pure[i][2]
# Patch end.

pyz = PYZ(a.pure, a.zipped_data, cipher=block_cipher)
```

- Generating final bundle by this patched `foo.spec`:

```
pyinstaller foo.spec
```

If following this example, please

- Replacing all the `/path/to/src` with actual path
- Replacing all the `pyarmor_runtime_000000` with actual name

how to verify obfuscated scripts have been packed

Inserting some print statements in the `foo.spec` to print which script is replaced, or add some code only works in the obfuscated script.

For example, add one line in main script `foo.py`

```
print('this is __pyarmor__', __pyarmor__)
```

If it's not obfuscated, the final bundle will raise error.

notes

3.4.4 Insight Into RFT Mode

For a simple script, pyarmor may reform the scripts automatically. In most of cases, it need extra work to make it works.

This chapter describes how RFT mode work, it's helpful to solve RFT mode issues of complex package and scripts.

What's RFT mode changed?

- function
- class
- method
- global variable
- local variable

- builtin name
- import name

What's RFT mode not changed?

- argument in function definition
- keyword argument name in call
- all the strings defined in the module attribute `__all__`
- all the name starts with `__`

It's simple to decide whether or not transform a single name, but it's difficult for each name in attribute chains. For example,

```
foo().stack[2].count = 3
(a+b).tostr().get()
```

So how to handle attribute `stack`, `count`, `tostr` and `get`? The problem is that it's impossible to confirm function return type or expression result type.In some cases, it may be valid to return different types with different arguments.

There are 2 methods for RFT mode to handle name in the attribute chains which don't know parent type.

- **rft-auto-exclude**

  This is default method.

  The idea is search all attribute chains in the scripts and analysis each name in the chain. If not sure it's safe to rename, add it to exclude table, and do not touch all the names in exclude table.

  By default the file `.pyarmor/rft_exclude_table` is used to store exclude table.

  When pyarmor rft mode first run, exclude table is empty. It scans each script and append unknown names to exclude table. After all the scripts are obfuscated, it stores all the names in the exclude table to the file `.pyarmor/rft_exclude_table`.

  RFT mode doesn't remove this file, only append new names to it repeatedly, please delete it manually when needed.

  When second run rft mode, it loads exclude table from `.pyarmor/rft_exclude_table`. Comparing with the first time exclude table is empty, obviously the second time more names are kept, it may fix some name errors.

  It's simple to use, but may leave more names not changed.

- **rft-auto-include**

  This method first search all the functions, classes and methods in the scripts, add them to include table, and transform all of them. If same name is used in attribute chains, but can't make sure its type, leave attribute name as it is.

  For a simple script, Pyarmor could transform the script automatically. But for a complex script, it may raise name binding error. For example:

  ```
  $ python dist/foo.py

  AttributeError: module 'foo' has no attribute 'register_namespace'
  ```

  In order to fix this problem, exclude the problem name, leave it as it is by this way:

```
$ pyarmor cfg rft_excludes + "register_namespace"
$ pyarmor gen --enable-rft foo.py
$ python dist/foo.py
```

Repeat these steps to exclude all problem names, until it works.

This method could transform more names, but need more efforts to make the scripts work.

Enable RFT Mode

Enable RFT mode in command line:

```
$ pyarmor gen --enable-rft foo.py
```

Enable it by **`pyarmor cfg`**:

```
$ pyarmor cfg enable_rft=1
$ pyarmor gen foo.py
```

Enable **rft-auto-include** method by disable `rft_auto_exclude`:

```
$ pyarmor cfg rft_auto_exclude=0
```

Enable **rft-auto-exclude** method again:

```
$ pyarmor cfg rft_auto_exclude=1
```

Check transformed script

When trace rft mode is enabled, RFT mode will generate transformed script in the path `.pyarmor/rft` with full package name:

```
$ pyarmor cfg trace_rft 1
$ pyarmor gen --enable-rft foo.py
$ ls .pyarmor/rft
```

Check the transformed script:

```
$ cat .pyarmor/rft/foo.py
```

Note: This feature only works for Python 3.9+

Trace rft log

When both of trace log and trace rft are enabled, RFT mode will log which names and attributes are transformed:

```
$ pyarmor cfg enable_trace=1 trace_rft=1
$ pyarmor gen --enable-rft foo.py
$ grep trace.rft .pyarmor/pyarmor.trace.log

trace.rft            foo:1 (import sys as pyarmor__1)
trace.rft            foo:12 (self.wScan->self.pyarmor__4)
```

The first log means module `sys` is transformed to `pyarmor__1`

The second log means `wScan` is transformed to `pyarmor__4`

Exclude name rule

When RFT scripts complain of name not found error, just exclude this name. For example, if no found name `mouse_keybd`, exclude this name by this command:

