Pyarmor Documentation

Release 9.0.8

Jondy Zhao

Jul 21, 2025

CONTENTS

Version
9.0.8

Homepage
https://pyarmor.dashingsoft.com/

Contact
pyarmor@163.com

Authors
Jondy

Copyright

Important: New features introduced by Pyarmor 9.1.0 are moved to Pyarmor Learning System

https://eke.dashingsoft.com/pyarmor/docs/en/index.html

It includes 3 commands: *pyarmor init*, *pyarmor env*, *pyarmor build*

And new obfuscated script types: *rft*, *mini*, *vmc*, *ecc*

CHAPTER

ONE

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

TWO

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

THREE

TABLE OF CONTENTS

3.1 Tutorials

3.1.1 Getting Started

Content

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
```

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
```

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

Since v8.5.0, it checks local time by default. If need to check internet time, configure *nts* to any NTP server. For example:

```
$ pyarmor cfg nts=pool.ntp.org
```

Actually this is the default configuration in previous versions. Sometimes NTP server may return *RuntimeError: Resource temporarily unavailable*, using HTTP service may solve this. For example:

```
$ pyarmor cfg nts=http://worldtimeapi.org/api
```

Binding obfuscated scripts to device

Since Pyarmor 8.4.6, got target machine hardware informations by *python -m pyarmor.cli.hdinfo*:

```
Default Harddisk Serial Number: 'HXS2000CN2A'
Default Mac address: '00:16:3e:35:19:3d'
Default IPv4 address: '128.16.4.10'
```

Before Pyarmor 8.4.6, using *pyarmor-7 hdinfo* to get hardware information.

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
        └── 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

Prerequisite

Pyarmor requires Python and C library (glibc or musl).

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 pyarmor
```

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

```
C:\> pip install 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.

If you need generate obfuscated scripts to run in other platforms, install the corresponding packages:

```
$ pip install pyarmor.cli.core.windows
$ pip install pyarmor.cli.core.themida
$ pip install pyarmor.cli.core.linux
$ pip install pyarmor.cli.core.darwin
$ pip install pyarmor.cli.core.freebsd
$ pip install pyarmor.cli.core.android
$ pip install pyarmor.cli.core.alpine
```

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

Note: If only using Pyarmor 8+ features, installing *pyarmor.cli* instead of *pyarmor*, could significantly decrease downloaded file size. For example:

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

Note: If need install old version Pyarmor, just specify the exact version. For example:

```
$ pip install pyarmor==8.5.12
```

For more information, please check *pip* doc

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
- **pyarmor-auth** used by Group License to support unlimited docker containers

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

Deprecated since version 8.2.9.

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.

Note: Do not use this method, it may not work since v8.2.9

Installation in offline device

All the Pyarmor pacakges are published in the PyPI, download them and copy to offlice device.

First install *pyarmor.cli.core*

Next install *pyarmor* or *pyarmor.cli*

For example, install offline Pyarmor 8.2.9 in Linux for Python 3.10:

```
$ pip install pyarmor.cli.core-3.2.9-cp310-none-manylinux1_x86_64.whl
$ pip install pyarmor-8.2.9.zip
```

In Android or FreeBSD, there is no wheel in *pyarmor.cli.core*, it should install source distribution and extra package *pyarmor.cli.core.android* or *pyarmor.cli.core.freebsd*. For example, install offline Pyarmor in Android for Python 3.10:

```
$ pip install pyarmor.cli.core-3.2.9.zip
$ pip install pyarmor.cli.core.android-3.2.9-cp310-none-any.whl
$ pip install pyarmor-8.2.9.zip
```

For some arches like *ppc64le*, *mips32el*, *mips64el*, *riscv64*, *loongarch64*, it need install *pyarmor.cli.core.linux* (glibc) or *pyarmor.cli.core.alpine* (musl). For example:

```
$ pip install pyarmor.cli.core-8.5.9.zip
$ pip install pyarmor.cli.core.linux-6.5.2-cp310-none-any.whl
$ pip install pyarmor.cli-8.5.9.zip
```

If need cross platform obfuscation, also install the corresponding platform package

- *pyarmor.cli.core.freebsd*
- *pyarmor.cli.core.android*
- *pyarmor.cli.core.windows*
- *pyarmor.cli.core.themida*
- *pyarmor.cli.core.linux*
- *pyarmor.cli.core.alpine*
- *pyarmor.cli.core.darwin*

For example, if need Themida protection, then install themida package:

```
$ pip install pyarmor.cli.themida-3.2.9-cp310-none-any.whl
```

In Linux to generate for Windows, install windows package:

```
$ pip install pyarmor.cli.windows-3.2.9-cp310-none-any.whl
```

If only using Pyarmor 8+ features, it's recommend to install *pyarmor.cli* instead of *pyarmor*, the former file size is significantly less than the latter. For example:

```
$ pip install pyarmor.cli-8.2.9.zip
```

Termux issues

In Termux, after installation it need patch extensions. For example:

```
$ patchelf --add-needed libpython3.11.so.0.1 /data/data/com.termux/files/usr/lib/python3.
↪11/site-packages/pyarmor/cli/core/android/aarch64/pytransform3.so
$ patchelf --add-needed libpython3.11.so.0.1 /data/data/com.termux/files/usr/lib/python3.
↪11/site-packages/pyarmor/cli/core/android/aarch64/pyarmor_runtime.so
```

Sometimes, it need set runpath too. For example:

```
$ patchelf --set-rpath /data/data/com.termux/files/usr/lib /path/to/{pytransform3,
↪pyarmor_runtime}.so
```

Otherwise it will raise error *dlopen failed: cannot locate symbol "PyFloat_Type"*

Run Pyarmor from Python script

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

For example,

```
from pyarmor.cli.__main__ import main_entry

args = ['gen', '-O', 'dist', '--platform', 'linux.x86_64,windows.x86_64', 'foo.py']
main_entry(args)
```

Run it by Python interpreter:

```
$ python tool.py
```

It's same as this command:

```
$ pyarmor gen -O dist --platform linux.x86_64,windows.x86_64 foo.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
$ pip uninstall pyarmor.cli.core.windows
$ pip uninstall pyarmor.cli.core.themida
$ pip uninstall pyarmor.cli.core.linux
$ pip uninstall pyarmor.cli.core.darwin
$ pip uninstall pyarmor.cli.core.freebsd
$ pip uninstall pyarmor.cli.core.android
$ pip uninstall pyarmor.cli.core.alpine

$ 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

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 generate `pyarmor.debug.log` to get more information:

```
$ pyarmor -d gen foo.py
$ cat pyarmor.debug.log
```

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.trace.log`. For example:

```
$ pyarmor gen foo.py
$ cat 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* prevents plain scripts visiting module attributes

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
```

(continues on next page)

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

(continued from previous page)

```
    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.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
```

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/
   ```

Only copy `*.yaml` and `*.json`:

```
$ pyarmor cfg data_files="*.yaml *.json"
```

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/messages.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 the obfuscated scripts, so first make sure PyInstaller has been installed. If not, simple install it by this command:

```
$ pip install pyinstaller
```

Packing to one file

New in version 8.5.4.

Packing script to one file only need one command:

```
$ pyarmor gen --pack onefile foo.py
```

Run the final bundle:

```
$ dist/foo
```

Pyarmor will automatically obfuscate *foo.py* and all the other used modules and packages in the same path, then pack the obfuscated to one bundle.

Important: Please pass plain script in command line, for example, *foo.py* should not been obfuscated.

Packing to one folder

New in version 8.5.4.

Packing script to one folder:

```
$ pyarmor gen --pack onedir foo.py
```

Run the final bundle:

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

Using .spec file

New in version 8.5.8.

If the plain script has been packed by one spec file. For example:

```
$ pyinstaller foo.spec
$ dist/foo
```

Then pass this specfile to `--pack` to let Pyarmor pack the obfuscated scripts. For example:

```
$ pyarmor gen --pack foo.spec -r foo.py joker/
$ dist/foo
```

Note that all the other scripts or packages must be list after main script, otherwise they won't be obfuscated by this way.

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

3.1.4 Advanced Tutorial

Contents

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

```
import sys

def sum2(a, b):
    return a + b

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
```

(continues on next page)

(continued from previous page)

```
    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]:

```
$ 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:

[2] This feature only works for Python 3.9+
[3] This feature is only available for *Pyarmor Pro*.

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

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
```

(continues on next page)

(continued from previous page)

```
$ 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.

Note: This option doesn't touch any docstring

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/messages.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 =
```

(continues on next page)

(continued from previous page)

```
[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 Python versions

New in version 8.3.

This guide how to obfuscate the script *foo.py* which works with both Python 3.8 and 3.9.

First install Pyarmor for each Python version:

```
$ python3.8 -m pip install pyarmor
$ python3.9 -m pip install pyarmor
```

If you have Pyarmor license, register Pyarmor by any Python version:

```
$ python3.8 -m pyarmor.cli reg pyarmor-regfile-xxxx.zip
```

Enable builtin plugin `MultiPythonPlugin`:

```
$ python3.8 -m pyarmor.cli cfg plugins + "MultiPythonPlugin"
```

Obfuscate the script to different output path by each Python version:

```
$ python3.8 -m pyarmor.cli gen -O dist1 foo.py
$ python3.9 -m pyarmor.cli gen -O dist2 foo.py
```

Then merge 2 output paths by any Python version:

```
$ python3.8 -m pyarmor.cli.merge -O dist dist1 dist2
```

The final output path is `dist`:

```
$ python3.8 dist/foo.py
$ python3.9 dist/foo.py
```

Using shared runtime package

It's possible generating runtime package once and use it later.

First generate runtime package:

```
$ pyarmor gen runtime -O build/my_runtime1
```

Then obfuscate scripts with it:

```
$ pyarmor gen --use-runtime build/my_runtime1 foo.py
```

But it need copy shared runtime package to *dist* path:

```
# pyarmor_runtime_000000 need to replaced with real name
$ ls build/my_runtime1/
$ cp -a build/my_runime1/pyarmor_runtime_000000 dist/
```

The other options could be used to generate shared runtime package, for examples:

```
$ pyarmor gen runtime -e .10 -O build/my_runtime2
$ pyarmor gen --platform windows.x86_64,linux.x86_64 build/my_runtime3
```

If using *outer key* with runtime package, it need specify *–outer* both generating runtime package and obfuscating scripts:

```
$ pyarmor gen runtime --outer -O build/my_outer_runtime
$ pyarmor gen --outer --use-runtime build/my_outer_runtime foo.py

$ cp -a build/my_outer_runtime/pyarmor_runtime_000000 dist/
$ pyarmor gen key -e .10
$ mv dist/pyarmor.rkey dist/pyarmor_runtime_000000
```

Working with old runtime key

If still need check Pyarmor 7 runtime key in the obfuscated scripts of Pyarmor 9, here it's one possible solution

The idea is still using Pyarmor 7 obfuscated script to verify old runtime key, in Pyarmor 9 obfuscated script check old runtime key by calling Pyarmor 7 obfuscated script indirectly (IPC)

3.1.5 Customization and Extension

Contents

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`:

```
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:

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

```
$ 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)
        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()
```

(continues on next page)

(continued from previous page)

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

(continues on next page)

(continued from previous page)

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

(continues on next page)

(continued from previous page)

```
            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?

The following options could improve security

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

The following options hide module attributes

- `--private`
- `--restrict` also not allow plain script import obfuscated module

The following options prevent functions or modules from being 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`

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

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

- *--enable-rft*

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

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

Without the filter, all of the string constants in the scripts are encrypted, which 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 a 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 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* or *--restrict*
- *--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 unnecessary code here to another module could improve security.

3.2.2 Protecting Runtime Memory Data

Pyarmor focuses on protecting Python scripts, through several irreversible obfuscation methods, 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 the 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 *--pack* to implement this, and need move real code from main script to one module, because *--private* doesn't work for main script.

First configure necessary items[1]:

```
$ pyarmor cfg check_debugger=1 check_interp=1
```

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

Next pack the script by the following options[2]:

```
$ pyarmor gen --mix-str --assert-call --assert-import --private --pack onedir foo.py␣
↪real_foo.py
```

Then protect all the binary files in the output path `dist/foo/` through external tools, make sure these binary files can 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:

[2] 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.

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

...
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 8.3.

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.

First create script `.pyarmor/hooks/pyarmor_runtime.py`, and do some checks in the function *bootstrap()*. For example:

```
def bootstrap(user_data):
    from ctypes import windll
    if windll.kernel32.IsDebuggerPresent():
        print('found debugger')
        return False
```

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
```

(continues on next page)

(continued from previous page)

```
    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:
```

(continues on next page)

(continued from previous page)

```
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.

Changed in version 8.2.2: Do not use *--restrict* with *--pack*, it doesn't work.

When packing the scripts, Pyarmor could also protect system packages in the bundle. The idea is to list all the dependent modules and packages and obfuscate them too.

Here it's an example to protect system packages for script `foo.py`.

We need generate a file `file.list` list all the dependent modules and packages of `foo.py` by using PyInstaller features.

First generate `foo.spec`:

```
$ pyi-makespec foo.py
```

Then patch `foo.spec`:

```
a = Analysis(
    ...
)

# Patched by Pyarmor to generate file.list
_filelist = []
_package = None
for _src in sort([_src for _name, _src, _type in a.pure]):
    if _src.endswith('__init__.py'):
        _package = _src.replace('__init__.py', '')
        _filelist.append(_package)
    elif _package is None:
        _filelist.append(_src)
    elif not _src.startswith(_package):
        _package = None
        _filelist.append(_src)
```

(continues on next page)

(continued from previous page)

```
with open('file.list', 'w') as _file:
    _file.write('\n'.join(_filelist))
# End of patch
```

Next pack `foo.py` by PyInstaller and generate `file.list` at the same time:

```
$ pyinstaller foo.py
```

Finally repack the script with the following options:

```
$ pyarmor gen --assert-call --assert-import --pack dist/foo/foo foo.py @file.list
```

This example only guides how to do, please write your own patch script and use other necessary options to obfuscate scripts. For example, you could manually edit `file.list` to meet needs.

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 Using Pyarmor License

Contents

Prerequisite

First of all

1. One *activation file* of *Pyarmor License*, refer to *License Types* to purchase right one
2. One device has installed Pyarmor 9.0+
3. Internet connection
4. Product name which bind to this license

Initial registration

Any license need this step to request *registration file* from Pyarmor License Server by *activation file* like `pyarmor-regcode-xxxx.txt`:

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

Using *-p* to specify product name for this license, please replace "XXX" with real product name. For non-commercial use, replace it to `non-profits`.

If initial registration is successful, one *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 a product is in development, and the 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.

Using Pyarmor Basic or Pro

1. Refer to *Initial registration*, got *registration file* like *pyarmor-regfile-xxxx.zip*
2. Using *registration file* to register Pyarmor in other devices

Copy *registration file* to other machines, then run this 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.

This license can register Pyarmor on at most 100 devices

On each device it's enough to register Pyarmor once, do not register Pyarmor before each obfuscation

Do not register Pyarmor in the CI/CD pipeline or docker container by this *registration file*, each run will taken as one new device.

See also:

Using Pyarmor in CI Pipeline

Using CI License

New in version 9.0.

Refer to *Initial registration*, got *registration file* like *pyarmor-regfile-xxxx.zip*

Do not use `pyarmor-regfile-xxxx.zip` in CI/CD pipeline directly, it's only used to request CI regfile:

- In local device run the following command to request one CI regfile `pyarmor-ci-xxxx.zip`:

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

  Check CI license info in local machine:

  ```
  $ pyarmor --home temp reg pyarmor-ci-xxxx.zip
  ```

- In CI/CD pipeline, add 2 steps to register Pyarmor by CI regfile:

```
# Please replace "9.X.Y" with current Pyarmor version
pip install pyarmor=9.X.Y
pyarmor reg pyarmor-ci-xxxx.zip
```

  Check registration information in CI/CD pipeline:

```
pyarmor -v
```

Notes

- Do not request CI regfile in CI/CD pipeline
- CI regfile `pyarmor-ci-xxxx.zip` will be expired about in 360 days
- CI regfile may not work in future Pyarmor version
- Once CI regfile doesn't work, require new one
- One license can request <= 100 CI regfiles

Important: *Pyarmor CI* License doesn't work in local device

Even in the CI/CD pipeline, *Pyarmor CI* License also doesn't work in the runner which has its own disk. If the runner is not docker container, use *Pyarmor Pro* License instead.

See also:

Using Pyarmor in CI Pipeline

Check Device For Group License

Check one device works for group license by this way:

- First install Pyarmor 8.4.0+ trial version in this device
- Got machine id by the following command:

```
$ pyarmor reg -g 1
...
INFO     current machine id is "mc92c9f22c732b482fb485aad31d789f1"
INFO     device file has been generated successfully
```

- Reboot this device, check machine id is same or not
- If machine id is same after each reboot, group license works in this device. Otherwise group license doesn't work in this device.

For docker container, please check docker host as above. Only if docker host could work with group license, unlimited docker containers could be run in this docker host, refer to how-to/register section `run unlimited dockers in offline device`

If machine id of docker host is changed after reboot, group license doesn't work in any docker container

Most of physics machine, cloud server or VM like qemu, virtual box, vmware with same disk image work with Group license. Most of runners in CI/CD pipeline could not use Group License.

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.

Only the machine id of device is not changed after reboot, it could be used as group device. Most of physics machine, cloud server or VM like Qemu, Virtual box, Vmware with same disk image work with Group license. Refer to *Check Device For Group License*

The allocated device No. is never free, if a device is reinstalled, it need allocate new one.

Basic steps:

1. Using activation 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 activation 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:

```
$ pyarmor reg -g 1

INFO     Python 3.12.0
INFO     Pyarmor 8.4.7 (trial), 000000, non-profits
INFO     Platform darwin.x86_64
INFO     generating device file ".pyarmor/group/pyarmor-group-device.1"
INFO     current machine id is "mc92c9f22c732b482fb485aad31d789f1"
INFO     device file has been generated successfully
```

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

In order to make sure group license works for this device, reboot this device, and run this command again:

```
$ pyarmor reg -g 1

...
INFO     current machine id is "mc92c9f22c732b482fb485aad31d789f1"
...
```

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

Make sure this machine id is same after reboot.

Because group license is bind to device, so machine id should keep same after reboot. If it's changed after reboot, group license doesn't work in this device.

For VM machine, WSL(Windows Subsystem Linux) or any other system, please check the documentation to configure the network and harddisk, make sure network mac address and serial number of harddisk are fixed. If they're volatile, group license could not work in this system.

Generating offline device regfile

Generating offline device regfile needs an 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 or any computer which has internet connection and registration file, this file must be saved in the path `.pyarmor/group/`, then run the following 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
```

The device regfile `pyarmor-device-regfile-xxxx.1.zip` is bind to machine id in the device file `pyarmor-group-device.1`.

Note: If there are new versions which fix any bug that machine id is changed after this device reboot, it need generate new device file `pyarmor-group-device.2` for this device by new Pyarmor version, and generate new device regfile `pyarmor-device-regfile-xxxx.2.zip` by new Pyarmor version too.

Because device no. 1 has been used, so it need use next device no. 2, that is to say, one device may occupy more than one device no. Generally it should not be problem because there are 100 device no. available.

Registering Pyarmor in offline device

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

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

INFO     Python 3.12.0
INFO     Pyarmor 8.4.7 (trial), 000000, non-profits
INFO     Platform darwin.x86_64
INFO     register "/path/to/pyarmor-device-regfile-xxxx.1.zip"
INFO     machine id in group license: mc92c9f22c732b482fb485aad31d789f1
INFO     got machine id: mc92c9f22c732b482fb485aad31d789f1
INFO     this machine id matchs group license
INFO     This license registration information:

License Type    : pyarmor-group
License No.     : pyarmor-vax-006000
License To      : Tester
License Product : btarmor
```

(continues on next page)

(continued from previous page)

```
BCC Mode        : Yes
RFT Mode        : Yes

Notes
* Offline obfuscation
```

Note that this log says this device regfile is only for this machine id:

```
INFO     machine id in group license: mc92c9f22c732b482fb485aad31d789f1
```

And this log show machine id of this device:

```
INFO     got machine id: mc92c9f22c732b482fb485aad31d789f1
```

They must be matched, otherwise this device regfile doesn't work, it may need generate new device regfile for this device.