```
$ pyarmor cfg rft_excludes "mouse_keybd"
$ pyarmor gen --enable-rft foo.py
```

If no found name like `pyarmor__22`, find the original name in the trace log:

```
$ grep pyarmor__22 .pyarmor/pyarmor.trace.log

trace.rft            foo:65 (self.height->self.pyarmor__22)
trace.rft            foo:81 (self.height->self.pyarmor__22)
```

From search result, we know `height` is the source of `pyarmor__22`, let's append it to exclude table:

```
$ pyarmor cfg rft_excludes + "height"
$ pyarmor gen --enable-rft foo.py
$ python dist/foo.py
```

Repeat these step until all the problem names are excluded.

Handle wild card form of import

The wild card form of import — *from module import* * — is a special case.

If this module is in the obfuscated package, RFT mode will parse the source and check the module's namespace for a variable named `__all__`

If this module is outer package, RFT mode could not get the source. So RFT mode will import it and query module attribute `__all__`. If this module could not be imported, it may raise `ModuleNotFoundError`, please set `PYTHONPATH` or any other way let Python could import this module.

If `__all__` is not defined, the set of public names includes all names found in the module's namespace which do not begin with an underscore character ('_').

Handle module attribute `__all__`

By default RFT mode doesn't touch all the names in the module `__all__`. If this name is defined as a Class, its methods and attributes are not changed.

It's possible to ignore this attribute by this command:

```
$ pyarmor cfg rft_export__all__ 0
```

It will transform names in the `__all__`, but it may not work if it's imported by other scripts.

3.4.5 Insight Into BCC Mode

BCC mode could convert most of functions and methods in the scripts to equivalent C functions, those c functions will be compiled to machine instructions directly, then called by obfuscated scripts.

It requires `c` compiler. In Linux and Darwin, `gcc` and `clang` is OK. In Windows, only `clang.exe` works. It could be configured by one of these ways:

- If there is any `clang.exe`, it's OK if it could be run in other path.
- Download and install Windows version of LLVM
- Download *https://pyarmor.dashingsoft.com/downloads/tools/clang-9.0.zip*, it's about 26M bytes, there is only one file in it. Unzip it and save `clang.exe` to `$HOME/.pyarmor/`. `$HOME` is home path of current logon user, check the environment variable `HOME` to get the real path.

Enable BCC mode

After compiler works, using *`--enable-bcc`* to enable BCC mode:

```
$ pyarmor gen --enable-bcc foo.py
```

All the source in module level is not converted to C function.

Trace bcc log

To check which functions are converted to C function, enable trace mode before obfuscate the script:

```
$ pyarmor cfg enable_trace=1
$ pyarmor gen --enable-bcc foo.py
```

Then check the trace log:

```
$ ls .pyarmor/pyarmor.trace.log
$ grep trace.bcc .pyarmor/pyarmor.trace.log

trace.bcc            foo:5:hello
trace.bcc            foo:9:sum2
trace.bcc            foo:12:main
```

The first log means `foo.py` line 5 function `hello` is protected by bcc. The second log means `foo.py` line 9 function `sum2` is protected by bcc.

Ignore module or function

When BCC scripts reports errors, a quick workaround is to ignore these problem modules or functions. Because BCC mode converts some functions to C code, these functions are not compatible with Python function object. They may not be called by outer Python scripts, and can't be fixed in Pyarmor side. In this case use configuration option `bcc:excludes` and `bcc:disabled` to ignore function or module, and make all the others work.

To ignore one module `pkgname.modname` by this command:

```
$ pyarmor cfg -p pkgname.modname bcc:disabled=1
```

To ignore one function in one module by this command:

```
$ pyarmor cfg -p pkgname.modname bcc:excludes + "function name"
```

Use `-p` to specify module name and option `bcc:excludes` for function name. No `-p`, same name function in the other scripts will be ignored too.

Exclude more functions by this way:

```
$ pyarmor cfg -p foo bcc:excludes + "hello foo2"
```

Let's enable trace mode to check these functions are ignored:

```
$ pyarmor cfg enable_trace 1
$ pyarmor gen --enable-bcc foo.py
$ grep trace.bcc .pyarmor/pyarmor.trace.log
```

Another example, in the following commands BCC mode ignores `joker/card.py`, but handle all the other scripts in package `joker`:

```
$ pyarmor cfg -p joker.card bcc:disabled=1
$ pyarmor gen --enable-bcc /path/to/pkg/joker
```

By both of `bcc:excludes` and `bcc:disabled`, make all the problem code fallback to default obfuscation mode, and let others could be converted to c function and work fine.

Changed features

Here are some changed features in the BCC mode:

- Calling *raise* without argument not in the exception handler will raise different exception.