Check registration information:

```
$ pyarmor -v
```

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

Run unlimited dockers in offline device

New in version 8.3.

Group license supports unlimited dockers which uses default bridge network and not highly customized, the docker containers use same device regfile of host.

how it works

1. Each docker host is taken as an offlice device and must be registered as above.
2. Then start an auth-server in docker host to listen auth-request from docker container.
3. When run Pyarmor in docker container, it will send auth-request to auth-server in docker host, and verify the result returned from docker host.

Linux Docker Host

The practice for group license with unlimited docker containers:

- Docker host, Ubuntu x86_64, Python 3.8
- Docker container, Ubuntu x86_64, Python 3.11

The prerequisite in docker host:

- offline device regfile `pyarmor-device-regfile-xxxx.1.zip` as above
- Pyarmor 8.4.1+

First copy the following files to docker host:

- pyarmor-8.4.2.tar.gz
- pyarmor.cli.core-5.4.1-cp38-none-manylinux1_x86_64.whl

- pyarmor.cli.core-5.4.1-cp311-none-manylinux1_x86_64.whl
- pyarmor-device-regfile-6000.1.zip

Then run the following commands in the docker host:

```
$ python3 --version
Python 3.8.10

$ pip install pyarmor.cli.core-5.4.1-cp38-none-manylinux1_x86_64.whl
$ pip install pyarmor-8.4.1.tar.bgz
```

Next start `pyarmor-auth` to listen the request from docker containers:

```
$ pyarmor-auth pyarmor-device-regfile-6000.1.zip

2023-06-24 09:43:14,939: work path: /root/.pyarmor/docker
2023-06-24 09:43:14,940: register "pyarmor-device-regfile-6000.1.zip"
2023-06-24 09:43:15,016: listen container auth request on 0.0.0.0:29092
```

Do not close this console, open another console to run dockers.

For Linux container run it with extra `--add-host=host.docker.internal:host-gateway`:

```
$ docker run -it --add-host=host.docker.internal:host-gateway python bash

root@86b180b28a50:/# python --version
Python 3.11.4
root@86b180b28a50:/#
```

In docker host open third console to copy files to container:

```
$ docker cp pyarmor-8.4.1.tar.gz 86b180b28a50:/
$ docker cp pyarmor.cli.core-5.4.1-cp311-none-manylinux1_x86_64.whl 86b180b28a50:/
$ docker cp pyarmor-device-regfile-6000.1.zip 86b180b28a50:/
```

In docker container, register Pyarmor with same device regfile. For example:

```
root@86b180b28a50:/# pip install pyarmor.cli.core-5.4.1-cp311-none-manylinux1_x86_64.whl
root@86b180b28a50:/# pip install pyarmor-8.4.1.tar.gz
root@86b180b28a50:/# pyarmor reg pyarmor-device-regfile-6000.1.zip
root@86b180b28a50:/# pyarmor -v
```

If everything is fine, it should print group license information. And then test it with simple script:

```
root@86b180b28a50:/# echo "print('hello world')" > foo.py
root@86b180b28a50:/# pyarmor gen --enable-rft foo.py
```

When need to verify license, the docker container will send request to docker host. The *pyarmor-auth* console should print auth request from docker container, if there is no any request, please check docker network configuration, make sure IPv4 addresses of docker host and container are in the same network. For example, in docker container:

```
root@86b180b28a50:/# ifconfig -a

eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
      inet 172.17.0.2  netmask 255.255.0.0  broadcast 172.17.255.255
...
```

In docker host:

```
$ ifconig -a
docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
         inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
...
```

MacOS Docker Host

There is a little difference when docker host is MacOS, because docker container is running in Linux VM, not in MacOS directly.

So one solution is running *pyarmor-auth* in Linux VM, in this case, it should take this Linux VM as offline device, and generate device regfile for this Linux VM, not for **MacOS**, and start docker container with extra options:

```
$ docker run --add-host=host.docker.internal:172.17.0.1 ...
```

In this case, it may need some extra configuration for Linux VM to make sure the machine id could be fixed.

Refer to issue 1542 for more information.

Windows Docker Host

For Windows docker host, first check Windows network configuration:

```
C:> ipconfig

Ethernet adapter vEthernet (WSL):

     Connection-specific DNS Suffix  . :
     Link-local IPv6 Address . . . . . : fe80::8984:457:2335:588e%28
     IPv4 Address. . . . . . . . . . . : 172.22.32.1
     Subnet Mask . . . . . . . . . . . : 255.255.240.0
     Default Gateway . . . . . . . . . :
```

If there is IPv4 Address, for example `172.22.32.1`, which is in the same network as docker container, it's simple. Just take this Windows as offline device, and run *pyarmor-auth* on it, then start docker container with extra options:

```
$ docker run --add-host=host.docker.internal:172.22.32.1 ...
```

Anyway, *pyarmor-auth* must listen on any IPv4 address which is in the same network as docker container.

If there is no available IPv4 address in Windows, the other solution is running *pyarmor-auth* in WSL, in this case, WSL should be taken as offline device.

When something is wrong

1. Check docker container network:

```
root@86b180b28a50:/# ifconfig -a

eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
      inet 172.17.0.2  netmask 255.255.0.0  broadcast 172.17.255.255
...

root@86b180b28a50:/# ping host.docker.internal
PING host.docker.internal (172.17.0.1) 56(84) bytes of data.
64 bytes from host.docker.internal (172.17.0.1): icmp_seq=1 ttl=64 time=0.048 ms
...
```

If *ping* doesn't works, please check docker host network. If docker host is MacOS, it also checks Linux VM network. If docker host is Windows, also check WSL network.

And make sure IPv4 address of *host.docker.internal* is in same network as *eth0* which IPv4 address is *172.17.0.2*. In above example, it's *172.17.0.1*, so it's OK.

If not, also check docker host network. If docker host is MacOS, it may need run *pyarmor-auth* in Linux VM, not MacOS. If docker host is Windows, it may need run *pyarmor-auth* in WSL, not Windows.

Anyway, please configure the docker host/container network so that *pyarmor-auth* could listen in any IPv4 address which is in the same network as docker container.

2. Check docker host to make sure group license works:

```
$ pyarmor -d reg pyarmor-device-regfile-6000.1.zip
$ pyarmor -v
```

If run *pyarmor-auth* in Linux VM or WSL, please check group license could work in Linux VM or WSL. It may need generate new device regfile for Linux VM or WSL.

Using multiple Pyarmor Licenses in same device

Generally the registration information is sotred in the Pyarmor *Home Path*, the default value is `~/.pyarmor`. It means

- All Python virtual environments share same registration information
- It may not work to register other Pyarmor license in same device

When need many Pyarmor Licenses in one machine, set each license to different 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
```

What need to do after upgrading Pyarmor

Generally it need do nothing after upgrading Pyarmor, the registration information still works.

But in the following versions something is changed

- **Pyarmor 8.0**

  Old license for Pyarmor 7 doesn't work

  - Some old licenses can be upgraded to Basic License freely, refer to *upgrade old license*
  - Old license can't be upgraded to Pro or Group License

- **Pyarmor 9.0**

  A big change about using Pyarmor in CI/CD pipeline

  - *Pyarmor Basic*
    - *Upgrade to Pyarmor 9* freely
    - If using Pyarmor in CI/CD pipeline, refer to *Using Pyarmor in CI Pipeline*
  - *Pyarmor Pro*

    * If not using Pyarmor in CI/CD pipeline, *Upgrade to Pyarmor 9* freely
    * If using Pyarmor in CI/CD pipeline, 2 choices
        · Still use Pyarmor 8.x as before
        · Upgrade to Pyarmor 9, and purchase new *Pyarmor CI*

- *Pyarmor Group*

  It need generate device regfile for each offline device again by Pyarmor 9.0+, refer to *Upgrade to Pyarmor 9*

Upgrading old license

Not all the old license (Pyarmor 7) could be upgraded to latest version.

The old license could be upgraded to Pyarmor Basic freely only if it matches these conditions:

- Following new Pyarmor EULA
- The license no. starts with `pyarmor-vax-`
- The original activation file `pyarmor-regcode-xxxx.txt` exists and isn't used more than 100 times
- The old license is purchased before June 1, 2023. In principle, the old license purchased after Pyarmor 8 is available could not be upgraded to new license.

If failed to upgrade the old license, please purchase new license to use Pyarmor latest version.

The old license can't be upgraded 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.

Upgrade to Pyarmor 9

1. *Pyarmor Basic* and *Pyarmor Pro*

   If Pyarmor License has been registered in this device

   - First upgrade to Pyarmor 9:

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

   - When first time to obfuscate scripts, it will show hints:

   ```
   $ pyarmor gen foo.py

   ...
   Pyarmor 9 has big change on CI/CD pipeline
   If not using Pyarmor License in CI/CD pipeline
   Press "c" to continue
   Otherwise press "h" to check Pyarmor 9.0 Upgrade Notes

   Continue (c), Help (h), Quit (q):
   ```

   - Just press `c` to continue, there is no prompt later

   If Pyarmor License isn't registered in this device

   - First use *activation file* to generate new *registration file*:

   ```
   $ pip install -U pyarmor

   # Please replace XXX with real product name
   $ pyarmor reg -p XXX pyarmor-regcode-xxxx.txt
   ```

   - Save and backup new *registration file* `pyarmor-regfile-xxxx.zip`
   - Use this new regfile to register Pyarmor in other new device:

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

   If *activation file* is used too many times, please first install Pyarmor 8, then upgrade to Pyarmor 9

2. *Pyarmor Group* License

   It need generate device regfile again with Pyarmor 9.0+

   - First upgrade to Pyarmor 9:

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

   - Then generate device regfile as before

     For example, generate device regfile `pyarmor-device-regfile-6000.1.zip` for device no. 1:

   ```
   $ pyarmor reg -g 1 /path/to/pyarmor-regfile-6000.zip
   ```

- Finally, use new one to register Pyarmor in offline device:

```
$ pyarmor reg pyarmor-device-regfile-6000.1.zip
```

3.2.9 Using Pyarmor in CI Pipeline

Trial Version could be used in CI/CD pipeline by one step:

```
pip install pyarmor
```

For *Pyarmor Basic* and *Pyarmor CI* License

- Refer to *Initial registration*, first got *registration file* like `pyarmor-regfile-xxxx.zip`
- In local device run the following command to request one CI regfile `pyarmor-ci-xxxx.zip`:

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

  Check CI license info in local machine:

  ```
  $ pyarmor --home temp reg pyarmor-ci-xxxx.zip
  ```

- In CI/CD pipeline, add 2 steps to register Pyarmor by CI regfile:

  ```
  # Please replace "9.X.Y" with current Pyarmor version
  pip install pyarmor=9.X.Y
  pyarmor reg pyarmor-ci-xxxx.zip
  ```

  Check registration information in CI/CD pipeline:

  ```
  pyarmor -v
  ```

Notes

- Do not request CI regfile in CI/CD pipeline
- CI regfile `pyarmor-ci-xxxx.zip` will be expired about in 360 days
- CI regfile may not work in future Pyarmor version
- Once CI regfile doesn't work, require new one
- One license can request <= 100 CI regfiles

Note: In GitHub Action, it need one extra step, otherwise *CI license only works in CI/CD pipeline*

1. For Ubuntu, add this step:

   ```
   - run: sudo mv /dev/disk /dev/disk-none
   ```

2. For Darwin, add this step:

   ```
   - run: sudo mv /dev/rdisk0 /dev/rdisk0-none
   ```

Refer to this thread Error when using CI license in CI pipeline

---

Important: In CI/CD pipeline, each run will send license and docker information to Pyarmor License Server, excessive requests or requests beyond normal usage may be rejected by Pyarmor License Server

It's not allowed to install Pyarmor in your customer's docker image

Pyarmor Pro and *Pyarmor Group* License can't be used in CI/CD pipeline directly, but there is one workaround

- First obfuscate the scripts in local device and store them to another branch like *master-obf*
- Then in CI/CD pipeline to check this new branch

Here is an example, suppose test-project locates at *https://github.com/dashingsoft/test-project*, the directory tree as follows:

```
$ tree test-project

test-project
└── src
    ├── main.py
    ├── utils.py
    └── parent
        ├── child
        │   └── __init__.py
        └── __init__.py
```

In local device the scripts are obfuscated and are stored into another branch:

```
$ git clone https://github.com/dashingsoft/test-project

$ pip install pyarmor
$ pyarmor reg /path/to/pyarmor-regfile-5068.zip

# Create new branch
$ git checkout -B master-obf

# Create output path "dist" link to project path
$ ln -s test-project dist

# Obfuscate the script to "dist", which is same as "test-project"
# So "dist/src/main.py" is same as "test-project/src/main.py"
$ pyarmor gen -O dist -r --platform windows.x86_64,linux.x86_64,darwin.x86_64 test_
↪project/src

# Add runtime package to this branch
$ git add -f test_project/pyarmor_runtime_5068/*

# Commit the results
$ git commit -m'Build obfuscated scripts' .

# Push new branch to remote server
$ git push -u origin master-obf
```

In CI/CD pipeline, it need not install Pyarmor, just checkout branch *master-obf*, and work as before.

3.2.10 Work with Third-Party Libraries

Contents

There are countless big packages in the Python world, many packages I never use and which I 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 on my local machine.

Pyarmor provides rich options to meet various needs, for complex applications, please spend some time checking *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 and make it work 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 uses 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, if 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
   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 the list of problem libraries and possible solutions. You are welcome to create a 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
Cython	Should work with restrict_module = 0	

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

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)
```

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
```

Next use default options to obfuscate the scripts:

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

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

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

Note: It may requires v9.0.8+ and non-trial version. Because Nuitka will convert package *pyarmor_runtime_000000/__init__.py* to *pyarmor_runtime_000000_init_.py*, it also results in `RuntimeError: unauthorized use of script`, this is fixed in v9.0.8

---

streamlit

It need change default configurations. At least:

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

This first one solves issue *RuntimeError: unauthorized use of script (1:1102)*

Then second one solves issue *RuntimeError: the format of obfuscated script is incorrect (1:1082)*

Now obfuscate the scripts:

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

It may still not work because of Streamlit may patch code object by itself

Cython

Here it's an example show how to *cythonize* a python script *foo.py* obfuscated by pyarmor:

```
print('Hello Cython')
```

First obfuscate it with some extra options:

```
$ pyarmor cfg restrict_module=0
$ pyarmor gen foo.py
$ ls dist/
foo.py pyarmor_runtime_000000
```

The obfuscated script and runtime files will be saved in the path *dist*

Next *cythonize* the obfuscated script *dist/foo.py* to *foo.c*:

```
$ cd dist
$ cythonize -3 foo.py
```

Then compile *foo.c* to the extension modules(it may need extra cfalg `-fPIC` in some platforms):

```
$ gcc -shared $(python-config --cflags) $(python-config --ldflags) \
      -o foo$(python-config --extension-suffix) foo.c
```

Finally test it, remove *dist/foo.py* and import the extension module:

```
$ rm foo.py
$ python -c 'import foo'
```

It will print *Hello Cython* as expected.

3.3 References

3.3.1 Concepts

Activation File
A text file used for 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
- **FreeBSD**
  - freebsd.x86_64
- **Alpine Linux (musl-c)**
  - alpine.x86_64
  - alpine.aarch64
- **Android**
  - android.x86_64
  - android.x86
  - android.aarch64
  - android.armv7

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 CI
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*

Since Pyarmor 8.3, *pyarmor.cli.runtime* is split into serval packages:

- *pyarmor.cli.core.freebsd*
- *pyarmor.cli.core.android*
- *pyarmor.cli.core.windows*
- *pyarmor.cli.core.themida*
- *pyarmor.cli.core.linux*
- *pyarmor.cli.core.alpine*
- *pyarmor.cli.core.darwin*

All of them are published in the PyPI

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 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	generate debug log file ...
--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

Generate debug log `pyarmor.debug.log`

When something is wrong, use this option to generate `pyarmor.debug.log` to get more information. For example:

```
$ pyarmor -d gen foo.py
$ cat pyarmor.debug.log
```

--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 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 . . .

--exclude PATTERN exclude scripts or paths . . .

-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 <onefile|onedir|FC|DC|NAME.spec> Obfuscate scripts then pack ...

--use-runtime PATH use shared runtime package ...

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.

Since 8.2.2, it also supports list all the scripts and packages in one file, and pass it with prefix `@`. For example:

```
pyarmor gen -r @filelist
```

The content of `filelist` includes 2 scripts and 2 packages:

```
src/foo.py
src/utils.py
libs/dbpkg
libs/config
```

-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.

--exclude PATTERN

Exclude scripts or paths, use this option many times to exclude more

The pattern is same as the Python standard library fnmatch

Exclude one exact script:

```
$ pyarmor gen --exclude "src/test.py" src
```

Exclude one exact path:

```
$ pyarmor gen -r --exclude "./test" .
```

Exclude `test.py` in any path:

```
$ pyarmor gen -r --exclude "*/test.py" src
```

Exclude any `test` path:

```
$ pyarmor gen -r --exclude "*/test" src
```

-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 2 forms:

- A number stands for valid days
- A date with ISO format `YYYY-MM-DD`

For example:

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

It will check local time by default. Check the default server by this command:

```
$ pyarmor cfg nts
...
Current settings
  nts = local
...
```

If need to check network time, just configure *nts* to remote server. For example:

```
$ pyarmor cfg nts=pool.ntp.org
```

Before v8.8.4, only supports NTP protocol, the default server can be changed to any valid NTP server. For example:

```
$ pyarmor cfg nts=108.59.2.24
```

Since v8.8.4, it also supports HTTP server, and multiple servers. If the first server doesn't work, then uses the second, and so on. When using HTTP protocol, just provide one valid URL. For example:

```
$ pyarmor cfg nts=http://worldtimeapi.org/api
```

The following example uses multiple servers, both NTP and HTTP:

```
$ pyarmor cfg nts=pool.ntp.org,http://worldtimeapi.org/api
```

And special name *local* could be used to get local time. For exmaple:

```
$ pyarmor cfg nts="pool.ntp.org,http://worldtimeapi.org/api,local"
```

-b DEV, **--bind-device** DEV

Bind obfuscated script to specified device

Use this option multiple times to bind multiple machines

Using *pyarmor-7 hdinfo* to get hardware information.

Since Pyarmor 8.4.6, *python -m pyarmor.cli.hdinfo* works too:

```
Machine ID: 'mc92c9f22c732b482fb485aad31d789f1'
Default Harddisk Serial Number: 'HXS2000CN2A'
Default Mac address: '00:16:3e:35:19:3d'
Default IPv4 address: '128.16.4.10'
Domain: 'dashingsoft.com'
```

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
```

Since Pyarmor 8.5.0, it also supports machine id and domain name got by *python -m pyarmor.cli.hdinfo*. For example:

```
$ pyarmor gen -b mc92c9f22c732b482fb485aad31d789f1 foo.py
$ pyarmor gen -b "{dashingsoft.com}" 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.

In this case, try to call one empty function in loop statement. For example:

```
 def pyarmor_check_license():
     pass

 def main():
     while True:
         check_pyarmor_license()
         sleep(0.01)

Besides, if bcc mode is enabled, it also need exclude this empty function. For example::

  $ pyarmor cfg bcc:excludes = pyarmor_check_license
```

--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 attributes of the obfuscated scripts could not be seen by plain script or Python interpreter.

Note that main script is always not private, otherwise the obfuscated script can't be executed by Python interpreter. If need protect the attributes of main script, move real code to one module. For example, suppose the original script is *foo.py*:

```
def main():
    print('This is main code')

if __name__ == '__main__':
    main()
```

Just copy it to *real_foo.py*, then create new *foo.py* like this:

```
from real_foo import main

if __name__ == '__main__':
    main()
```

And obfuscate them:

```
$ pyarmor gen --private foo.py real_foo.py
```

Changed in version 8.5.3: In previous versions, plain script could not import the module obfuscated by *–private*, now plain script could import the obfuscated module, but can't visit its attributes.

--restrict

Enable restrict mode for package

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, it's used to obfuscate attribute name in chains. For example:

```
obj.attr          ==> getattr(obj, 'xxxx')
obj.attr = value  ==> setattr(obj, 'xxxx', value)
```

Generally when RFT Mode is available, it need not this option.

--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

This option doesn't touch any docstring

It may reduce performance if there are too many strings, in this case, only mix important strings by filter.

See also:

Filter mix string in *Advanced Tutorial*

--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
```

(continues on next page)

(continued from previous page)

```
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 <onefile,onedir,FC,DC,NAME.spec>

Obfuscate script first, then pack the obfuscated scripts to bundle

Before v8.5.4, user need first generate an executable file by PyInstaller

Now everything is done by Pyarmor

The old method still works, but it's deprecated.

Changed in version 8.5.4: supports *onefile* and *onedir*

Changed in version 8.5.8: supports specfile

Once this option is set to *onefile* or *onedir*, pyarmor will analysis the source of main script, and find all the imported modules and packages which are in the same path of main script. All of these used modules and packages will be obfuscated automatically

Then pyarmor will call PyInstaller to pack the obfuscated scripts to one file or one folder bundle.

For example, generate one file bundle:

```
$ pyarmor gen --pack onefile foo.py
$ ls dist/
```

Sometimes it need specify option *-r* to make sure all the child packages are obfuscated. For example:

```
$ pyarmor gen --pack onedir -r foo.py
```

PyInstaller will ask for confirm if output path exists, if need remove output path siliently, use mode *FC* or *DC*. Here *F* stands for *onefile*, *D* stands for *onedir*, *C* stands for clean output siliently. For example:

```
$ pyarmor gen --pack FC foo.py
```

If plain script has been packed by one *.spec* file, it could be used by *--pack* to pack the obfuscated script. For example:

```
$ pyarmor gen --pack foo.spec -r foo.py util.py joker/
```

Note that by this way Pyarmor only obfuscates the scripts in the command line, so specify all the scripts and packages need to be obfuscated.

See also:

Insight Into Pack Command

--use-runtime PATH

Use shared runtime package at PATH

The runtime package must be generated by *pyarmor gen runtime*

If using *outer key*, *--outer* must be specified both in command *pyarmor gen runtime* and *pyarmor gen*

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.[1]
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.

Special output **pipe**

If ouptput path is `pipe`, the generated key is not save to file, but return the key content (bytes) directly.

Generally it's used to generate runtime key by web api and send key to customer by internet.

For example,

```
from pyarmor.cli.__main__ import main_entry

args = ['gen', 'key', '-O', 'pipe', '-e', '2023-10-21']
data = main_entry(args)

with open('pyarmor.rkey', 'wb') as f:
    f.write(data)
```

pyarmor gen runtime

Generate shared *runtime package*.

Syntax

pyarmor gen runtime <options>

Options

-O PATH, --output PATH output path

--outer enable outer runtime key

--platform NAME cross platform obfuscation

-e DATE, --expired DATE set expired date

[1] If runtime package supports multiple Python versions and multiple platforms, it need copy key file to each sub-folder *pyXY* in the runtime package or configure outer_keyname with prefix *../*, for example, *pyarmor cfg outer_keyname=../pyarmor.rkey*. Refer to issue 1599

--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 shared *runtime package* and store it to `-O`, the options in this command have same meaning as in the *pyarmor gen*. For example:

```
$ pyarmor gen runtime -O build/my_runtime1
$ ls build/my_runtime1/

$ pyarmor gen --use-runtime build/my_runtime1 foo.py
$ cp -a build/my_runime1/pyarmor_runtime_000000 dist/
```

It also uses other options to generate shared runtime package:

```
$ pyarmor gen runtime -e .10 --bind-device 10:52:fa:2d:26 -O build/my_runtime2
$ pyarmor gen runtime --platform windows.x86_64 -e .10 -O build/my_runtime3
```

If shared *runtime package* is generated by `--outer`, also obfuscate scripts by `--outer`:

```
$ pyarmor gen runtime --outer -O build/my_outer_runtime
$ pyarmor gen --outer --use-runtime build/my_outer_runtime foo.py

$ cp -a build/my_outer_runtime/pyarmor_runtime_000000 dist/
$ pyarmor gen key -e .10
$ mv dist/pyarmor.rkey dist/pyarmor_runtime_000000
```

Please replace `pyarmor_runtime_000000` with real name

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, it must leave blank around =:

```
$ 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.*/
```

Note that in Windows Command Box, it may need quota join char. For example:

```
$ pyarmor cfg rft_excludes "^" "/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 needs an internet connectection for 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

pyarmor init

New in version 9.1.

Refer to https://eke.dashingsoft.com/pyarmor/docs/en/user/man.html

pyarmor env

New in version 9.1.

Refer to https://eke.dashingsoft.com/pyarmor/docs/en/user/man.html

pyarmor build

New in version 9.1.

Refer to https://eke.dashingsoft.com/pyarmor/docs/en/user/man.html

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
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	3.0~3.6	3.7~3.10	3.11	3.12	3.13	Remark
pyarmor 8 RFT Mode	No	No	Y	Y	Y	Y	[1]
pyarmor 8 BCC Mode	No	No	Y	Y	Y	Y	
pyarmor 8 others	No	No	Y	Y	Y	Y	
pyarmor-7	Y	Y	Y	No	No	No	

Supported platforms

Table 3: Table-2. Supported Platforms (1)

OS	Windows	Apple[2]		Linux[3]			
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[4]	Y	Y	Y	Y	Y	Y	Y

Table 4: Table-3. Supported Platforms (2)[5]

OS	FreeBSD	Alpine Linux		Android			
Arch	x86_64	x86_64	aarch64	x86/x86_64	aarch64	armv7	armv6
pyarmor 8 RFT Mode	Y	Y	Y	Y	Y	Y	No
pyarmor 8 BCC Mode	Y	Y	Y	Y	Y	Y	No
pyarmor 8 others	Y	Y	Y	Y	Y	Y	No
pyarmor-7	Y	Y	Y	Y	Y	Y	Y

[1] `N/y` means not yet now, but will be supported in future.
[2] Apple Silcon only supports Python 3.9+
[3] This Linux is built with glibc
[4] pyarmor-7 also supports more linux arches, refer to Pyarmor 7.x platforms.
[5] These platforms are introduced in Pyarmor 8.3

Table 5: Table-4. Supported Platforms (3)[Page 88, 6]

OS	Linux	Linux, Alpine Linux
Arch	loongarch64	ppc64le, mips32el/64el, riscv64[Page 88, 7]
pyarmor 8 RFT Mode	Y	Y
pyarmor 8 BCC Mode	N	N
JIT Feature	N	Y
pyarmor 8 others	Y	Y

notes

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:

[6] The prebuilt extensions for these arches are published in the package pyarmor.cli.core.linux and pyarmor.cli.core.alpine respectively since Pyarmor 8.5.9 (not tested)

[7] For riscv64, only support Python 3.10+

class PluginName

static post_script(*ctx*, *res*, *source*)

This method is optional.

This method is called after each script has been obfuscated

Generally it's used to patch the obfuscated *source*, and return patched *source*

Parameters

- **ctx** (*Context*) – building context
- **res** (*FileResource*) – instance of *pyarmor.cli.resource.FileResource*
- **source** (*str*) – the source of obfuscated script

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__()*

Special hook script

New in version 8.3.

If want to do something before obfuscated scripts are executed, it need use a special hook script `.pyarmor/hooks/pyarmor_runtime.py`, it will be called when initializing Python extension *pyarmor_runtime*.

First create script `.pyarmor/hooks/pyarmor_runtime.py` and define all in the hook function *bootstrap()*, only this function will be called.

bootstrap(*user_data*)

Parameters
user_data (*bytes*) – user data in runtime key

Returns
False, quit and raise protection exception Any others, continue to execute the obfuscated scripts.

Raises

- **SystemExit** – quit without traceback
- **ohter Exception** – quit with traceback

An example script:

```
def bootstrap(user_data):
    # Import everything in the function, not in the module level
    import sys
    import time
    from struct import calcsize

    print('user data is', user_data)

    # Check platform
    if sys.platform == 'win32' and calcsize('P'.encode()) * 8 == 32:
        raise SystemExit('no support for 32-bit windows')

    # Check debugger in Windows
    if sys.platform == 'win32':
        from ctypes import windll
        if windll.kernel32.IsDebuggerPresent():
            print('found debugger')
```

(continues on next page)

(continued from previous page)

```
            return False

    # In this example, user_data is timestamp
    if time.time() > int(user_data.decode()):
        return False
```

Check it, first copy this script to `.pyarmor/hooks/pyarmor_runtime.py`:

```
$ pyarmor gen --bind-data 12345 foo.py
$ python dist/foo.py

user data is b'12345'
Traceback (most recent call last):
  File "dist/foo.py", line 2, in <module>
  ...
RuntimeError: unauthorized use of script (1:10325)
```

If need query hardware information, the simple way is to import Pyarmor extension *pytransform3*. For example, in Windows, copy the corresponding *pytransform3.pyd* to target device, then get machine id by this way

```
def bootstrap(user_data):
    from pytransform3 import get_hd_info
    # Refer to pyarmor/cli/get_hd_info.py
    print('Machine ID: %s' % get_hd_info(22).decode())
```

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 unused

Returns
arg == 4 return domain 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, function or method 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 are all the list of 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 6: 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 7: 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 requests 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.
no registration code found in pyarmor-regcode-xxxx.txt	Download *pyarmor-regcode-xxxx.txt* again, check its content, make sure it's same as email body

update license token failed	If run register command more than 3 times in 1 minute, wait for 5 minutes, and try again.
	If more than 100 runs in different devices or docker containers in 24 hours, please wait until any token is released
	If the date time of this device is not correct, restore it to current date
	If not, try to open *http://pyarmor.dashingsoft.com//api/auth2/* in web browser
	If the page says *NO:missing parameters*, it means network is fine, and license server is fine.
	If Pyarmor is prior to v8.5.3, upgrade Pyarmor to v8.5.3+, then check Python interpreter by the following commands:
	`$ python`
	`>>> from urllib.request import urlopen`
	`>>> res = urlopen('http://pyarmor.dashingsoft.com//api/auth2/')`
	`>>> print(res.read())`
	`b'NO:missing parameter'`
	If not return this, but raises exception, it's firewall problem, please configure it to allow Python interpreter to visit *pyarmor.dashingsoft.com:80*

Runtime Errors

Error messages reported by pyarmor

If it has an error code, it could be customized.