```
>>> raise
RuntimeError: No active exception to re-raise

# In BCC mode
>>> raise
UnboundlocalError: local variable referenced before assignment
```

- Some exception messages may different from the plain script.
- Most of function attributes which starts with __ doesn't exists, or the value is different from the original.

Unsupported features

If a function uses any unsupported features, it could not be converted into C code.

Here list unsupported features for BCC mode:

```
unsupport_nodes = (
    ast.ExtSlice,

    ast.AsyncFunctionDef, ast.AsyncFor, ast.AsyncWith,
    ast.Await, ast.Yield, ast.YieldFrom, ast.GeneratorExp,

    ast.NamedExpr,

    ast.MatchValue, ast.MatchSingleton, ast.MatchSequence,
```

(continues on next page)

(continued from previous page)

```
    ast.MatchMapping, ast.MatchClass, ast.MatchStar,
    ast.MatchAs, ast.MatchOr
)
```

And unsupported functions:

- exec
- eval
- super
- locals
- sys._getframe
- sys.exc_info

For example, the following functions are not obfuscated by BCC mode, because they use unsupported features or unsupported functions:

```
async def nested():
    return 42

def foo1():
    for n range(10):
        yield n

def foo2():
   frame = sys._getframe(2)
   print('parent frame is', frame)
```

3.4.6 Security and Performance

About Security

Pyarmor focus on protecting Python scripts, by several irreversible obfuscation methods, now Pyarmor make sure the obfuscated scripts can't be restored by any way.

Pyarmor provides rich options to obfuscate scripts to balance security and performance. If anyone announces he could broken pyarmor, please try a simple script with different security options, refer to *Highest security and performance*. If any irreversible obfuscation could be broken, report this security issue to pyarmor@163.com. Do not paste any hack link in pyarmor project.

However Pyarmor isn't good at memory protection and anti-debug. Generally even debugger tracing binary extension `pyarmor_runtime` could not help to restore obfuscated scripts, but it may bypass runtime key verification.

If you care about runtime memory data protection and anti-debug, check *Protecting Runtime Memory Data*

About Performance

Though the highest security could protect Python scripts from any hack method, but it may reduce performance. In most of cases, we need pick the right options to balance security and performance.

Here we test some options to understand their impact on performance. All the following tests use 2 scripts `benchmark.py` and `testben.py`. Note that the test results are different even run same test script in same machine twice, not speak of different test script in different machine. So the test data in these tables are only guideline, not exact.

The content of `benchmark.py`

```
import sys


class BenTest(object):

    def __init__(self):
        self.a = 1
        self.b = "b"
        self.c = []
        self.d = {}


def foo():
    ret = []
    for i in range(100000):
        ret.extend(sys.version_info[:2])
        ret.append(BenTest())
    return len(ret)
```

The content of `testben.py`

```
import benchmark
import sys
import time


def metric(func):
    if not hasattr(time, 'process_time'):
        time.process_time = time.clock

    def wrap(*args, **kwargs):
        t1 = time.process_time()
        result = func(*args, **kwargs)
        t2 = time.process_time()
        print('%-16s: %10.3f ms' % (func.__name__, ((t2 - t1) * 1000)))
        return result
    return wrap


def test_import():
    t1 = time.process_time()
    import benchmark2 as m2
    t2 = time.process_time()
    print('%-16s: %10.3f ms' % ('test_import', ((t2 - t1) * 1000)))
    return m2


@metric
def test_foo():
    benchmark.foo()


if __name__ == '__main__':
    print('Python %s.%s' % sys.version_info[:2])
    test_import()
    test_foo()
```

Different Python Version Performance

First obfuscate the scripts with default options, run it in different Python version, compare the elapsed time with original scripts.

In order to test the difference without and with `__pycache__`, run scripts twice.

There are 3 check points:

1. `Import fresh module` without `__pycache__`
2. `Import module 2nd` with `__pycache__`
3. `Run function "foo"`, an obfuscated class is called 10,000 times

Here are test steps:

```
$ rm -rf dist __pycache__

$ cp benchmark.py benchmark2.py
$ python testben.py

Python 3.7
test_import     :   1.303 ms
test_foo        : 250.360 ms

$ python testben.py

Python 3.7
test_import     :   0.290 ms
test_foo        : 252.273 ms

$ pyarmor gen testben.py benchmark.py benchmark2.py
$ python dist/testben.py

Python 3.7
test_import     :   0.907 ms
test_foo        : 311.076 ms

$ python dist/testben.py

Python 3.7
test_import     :   0.454 ms
test_foo        : 359.138 ms
```

Table 8: Table-1. Pyarmor Performance with Python Version

Time (ms)	Import fresh module		Import module 2nd		Run function “foo”	
Python	Origin	Pyarmor	Origin	Pyarmor	Origin	Pyarmor
3.7	1.303	0.907	0.290	0.454	252.2	311.0
3.8	1.305	0.790	0.286	0.338	272.232	295.973
3.9	1.198	1.681	0.265	0.449	267.561	331.668
3.10	1.070	1.026	0.408	0.300	281.603	322.608
3.11	1.510	0.832	0.464	0.616	164.104	289.866

RFT Mode Performance

RFT mode should be same fast as original scripts.

Here we compare RFT mode with default options, the test data is got by this way.

First obfuscate scripts with default options, then run it.

Then obfuscate scripts with RFT mode, and run it again:

```
$ rm -rf dist
$ pyarmor gen testben.py benchmark.py benchmark2.py
$ python dist/testben.py

$ rm -rf dist
$ pyarmor gen --enable-rft testben.py benchmark.py benchmark2.py
$ python dist/testben.py
```

Table 9: Table-2. Performance of RFT Mode

Time (ms)	Import fresh module		Run function "foo"		Remark
Python	Pyarmor	RFT Mode	Pyarmor	RFT Mode	
3.7	1.083	1.317	334.313	324.023	
3.8	0.774	1.109	239.217	241.697	
3.9	0.775	0.809	304.838	301.789	
3.10	2.182	1.049	310.046	339.414	
3.11	0.882	0.984	258.309	264.070	

Next, we compare RFT mode and `--obf-code 0` with original scripts by this way:

```
$ rm -rf dist __pycache__
$ python testben.py
...

$ pyarmor gen --enable-rft --obf-code=0 testben.py benchmark.py benchmark2.py
$ python testben.py
...
```

Table 10: Table-2.1 Performance of RFT Mode and obf-code 0

Time (ms)	Import fresh module		Run function "foo"		Remark
Python	Pyarmor	RFT Mode	Pyarmor	RFT Mode	
3.7	0.757	1.844	307.325	272.672	
3.8	0.791	0.747	276.865	243.436	
3.9	1.276	0.986	246.407	236.138	
3.10	2.563	1.142	256.583	260.196	
3.11	0.952	0.938	185.435	154.390	

They're almost same.

BCC Mode Performance

BCC mode converts some code to C function, it need extra time to load binary code, but function may be faster. The following test data got by this way:

```
$ rm -rf dist __pycache__
$ python testben.py
...

$ python testben.py
...

$ pyarmor gen --enable-bcc testben.py benchmark.py benchmark2.py
$ python dist/testben.py
```

(continues on next page)

(continued from previous page)

```
...

$ python dist/testben.py
...
```

Table 11: Table-3. Performance of BCC Mode with Python Version

Time (ms)	Import fresh module		Import module 2nd		Run function “foo”	
Python	Origin	BCC Mode	Origin	BCC Mode	Origin	BCC Mode
3.7	1.086	1.177	0.342	0.391	344.640	271.426
3.8	1.099	1.397	0.351	0.400	291.244	251.520
3.9	1.229	1.076	0.538	0.362	306.594	254.458
3.10	1.267	0.999	0.255	0.796	302.398	247.154
3.11	1.146	1.056	0.273	0.536	206.311	189.582

Impact of Different Options

In order to facilitate comparison, each option is used separately. For example, test *--no-wrap* by this way:

```
$ rm -rf dist __pycache__
$ pyarmor testben.py
...

$ pyarmor gen --no-wrap testben.py benchmark.py benchmark2.py
$ pyarmor dist/testben.py

Python 3.7
test_import     :      0.971 ms
test_foo        :    306.261 ms
```

Table 12: Table-4. Impact of Different Options

Option	Performance	Security
--no-wrap	Increase	Reduce
--obf-module 0	Slightly increase	Slightly reduce
--obf-code 0	Remarkable increase	Remarkable reduce
--obf-code 2	Reduce	Increase
--enable-rft	Almost same	Remarkable increase
--enable-themida	Remarkable reduce	Remarkable increase
--mix-str	Reduce	Increase
--assert-call	Reduce	Increase
--assert-import	Slightly reduce	Increase
--private	Reduce	Increase
--restrict	Reduce	Increase

3.4.7 Localization and Internationalization

When building obfuscated scripts

For example:

```
pyarmor gen foo.py
```

Pyarmor first searches file `messages.cfg` in the *local path*, then searches in the *global path*

If `messages.cfg` exists, then read this file and save customized message to *runtime key*

If this file is not encoded by `utf-8`, set the right encoding `XXX` by this command:

```
$ pyarmor cfg messages=messages.cfg:XXX
```

See also:

Table-2. Runtime Errors of Obfuscated Scripts

When launching obfuscated scripts

For example:

```
python dist/foo.py
```

When something is wrong, the obfuscated script need report error which has an error code:

First decide default language by checking the following items in turn

- *PYARMOR_LANG*
- First part of *LANG*. For example, `en_US` or `zh_CN`

Then search error message table in the *runtime key*, if there is an error message both of language code and error code are matched, then return it.

Otherwise return default error message.

3.5 License Types

Contents

- *Introduction*
- *License types*
  - *License features*
- *Purchasing license*
  - *Refund policy*
- *Upgrading old license*
  - *Upgrading old license to Pyarmor Basic*

3.5.1 Introduction

This documentation is only apply to Pyarmor 8.0 plus.

Pyarmor is published as shareware, free trial version never expires, but there are some limitations:

(1) Can not obfuscate big scripts[1]

(2) Can not use feature mix-str[2] to obfuscate string constant in scripts

[1] Big Script means file size exceeds a certain value.
[2] Mix Str: obfuscating string constant in script

(3) Can not use RFT Mode[3], BCC Mode[4]

(4) Can not be used for any commercial product without permission

(5) Can not change runtime package name "pyarmor_runtime_000000"

(6) Can not be used to provide obfuscation service in any form

(7) Can not use obf-code > 1

These limitations can be unlocked by different License Types except last one.

3.5.2 License types

Pyarmor has 3 kind of licenses:

Pyarmor Basic Basic license could unlock limitations (1) (2) (5) (7).

Each obfuscation need online verify license.

Pyarmor Pro Pro license could unlock limitations (1) (2) (3) (4) (5) (7).

Each obfuscation need online verify license.

Pyarmor Group Group license could unlock limitations (1) (2) (3) (4) (5) (7).

Offline obfuscation.

Refer to *use Pyarmor License*

For the obfuscated scripts run in the customer's device, Pyarmor has no any limitations, it's totally controlled by users. Pyarmor only cares about build machine.

Each license has an unique number, the format is `pyarmor-vax-xxxxxx`, which x stands for a digital.

Each product requires one License No. So any product in global also has an unique number in Pyarmor world.

If user has many products, and has purchased one license for the first product. The second product could use first product license only if sale income of the second product less than 10x license fees. Once greater than 10x license fees, the second product need purchase its own license. It's same to user's other products.

One product in Pyarmor world means a product name and everything that makes up this name.

It includes all the devices to develop, build, debug, test product.

It also includes product current version, history versions and all the future versions.

One product may has several variants, each variant name is composed of product name plus feature name. As long as the proportion of the variable part is far less than that of the common part, they're considered as "one product".

Pyarmor License could be installed in many machines and devices which belong to licensed product. But there is limitation to be used at the same time.

In 24 hours only less than 100 devices can use one same Pyarmor License. Pyarmor License be used means use any feature of Pyarmor in one machine. Running obfuscated scripts generated by Pyarmor is not considered as Pyarmor License be used.

In details read EULA of Pyarmor

What's one product

First of all, if not for sale, all the Python scripts are belong to one product "non-profits".

Pyarmor is one product, it includes:

[3] RFT Mode: renaming function/class/method/variable in Python scripts

[4] BCC Mode: Transforming some Python functions in scripts to c functions, compile them to machine instructions directly

- Pyarmor basic, Pyarmor pro, and Pyarmor group
- pyarmor-webui which provides graphics interface for pyarmor.
- the order system of Pyarmor is a Django's app running in cloud-server. This Django's app also belongs to one product Pyarmor.
- the laptop used to develop Pyarmor, the PCs used to test Pyarmor, the cloud-server to serve order system of Pyarmor, all of them belong to one product Pyarmor.
- Pyarmor 7.x, Pyarmor 8.x and Pyarmor 9.x

Microsoft Office is not one product, because each product in Microsoft Office is functional independence. For example, Microsoft Word and Microsoft Excel belong to Microsoft Office, but they're totally different.

Microsoft Word is one product, and Microsoft Word 2003Word 2007 etc. are belong to one product Microsoft word.

License features

Table 13: Table-1. License Features

Features	Trial	Basic	Pro	Group	Remark
Basic Obfuscation	Y	Y	Y	Y	[5]
Expired Script	Y	Y	Y	Y	[6]
Bind Device	Y	Y	Y	Y	[7]
JIT Protection	Y	Y	Y	Y	[8]
Assert Protection	Y	Y	Y	Y	[9]
Themedia Protection	Y	Y	Y	Y	[10]
Big Script	No	Y	Y	Y	
Mix Str	No	Y	Y	Y	
obf-code > 1	No	Y	Y	Y	[11]
RFT MODE	No	No	Y	Y	
BCC MODE	No	No	Y	Y	

notes

3.5.3 Purchasing license

Open shopping cart in any web browser:

https://order.mycommerce.com/product?vendorid=200089125&productid=301044051

If you have Pyarmor 8.0+ installed, this command also could open shopping cart:

```
$ pyarmor reg --buy
```

In the shopping cart, select License Type and complete the payment online.

Please fill reg-name with personal or company name when placing order.

[5] Basic Obfuscation: obfuscating the scripts by default options
[6] Expired Script: obfuscated scripts has expired date
[7] Bind Device: obfuscated scripts only run in specified devices
[8] JIT Protection: processing some sensitive data by runtime generated binary code
[9] Assert Protection: preventing others from hacking obfuscated scripts
[10] Themedia Protection: using Themedia to protect Windows dlls
[11] `--obf-code=2` is new in Pyarmor 8.2

Table 14: Table-2. License Prices

License Type	Net Price($)	Remark
Basic	52	
Pro	89	
Group	158	

An activation file named `pyarmor-regcode-xxxx.txt` will be sent by email immediately after payment is completed successfully.

Following the guide in activation file to take the purchased license effects. Or check *Using Pyarmor License*

There are no additional license fees, apart from the cost of the license. And it only needs to be paid once, not periodically

Refund policy

If activation file isn't used, and purchasing date is in 30 days, refund is acceptable. Please

1. Email to Ordersupport@mycommerce.com with order information and ask for refund.
2. Or click FindMyOrder page to submit refund request

Out of 30 days, or activation file has been used, refund request will be rejected.

3.5.4 Upgrading old license

Not all the old license could be upgraded to latest version.

The old license could be upgraded to Pyarmor Basic freely only if it matches these conditions:

- Following new EULA of Pyarmor
- The license no. starts with `pyarmor-vax-`
- The original activation file `pyarmor-regcode-xxxx.txt` is used not more than 100 times and exists

If failed to upgrade the old license, please purchase new license to use Pyarmor latest version.

The old license can't upgrade to Pyarmor Pro and Group.

Upgrading old license to Pyarmor Basic

First find the activation file `pyarmor-regcode-xxxx.txt`, which is sent to registration email when purchasing the license.

Next install Pyarmor 8.2+, according to new EULA of Pyarmor, each license is only for one product.

Assume this license will be used to obfuscate product `XXX`, run this command:

```
$ pyarmor reg -u -p "XXX" pyarmor-regcode-xxxx.txt
```

Check the upgraded license information:

```
$ pyarmor -v
```

After upgrade successfully, do not use activation file `pyarmor-regcode-xxxx.txt` again, it's invalid now. A new *registration file* like `pyarmor-regfile-xxxx.zip` will be generated at the same time.

In other devices using this new *registration file* to register Pyarmor by this command:

```
$ pyarmor reg pyarmor-regfile-xxxx.zip
```

After successful registration, all obfuscations will automatically apply this license, and each obfuscation requires online license verification.

If old license is used by many products (mainly old personal license), only one product could be used after upgrading. For the others, it need purchase new license.

3.6 FAQ

3.6.1 Asking questions in GitHub

Before ask question, please try these solutions:

- If using **`pyarmor-7`** or Pyarmor < 8.0, please check Pyarmor 7.x Doc
- Check *the detailed table of contents*
- If you have not read *Getting Started*, read it
- Check *Error Messages*
- If you have trouble in pack, check *Insight Into Pack Command*
- If you have trouble in *RFT Mode*, check *Using rftmode pro*
- If you have trouble in *BCC Mode*, check *Using bccmode pro*
- If you have trouble with third-party libraries, check *Work with Third-Party Libraries*
- If it's related to security and performance, check *Security and Performance*
- Look through this page
- Enable debug mode and trace log, check console log and trace log to find more information
- Make sure the scripts work without obfuscation
- Do a simple test, obfuscate a hello world script, and run it with python
- If not using latest Pyarmor version, try to upgrade Pyarmor to latest version.
- Search in the Pyarmor issues
- Search in the Pyarmor discussions

Please report bug in issues and ask questions in discussions

When report bug in issues, please copy the whole command line **`pyarmor gen`** and first 4 lines in the console, do not mask version and platform information, and do not paste snapshot image:

```
$ pyarmor gen -O dist --assert-call foo.py
INFO     Python 3.10.0
INFO     Pyarmor 8.1.1 (trial), 000000, non-profits
INFO     Platform darwin.x86_64
```

3.6.2 Packing

In the old pyarmor 7, I'm using "pyarmor pack ...", I could not find any relate information for this in the pyarmor 8.2. How to solve this?

There is no identical pack in Pyarmor 8, Pyarmor 8+ only provide repack function to handle bundle of PyInstaller. Refer to basic tutorial, topic insight into pack and this solved issue Pyarmor pack missing in pyarmor 8.0

3.6.3 License

I am interested to know if the users are entitled to updates to ensure compatibility with future versions of Python.

No. Pyarmor license works with current Pyarmor version forever, but may not work with future Pyarmor version. I can't make sure current Pyarmor version could support all the future versions of Python, so the answer is no.

we use Docker to build/obfuscate the code locally then publish the Docker file to the client. After the build stage, the whole environment (and the license) is gone. I wonder how the workflow would be? Can I add the license file to the pipeline and register every time and build?

It's no problem to run Pyarmor in Docker or CI pipeline to obfuscate application. Each build registering pyarmor with `pyarmor-regfile-xxxx.zip` which is generated in initial registration. But It's not allowed to distribute package pyarmor and *Pyarmor Basic*, *Pyarmor Pro*, *Pyarmor Group* License to customer, and don't run too many build dockers.

We are currently using a trial license for testing, but unfortunately our scripts are big and we are not able to statistically test the operation of Pyarmor. Do you have a commercial trial license for a certain trial period so that we can test the operation of Pyarmor for our scripts?

Sorry, Pyarmor is a small tool and only cost small money, there is no demo license plan.

Most of features could be verified in trial version, other advanced features, for example, mix-str, bcc mode and rft mode, could be configured to ignore one function or one script so that all the others could work with these advanced features.

Is the Internet connection only required to generate the obfuscated script? No internet connection is required on the target device that uses such script?

No internet connection is required on target device.

Pyarmor has no any control or limitation to obfuscated scripts, the behaviors of obfuscated scripts are totally defined by user.

Please check Pyarmor EULA 3.4.1

Our company has a suite of products that we offer together or separately to our clients. Do we need a different license for each of them?

For a suite of products, if each product is different totally, for example, a suite "Microsoft Office" includes "Microsoft Excel", "Microsoft Word", each product need one license.

If a suite of products share most of Python scripts, as long as the proportion of the variable part of each product is far less than that of the common part, they're considered as "one product".

If each product in a suite of products is functionally complementary, for example, product "Editor" for editing the file, product "Viewer" for view the file, they're considered as "one product"

Upgrading

If we buy version 8 license, is it compatible with earlier versions like 6.7.2?

No. Pyarmor 8 license can't be used with earlier versions, it may report HTTP 401 error or some unknown errors.

Can we obfuscate our code base with the same level as current? (we are obfuscating our code using super plus mode ("–advanced 5"). Is that available on Pyarmor Basic?

The old license is valid for ever. In this case need not upgrade old license to Pyarmor Basic license, just install Pyarmor 8.x, and using `pyarmor-7` with old license.

Check *License Types* for more information about upgrading

If we upgrade the old license, will the current license expire? (no more available in terms of Pyarmor v7?

If upgrade old license to any Pyarmor 8 license, the current license is no more available in the terms of Pyarmor 7.

How long is the current license valid? Is there a published end-of-support schedule?

The license is valid for ever with Pyarmor version when purchasing this license, but may not work for future Pyarmor, there is no schedule about in which version current license doesn't work.

Since the first release Pyarmor changed its license 3 times