Table 8: 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. 3. Upgrade to Pyarmor 8.4.4+, then check network time by http server. For example, set time server by this command *pyarmor cfg nts=http://your.http-server.com/api/v2/*
	Protection Exception	If using `--assert-call` or `assert-import`, check section *Filter assert function and import* in the *Advanced Tutorial*, ignore those problem functions and modules by the traceback.

Error messages reported by Python interpreter

Generally they are not pyarmor issues. Please consult Python documentation or google error message to fix them.

Table 9: 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 is a list of some outer errors. Most of them are caused by missing some system libraries, or unexpected configuration. It has nothing to do with 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
```

ImportError: libdl.so: cannot open shared object file: No such file or directory

When running obfuscated scripts in unmatched platform, it may raise this error.

In this case checking dependencies by *ldd /path/to/pyarmor_runtime.so* to make sure it works. If not, please select right *–platform* to obfuscate the scripts.

For example, when obfuscating the scripts in Linux with target platform Termux, somethimes it need specify *–platform linux.aarch64*, not *–platform android.aarch64*, more information refer to issue 1674

No such file or directory: 'nul'

Generally something is wrong with Windows System.

Try to create NUL device again by this command:

```
sc create null binpath=C:\Windows\System32\drivers\null.sys type=kernel start=auto␣
↪error=normal
```

Then start null:

```
sc start null
```

If it works, this problem should be fixed.

If not, please google the solution. It's Windows System issue.

3.3.6 Pyarmor Check List

Pyarmor team handles too many wrong usage issues, all of them are collected and organized into this document. When something is wrong, please check this doc at first.

Build Device

If something is wrong in build device, check this section.

Bootstrap failed

1. Check package *pyarmor.cli.core* has been installed

   Search extension module *pytransform3.pyd* or *pytransform3.so* in *pyarmor* package source

   If not exists, please check installation documentation to install it

2. Check extension *pytransform3* is exact for this Python and platform

   Test it by Python interpreter:

   ```
   $ python
   >>> from pyarmor.cli.core import Pytransform3
   >>> Pytransform3.version()
   1
   >>>
   ```

   If not return this, check the dependencies of extension *pytransform3*

   In Linux

   ```
   $ ldd /path/to/pytransform3.so
   ```

   In MacOS

   ```
   $ codesign -v /path/to/pytransform3.so
   $ otool -L /path/to/pytransform3.so
   ```

   In Windows, download [cygcheck] then:

   ```
   C:\> cygcheck \path\to\pytransform3.pyd
   ```

   Or:

```
C:\> dumpbin /dependents \path\to\pytransform3.pyd
```

Try to install missed packages by error report to solve the problem

3. If nothing works, please check *Building Environments* to make sure Pyarmor supports this platform

   For LINUX variant system, check the content of packages like *pyarmor.cli.core.linux*, *pyarmor.cli.core.alpine* , *pyarmor.cli.core.android* etc. Each package includes many prebuilt *pytransform.so*. For example:

```
$ unzip ./pyarmor.cli.core.linux-6.5.3-cp310-none-any.whl

Archive:  ./pyarmor.cli.core.linux-6.5.3-cp310-none-any.whl
  inflating: pyarmor/cli/core/linux/__init__.py
  inflating: pyarmor/cli/core/linux/aarch64/pyarmor_runtime.so
  inflating: pyarmor/cli/core/linux/aarch64/pytransform3.so
  inflating: pyarmor/cli/core/linux/armv7/pyarmor_runtime.so
  inflating: pyarmor/cli/core/linux/armv7/pytransform3.so
  inflating: pyarmor/cli/core/linux/loongarch64/pyarmor_runtime.so
  inflating: pyarmor/cli/core/linux/loongarch64/pytransform3.so
  inflating: pyarmor/cli/core/linux/mips32el/pyarmor_runtime.so
  inflating: pyarmor/cli/core/linux/mips32el/pytransform3.so
  ...
```

   Check each *pytransform3.so* by *ldd* to find which one works, then copy them to package *pyarmor.cli.core*. For example:

```
$ cp pyarmor/cli/core/linux/loongarch64/*.so /path/to/pyarmor/cli/core
$ pyarmor gen foo.py
```

   Or install this package and set environment variable like this:

```
$ pip install ./pyarmor.cli.core.linux-6.5.3-cp310-none-any.whl
$ export PYARMOR_PLATFORM=linux.loongarch64
$ pyarmor gen foo.py
```

4. If this platform is supported, try to upgrade Python interpreter to latest patch version. For example, upgrade Python 3.11.0b2 to 3.11.9

Registration Failed

If it's using *Activation File* (*pyarmor-regcode-xxxx.txt*), make sure this file is not used more than 3 times, generally once initial registration completed, activation file `pyarmor-regcode-xxxx.txt` is invalid. It should use *Registration File* `pyarmor-regfile-xxxx.zip` for any next registration.

Basic/Pro License

1. If the date time of this device has been changed, restore it to current time
2. If run register command more than 3 times in 1 minute, wait for 5 minutes, and try again.
3. If run more than 3 docker containers in 1 minute in same docker host, run only one at the same time.
4. If have 100 runs in different devices or docker containers in 24 hours, waiting for 24 hours since the first run
5. If not, try to open http://pyarmor.dashingsoft.com/api/auth2/ in web browser or test it by *curl*

   If return "NO:missing parameters", it means network is fine, and license server is fine

Otherwise check network configuration

6. Check Python interpreter by the following commands (If there are many Python installed, make sure this Python interpreter is used to execute Pyarmor)

```
$ python
>>> from urllib.request import urlopen
>>> res = urlopen('http://pyarmor.dashingsoft.com/api/auth2/')
>>> print(res.read())
b'NO:missing parameter'
```

If it raises exception or return something else, it's firewall problem, please configure firewall to allow Python interpreter to visit pyarmor.dashingsoft.com at port 80 or 443

If it returns as above, but still failed to register, report issue with license no. like *pyarmor-vax-5068*

Group License

1. Check this offline device Machine ID is changed or not after reboot
   - It should make sure Pyarmor > 8.5.2, it's better to use the latest Pyarmor
   - Group License doesn't work in CI/CD pipeline with default runner
2. It's device registration file `pyarmor-device-regfile-????.N.zip` to be used to register Pyarmor in offline device. Do not use Group License *Registration File* `pyarmor-regfile-xxxx.zip` to register Pyarmor in the offline device
3. Does device registration file `pyarmor-device-regfile-????.N.zip` match this device? Each device registration file has one machine id.
4. If device Machine ID is same after reboot, and device registration file is matched this device, please report issue with the following information:
   - Machine ID
   - Device type, it's physics device, vm, ECS, docker container or something else
   - Linux, Windows or MacOS and arches
   - For Linux/MacOS, also provide the output of *uname -a*

Group License for docker container

1. Check docker host and container network, make sure they're in same network

Obfuscation failed

update license token failed

Please check above section *Registration Failed*

raise other exceptions

1. Try to obfuscate simple hello world script

   If not, check above section *Bootstrap Failed*, make sure Pyarmor supports this platform

2. Try to ignore local and global configuration. For example

   Rename path *.pyarmor* to *.pyarmor.bak*

   Rename path *~/.pyarmor/config* to *~/.pyarmor/config.bak*

   Then try to obfuscate the scripts

3. Try to use few options, check it works or not, and get the problem option
4. When report issue, please use debug option `-d` to generate *pyarmor.report.bug*, and
   - A script as simple as possible to reproduce issue
   - Do not use the options which doesn't make sense for this issue

Packing failed

1. Try to pack the original script by PyInstaller directly first, make sure it works and the final bundle works
2. Check *Insight Into Pack Command*

Target Device

If your own code in the obfuscated script still isn't executed, check *Bootstrap failed*, otherwise check *failed to run obfuscated scripts* or *failed to run the packed obfuscated scripts*

Bootstrap failed

1. Check is there Runtime Package

   In the output *dist* path, search *pyarmor_runtime.pyd* or *pyarmor_runtime.so*

   Runtime Package is a normal Python package, use it as third-party package, make sure it's in right place so that the obfuscated script could import it

2. Check the import statement in the obfuscated scripts

   Open the obfuscated script by any text editor, you can see the first statement is *from ... import*, make sure it works by Python import system

   If it doesn't work, try to use `-i` or `--prefix` to generate the obfuscated scripts again to fix it

3. Try to upgrade Python interpreter to latest patch version. For example, upgrade Python 3.11.0b2 to 3.11.9

If target device is different from build device

1. Make sure Python version (major.minor) is same in build/target device
2. If OS or arch is not same, make sure the right cross platform option `--platform` is used
3. Check extension *pyarmor_runtime* works in target machine

   In Linux

   ```
   $ ldd /path/to/pyarmor_runtime.so
   ```

   In MacOS

   ```
   $ codesign -v /path/to/pyarmor_runtime.so
   $ otool -L /path/to/pyarmor_runtime.so
   ```

   In Windows, download [cygcheck] then:

   ```
   C:\> cygcheck \path\to\pyarmor_runtime.pyd
   ```

   Or:

```
C:\> dumpbin /dependents \path\to\pyarmor_runtime.pyd
```

Try to install missed packages by error report to solve the problem

If still not work, please check *Building Environments* to make sure Pyarmor supports this platform

unauthorized use of script

1. Do not use restrict options like `--private`, `--restrict`, `--assert-call`, `--assert-import`
2. Use *pyarmor cfg assert.call:excludes "xxx"* and *pyarmor cfg assert.import:excludes "xxx"* to exclude problem modules and functions
3. Find the problem option, and report issue

Failed to run obfuscated scripts

1. Do not use `--enable-rft`, `--enable-bcc` and any restrict options like `--private`, `--restrict`, `--assert-call`, `--assert-import`, check the obfuscated scripts work or not. If it works, check the solutions in the **RFT Mode Problem**, **BCC Mode Problem** and **unauthorized use of script**
2. If it doesn't work, try to obfuscate one simple script, check it works or not
3. Add some print statement in the problem script, and get one script as simple as possible to reproduce the problem. It's better only use Python system packages. If really need third-party library, check *Work with Third-Party Libraries* first
4. Report issue with necessary information
   - Minimum pyarmor options, do not submit non-sense options
   - A simple script which only imports Python system package
   - How to run the obfuscated scripts and full trackback when it fails

RFT Mode Problem

Check solutions in *Using rftmode pro*

BCC Mode Problem

First use a hello world script to make sure it works. If it doesn't work, check your configuration and try again in clean environments

Check solutions in *Using bccmode pro*

Failed to run the packed obfuscated scripts

1. Do not pack the script, just use same options to obfuscate the script, and run the obfuscated script in target device, make sure it works, otherwise check solutions in above section
2. Do not obfuscate the scripts, pack the original script by PyInstaller directly, and execute the final executable in target device, make sure it works. Otherwise check PyInstaller documentation to find solutions
3. Try to use a few options to repeat check point 1 and 2, find the problem option, and report issue

Platform Issues

Darwin Apple Silicon may need codesign if pyarmor or the obfuscated script can't run at all

Notes

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`:

```
from pyarmor_runtime_000000 import __pyarmor__
__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. Also refer to option `--use-runtime` and command *pyarmor gen runtime* for using shared runtime package.

See also:

Changing runtime package name in the chapter *Customization and Extension*

Using shared runtime package in the chapter *Advanced Tutorial*

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.

See also:

command *pyarmor gen key*

Restrict modes

By default the obfuscated scripts can't be changed.

After using *--private*, the attributes of the obfuscated scripts could not be seen by plain script or Python interpreter.

After using *--restrict*, the attributes of the obfuscated scripts are protected as *--private*, and the obfuscated scripts could not be imported by plain script or Python interpreter,

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.
- Importing obfuscated module by *importlib.util.spec_from_file_location* need extra handle, refer to issue 846

See also:

Work with Third-Party Libraries

Generating cross platform scripts and *Obfuscating scripts for multiple Python versions* in the chapter *Advanced Tutorial*

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 has no pack feature, it need call PyInstaller to pack the obfuscated script to final bundle, so first install PyInstaller:

```
$ pip install pyinstaller
```

PyInstaller will analysis script to find imported modules and packages, once the script is obfuscated, nothing could be found, the final bundle complains of missing module.

Pyarmor provides option `--pack` to fix this problem, it supports the following values

- *onefile*: pack the obfuscated script to onefile
- *onedir*: pack the obfuscated script to onedir

- specfile: one `.spec` file used by PyInstaller to generate bundle

Once it's set, pyarmor will not only obfuscate the scripts, but also pack them to one bundle

Packing Scripts Automatically

Suppose our project tree like this:

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

Let's check what happens when the following commands are executed:

```
$ cd project
$ pyarmor gen --pack onefile foo.py
```

1. Pyarmor first call PyInstaller to analysis plain script *foo.py* to find all the imported moduels and packages
2. The imported module *util* and package *joker* are in the same path of *foo.py*, so Pyarmor will obfuscate *foo.py*, *util.py* and package *joker* by obfuscation options, and save them to path *.pyarmor/pack/dist*
3. For the other imported modules and packages, save them to hidden imports table
4. Finally pyarmor call PyInstaller again, pack all obfuscated scripts in *.pyarmor/pack/dist* and all the modules and packages in hidden imports table to one bundle.

Now let's run the final bundle, it's *dist/foo* or *dist/foo.exe*:

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

If need one folder bundle, just pass *onedir* to pack:

```
$ pyarmor gen --pack onedir foo.py
$ ls dist/foo
$ dist/foo/foo
```

Using specfile

In this project, there already has one `foo.spec` which could be used to pack plain script to onefile. For example:

```
$ pyinstaller foo.spec
$ dist/foo
```

In this case, pass it to *--pack* directly. For example:

```
$ pyarmor gen --pack foo.spec -r foo.py util.py joker/
```

What will Pyarmor do?

1. Pyarmor first obfuscates the scripts list in the command line, save them to *.pyarmor/pack/dist*
2. Next generates `foo.patched.spec` by `foo.spec`
3. Finally call PyInstaller to pack the bundle by `foo.patched.spec`

This patched specfile could replace plain scripts with obfuscated ones in *.pyarmor/pack/dist* when generating the bundle

---

Note: By this way, only listed scripts are obfuscated. If need obfuscate other used modules and packages, list all of them in command line.

Checking Obfuscated Scripts Have Been Packed

Add one line in the script `foo.py` or `joker/__init__.py`

```
print('this is __pyarmor__', __pyarmor__)
```

If it's not obfuscated, the final bundle will raise error. Because builtin name `__pyarmor__` is only available in the obfuscated scripts.

Using More PyInstaller Options

If need extra PyInstaller options, using configuration item `pack:pyi_options`. For example, reset it with one PyInstaller option `-w`:

```
$ pyarmor cfg pack:pyi_options = " -w"
```

Note that there need one leading whitespace in the `" -w"`, otherwise shell may complain of syntax error.

Let's append another option `-i`, it must be one whitespace between option `-i` and its value, do not use `=`. For example:

```
$ pyarmor cfg pack:pyi_options + " -i favion.ico"
```

Append another option:

```
$ pyarmor cfg pack:pyi_options + " --add-data joker/config.json:joker"
```

In Windows, maybe need use `;` as path separator instead of `:`:

```
C:/User/test> pyarmor cfg pack:pyi_options + "--add-data joker/config.json;joker"
```

All of them could be done by one command:

```
$ pyarmor cfg pack:pyi_options = " -w  -i favion.ico --add-data joker/config.json:joker"
```

See also:

pyarmor cfg

Using More Obfuscation Options

You can use any other obfuscation options to improve security. For example:

```
$ pyarmor gen --pack onefile --private foo.py
```

Anoter example, in Darwin, let obfuscated scripts work in both intel and Apple Silicon by extra option `--platform darwin.x86_64,darwin.arm64`:

```
$ pyarmor cfg pack:pyi_options = "--target-architecture universal2"
$ pyarmor gen --pack onefile --platform darwin.x86_64,darwin.arm64 foo.py
```

Note that some of them may not work. For example, *--restrict* can't be used with *--pack*.

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 to know how to using PyInstaller and using spec file, if not, learn it by yourself.

- First obfuscate the script by Pyarmor. List all the scripts and folders need to be obfuscated after main script, other obfuscation options could be used, but no *-i* or *--prefix*[1]:

  ```
  $ cd project/
  $ pyarmor gen -O obfdist -r foo.py util.py joker/
  ```

  Make sure the obfuscated script works:

  ```
  $ python obfdist/foo.py
  ```

- Then generate `foo.spec` by PyInstaller[2]:

  ```
  $ pyi-makespec --onefile foo.py
  ```

  Make sure it works and the final bundle works:

  ```
  $ pyinstaller foo.spec
  $ dist/foo
  ```

- Next patch `foo.spec` before line `pyz = PYZ`, this is major work

```
# Pyarmor patch start:

srcpath = ''
obfpath = 'obfdist'

def apply_pyarmor_patch(srcpath, obfpath):

    from PyInstaller.compat import is_win, is_cygwin
    extname = 'pyarmor_runtime' + ('.pyd' if is_win or is_cygwin else '.so')

    from glob import glob
```

(continues on next page)

[1] *-i* or *--prefix* results in runtime package could not be found
[2] Most of the other PyInstaller options could be used here

(continued from previous page)

```
    rtpkg = glob(os.path.join(obfpath, '*', extname))
    if len(rtpkg) != 1:
        raise RuntimeError('No runtime package found')
    rtpkg = os.path.basename(os.path.dirname(rtpkg[0]))

    extpath = os.path.join(rtpkg, extname)

    if hasattr(a.pure, '_code_cache'):
        code_cache = a.pure._code_cache
    else:
        from PyInstaller.config import CONF
        code_cache = CONF['code_cache'].get(id(a.pure))

    # Make sure both of them are absolute paths
    src = os.path.normcase(os.path.abspath(srcpath))
    obf = os.path.abspath(obfpath)
    n = len(src) + 1

    count = 0
    for i in range(len(a.scripts)):
        if os.path.normcase(a.scripts[i][1]).startswith(src):
            x = os.path.join(obf, a.scripts[i][1][n:])
            if os.path.exists(x):
                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 os.path.normcase(a.pure[i][1]).startswith(src):
            x = os.path.join(obf, a.pure[i][1][n:])
            if os.path.exists(x):
                code_cache.pop(a.pure[i][0], None)
                a.pure[i] = a.pure[i][0], x, a.pure[i][2]

    a.pure.append((rtpkg, os.path.join(obf, rtpkg, '__init__.py'), 'PYMODULE'))
    a.binaries.append((extpath, os.path.join(obf, extpath), 'EXTENSION'))

apply_pyarmor_patch(srcpath, obfpath)

# Pyarmor patch end.

# Before this line
# pyz = PYZ(...)
```

- Finally generate bundle by this patched `foo.spec`, use option `--clean` to to remove all cached files:

```
$ pyinstaller --clean foo.spec
```

If following this example, please

- Set `srcpath` to your path, in this example, it's current path
- Set `obfpath` to your real path, in this example, it's `obfdist`

how to verify obfuscated scripts have been packed

Insert debug code in the main script or imported modules and packages. For example

```
print('this is __pyarmor__', __pyarmor__)
```

If it's not obfuscated, the final bundle will raise error.

notes

Packing with PyInstaller Bundle

Deprecated since version 8.5.4: Use *--pack* `onefile` or `onedir` instead.

The option *--pack* also could accept an executable file generated by PyInstaller:

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

But only PyInstaller < 6.0 works by this method. If this option is set, pyarmor first obfuscates the scripts, then:

- 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

---

Important: Only listed scripts are obfuscated, if need obfuscate more scripts and sub packages, list all of them in command line. For example:

```
$ pyarmor gen --pack dist/foo/foo -r *.py dir1 dir2 ...
```

Segment fault in Apple M1

In Apple M1 if the final executable segment fault, please check codesign of runtime package:

```
$ codesign -v dist/foo/pyarmor_runtime_000000/pyarmor_runtime.so
```

And re-sign it if the code sign is invalid:

```
$ codesign -f -s dist/foo/pyarmor_runtime_000000/pyarmor_runtime.so
```

If you use *--enable-bcc* or *--enable-jit* to obfuscate the scripts, you need enable Allow Execution of JIT-compiled Code Entitlement

If your app doesn't have the new signature format, or is missing the DER entitlements in the signature, you'll need to re-sign the app on a Mac running macOS 11 or later, which includes the DER encoding by default.