- the initial license issued around year 2010 (I forget the exact date)
- the second license issued on 2019-10-10
- this is the third license, issued on 2023-03-10.

3.6.4 Purchasing

How to refund my order?

If this order isn't activated and in 30 days since purchasing, you can refund the order by one of ways

1. Email to Ordersupport@mycommerce.com with order information and ask for refund.
2. Or click FindMyOrder page to submit refund request

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

pyarmor, 49
pyarmor.cli, 49
pyarmor.cli.core, 49
pyarmor.cli.runtime, 49

Index

Symbols

-assert-call
    pyarmor-gen command line option, 56
-assert-import
    pyarmor-gen command line option, 57
-bind-data DATA
    pyarmor-gen command line option, 54
-enable <jit,rft,bcc,themida>
    pyarmor-gen command line option, 56
-enable-bcc
    pyarmor-gen command line option, 56
-enable-jit
    pyarmor-gen command line option, 56
-enable-rft
    pyarmor-gen command line option, 56
-enable-themida
    pyarmor-gen command line option, 56
-home PATH[,GLOBAL[,LOCAL[,REG]]]
    pyarmor command line option, 50
-mix-str
    pyarmor-gen command line option, 56
-no-wrap
    pyarmor-gen command line option, 56
-obf-code <0,1,2>
    pyarmor-gen command line option, 56
-obf-module <0,1>
    pyarmor-gen command line option, 56
-outer
    pyarmor-gen command line option, 55
-pack BUNDLE
    pyarmor-gen command line option, 57
-period N
    pyarmor-gen command line option, 54
-platform NAME
    pyarmor-gen command line option, 55
-prefix PREFIX
    pyarmor-gen command line option, 53
-private
    pyarmor-gen command line option, 55
-restrict
    pyarmor-gen command line option, 55
-O PATH, -output PATH
    pyarmor-gen command line option, 52
-b DEV, -bind-device DEV
    pyarmor-gen command line option, 53
-d, -debug
    pyarmor command line option, 50
-e DATE, -expired DATE
    pyarmor-gen command line option, 53
-g ID, -device ID
    pyarmor-reg command line option, 62
-g, -global
    pyarmor-cfg command line option, 60
-i
    pyarmor-gen command line option, 52
-p NAME
    pyarmor-cfg command line option, 60
-p NAME, -product NAME
    pyarmor-reg command line option, 62
-q, -silent
    pyarmor command line option, 50
-r, -recursive
    pyarmor-gen command line option, 52
-r, -reset
    pyarmor-cfg command line option, 60
-u, -upgrade
    pyarmor-reg command line option, 62
__assert_armored__() (*built-in function*), 68
__pyarmor__() (*built-in function*), 67

A

Activation File, **47**

B

BCC Mode, **47**
Build Machine, **47**

E

environment variable

LANG, 27, 66, 67, 87
PYARMOR_CC, 62
PYARMOR_CLI, 62
PYARMOR_HOME, 51, 62
PYARMOR_LANG, 27, 66, 87
PYARMOR_PLATFORM, 62
PYARMOR_RKEY, 38, 48, 58, 67
PYTHONPATH, 79
extension module, **47**

G

Global Path, **47**

H

Home Path, **48**
Hook script, **48**

J

JIT, **48**

L

LANG, 27, 67, 87
Local Path, **48**

O

Outer Key, **48**

P

Platform, **48**
Plugin script, **48**
PluginName (*built-in class*), 64
post_build() (*PluginName static method*), 64
post_key() (*PluginName static method*), 64
post_runtime() (*PluginName static method*), 65
Pyarmor, **48**
pyarmor (*module*), 49
Pyarmor Basic, **48**, **88**
pyarmor command line option
-home PATH[,GLOBAL[,LOCAL[,REG]]], 50
-d, -debug, 50
-q, -silent, 50
Pyarmor Group, **48**, **88**
Pyarmor Home, **49**
Pyarmor License, **49**
Pyarmor Package, **49**
Pyarmor Pro, **49**, **88**
Pyarmor Users, **49**
pyarmor-cfg command line option
-g, -global, 60
-p NAME, 60
-r, -reset, 60
pyarmor-gen command line option
-assert-call, 56
-assert-import, 57
-bind-data DATA, 54
-enable <jit,rft,bcc,themida>, 56
-enable-bcc, 56
-enable-jit, 56
-enable-rft, 56
-enable-themida, 56
-mix-str, 56
-no-wrap, 56
-obf-code <0,1,2>, 56
-obf-module <0,1>, 56
-outer, 55
-pack BUNDLE, 57
-period N, 54
-platform NAME, 55
-prefix PREFIX, 53
-private, 55
-restrict, 55
-O PATH, -output PATH, 52
-b DEV, -bind-device DEV, 53
-e DATE, -expired DATE, 53
-i, 52
-r, -recursive, 52
pyarmor-reg command line option
-g ID, -device ID, 62
-p NAME, -product NAME, 62
-u, -upgrade, 62
pyarmor.cli (*module*), 49
pyarmor.cli.core (*module*), 49
pyarmor.cli.runtime (*module*), 49
PYARMOR_HOME, 51
PYARMOR_LANG, 27, 87
PYARMOR_RKEY, 38, 48, 58
Python, **49**
Python Package, **49**
Python Script, **49**
PYTHONPATH, 79

R

Registration File, **49**
RFT Mode, **49**
Runtime Files, **49**
Runtime Key, **49**
Runtime Package, **49**

T

Target Device, **49**