If you're unable to use macOS 11 or later to re-sign your app, you can re-sign it from the command-line in macOS 10.14 and later. To do so, use the following command to re-sign the MyApp.app app bundle with DER entitlements by using a signing identity named "Your Codesign Identity" stored in the keychain:

```
$ codesign -s "Your Codesign Identity" -f --preserve-metadata --generate-entitlement-der␣
↪/path/to/MyApp.app
```

See also:

Using the latest code signature format

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 work.

This chapter describes how RFT mode work, it's helpful to solve RFT mode issues of complex package and scripts.

What does RFT mode change?

- function
- class
- method
- global variable
- local variable
- builtin name
- import name

What does RFT mode not change?

- 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.

  Note that in rft-auto-include mode, local variables will not be touched, but they're renamed in next obfuscation process, unless you explicitly disable it by *pyarmor cfg mix_localnames=0*.

  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.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.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.

Manual ruler

This is only for **rft-auto-include**:

```
$ pyarmor cfg rft_auto_exclude=0
```

The rule is used to transform name in chain attributes

One line one rule, the rule format:

```
patterns actions

patterns = pattern1.pattern2.pattern3...
actions = X.X.X...
```

Each pattern is same as pattern in `fnmatch`, each action `X` is either char `?` or any word. `?` means transform the corresponding attribute automatically, any other word means not transform this word.

For example, a ruler:

```
self.task.x self.task.?
```

apply to this script

```
class Sdipmk:

    def __init__(self):
        self.width = 100
        self.height = 200

    def move(self, x, y, absolute=False):
```

(continues on next page)

(continued from previous page)

```
8         self.task.x = int(abs(x*65536/self.width)) if absolute else int(x)
9         self.task.y = int(abs(y*65536/self.height)) if absolute else int(y)
10        return Mouse(MS_MOVE, x, y)
```

First configure this ruler by command:

```
$ pyarmor cfg rft_rulers "self.task.x self.task.?"
```

Then check the result:

```
$ pyarmor gen --enable-rft foo.py
$ grep trace.rft pyarmor.trace.log

trace.rft            foo:8 (self.task.x->self.task.pyarmor__2)
```

line 8 `self.task.x` will be transformed to `self.task.pyarmor__2`

Let's change action to `self.?.?`, and check the result:

```
$ pyarmor cfg rft_rulers "self.task.x self.?.?"
$ grep trace.rft pyarmor.trace.log

trace.rft            foo:8 (self.task.x->self.pyarmor__1.pyarmor__2)
```

Do not change action to `?.?.?`, it doesn't work, the first action can't be `?`

Let's add new ruler to change `self.task.y`, here need to use `^` to append new line to rulers:

```
$ pyarmor cfg rft_rulers ^"self.task.y self.?.?"
$ grep trace.rft pyarmor.trace.log

trace.rft            foo:8 (self.task.x->self.pyarmor__1.pyarmor__2)
trace.rft            foo:9 (self.task.y->self.pyarmor__1.pyarmor__3)
```

Actually, both of rulers can combined to one:

```
$ pyarmor cfg rft_rulers = "self.task.* self.?.?"
$ grep trace.rft pyarmor.trace.log

trace.rft            foo:8 (self.task.x->self.pyarmor__1.pyarmor__2)
trace.rft            foo:9 (self.task.y->self.pyarmor__1.pyarmor__3)
```

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.trace.log
$ grep trace.bcc 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.

If there is `!` after `trace.bcc`, it means this function is ignored by BCC mode. For example:

```
trace.bcc ! foo:29:Test.new (unsupported function "super")
```

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 functions or class methods in one module:

```
$ pyarmor cfg -p pkgname.modname bcc:excludes="name"
$ pyarmor cfg -p pkgname.modname bcc:excludes="name1 name2 name3"

$ pyarmor cfg -p pkgname.modname bcc:excludes="Class.method_1"
$ pyarmor cfg -p pkgname.modname bcc:excludes="Class.*"
```

If no option `-p`, same name function in the other scripts will be ignored too.

Here it's an example script `foo.py`

```
def hello_a():
    pass

def hello_b():
    pass

class Test(object):

    def __init__(self):
        pass

    def hello_a():
        pass
```

Exclude functions by one of forms:

```
$ pyarmor cfg -p foo bcc:excludes = "hello_a"
$ pyarmor cfg -p foo bcc:excludes = "hello_a hello_b"
$ pyarmor cfg -p foo bcc:excludes = "hello_*"

$ pyarmor cfg -p foo bcc:excludes = "Test.hello_a"
$ pyarmor cfg -p foo bcc:excludes = "Test.*"
$ pyarmor cfg -p foo bcc:excludes = "Test.__*__"

$ pyarmor cfg -p foo bcc:excludes = "hello_a Test.hello_a"
```

If want to BCC mode handle specified functions, use option *bcc:includes*:

```
# clear excludes
$ pyarmor cfg bcc:excludes = ""

# BCC mode only handles module function "hello_a"
$ pyarmor cfg -p foo bcc:includes = "hello_a"

# Need extra settings let BCC mode handle class method "Test.hello_a"
$ pyarmor cfg -p foo bcc:includes + "Test.hello_a"

# BCC mode handles all methods of class "Test" except method "__init__"
$ pyarmor cfg -p foo bcc:includes="Test.*" bcc:excludes="Test.__init__"
```

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.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
```

Both *bcc:includes* and *bcc:excludes* only work on top function and class method, they can't be used to filter nest function and methods of nest class.

For example,

```
def hello():

    def wrap():
        pass

    class Test:

        def __init__(self):
            pass
```

The nest function `wrap` and nest class `Test` can't be ignored by the following commands:

```
pyarmor cfg bcc:excludes = "wrap hello.wrap Test.__init__ hello.Test.__init__"
```

The only solution is to ignore top function `hello`:

```
pyarmor cfg bcc:excludes = hello
```

New in version 8.3.4: The option *bcc:includes*.

Changed in version 8.3.4: The option *bcc:excludes*, in previous version:

```
# Exclude module function "hello_a" and any method "hello_a"
pyarmor cfg bcc:excludes="hello_a"

# It doesn't work if there is class name in filter
pyarmor cfg bcc:excludes="Myclass.hello_a"
```

Now:

```
# Exclude module function "hello_a" and method "hello_a" in any class
pyarmor cfg bcc:excludes="hello_a *.hello_a"

# It works to ignore one method "Myclass.hello_a"
pyarmor cfg bcc:excludes="Myclass.hello_a"
```

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. For example, there is no *__qualname__* for BCC function.
- In the exception handler, *sys.exception()* will return *None*, so the functions depended on *sys.exception* may not work. For example

```
import traceback

def main():
    try:
        1 / 0
    except Exception as e:

        # In BCC mode, sys.exception() will return None, the output is:
        #    None
        print(sys.exception())

        # It doesn't work in BCC mode, the output is "NoneType: None"
        traceback.print_exc()

        # The traceback will be printed by this form in BCC mode
        # But the traceback doesn't include line no. and source
        traceback.print_exception(e)

main()
```

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,
    ast.MatchMapping, ast.MatchClass, ast.MatchStar,
    ast.MatchAs, ast.MatchOr
)
```

And unsupported functions:

- exec
- eval
- super
- locals
- sys._getframe

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)
```

Known issues

- When format string has syntax error, BCC mode may raise *SystemError: NULL object passed to Py_BuildValue*, instead of *SyntaxError* or *ValueError*.

  Found in test cases *lib/python3.12/test/test_fstring.py*:

  - test_invalid_syntax_error_message
  - test_missing_variable
  - test_syntax_error_for_starred_expressions
  - test_with_a_commas_and_an_underscore_in_format_specifier
  - test_with_an_underscore_and_a_comma_in_format_specifier
  - test_with_two_commas_in_format_specifier
  - test_with_two_underscore_in_format_specifier

- When generating BCC code, pyarmor may raise *RuntimeError: link bcc code failed*

  Try to add extra cflags *-DENABLE_BCC_MEMSET* for this platform. For example, use the following command for Windows X86:

  ```
  pyarmor cfg windows.x86.bcc:cflags += " -DENABLE_BCC_MEMSET"
  ```

  Or patch `pyarmor/cli/default.cfg` directly, the final value should be like this:

  ```
  Section: windows.x86.bcc

  cflags = --target=i686-elf-linux -O3 -Wno-unsequenced -fno-asynchronous-unwind-
  ↪tables -fno-unwind-tables -fno-stack-protector -fPIC -mno-sse -std=c99 -c -
  ↪DENABLE_BCC_MEMSET
  ```

3.4.6 Security and Performance

About Security

Pyarmor focuses on protecting Python scripts, through several irreversible obfuscation methods, Pyarmor makes sure the obfuscated scripts can't be restored in any way.

Pyarmor provides rich options to obfuscate scripts to balance security and performance. If anyone announces they have broken pyarmor, please try a simple script with different security options, refer to *Highest security and performance*. If any irreversible obfuscation has been 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, it may reduce performance. In most cases, we need to 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 the same test script in the same machine twice, not speak of different test scripts in different machines. So the test data in these tables are only guidelines, 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
```

(continues on next page)

(continued from previous page)

```
    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, and 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
```

(continues on next page)

(continued from previous page)

```
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 10: 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 11: 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 12: 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 the same.

BCC Mode Performance

BCC mode converts some code to C function, it needs extra time to load binary code, but the 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
...

$ python dist/testben.py
...
```

Table 13: 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 14: 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

This documentation is only apply to Pyarmor 8.0 plus.

Pyarmor is published in PyPI, free trial version never expires. Try it by the following commands:

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

There are some limitations in free version, for example, can't obfuscate big scripts etc. These limitations can be unlocked by different license types. Pyarmor has 4 kind of licenses:

- Basic
- Pro: Irreversible Obfuscation
- Group: Offline build
- CI: for CI/CD pipeline, new in v9.0

Table 15: Table-1. Compare Different Licenses

Feature	Free	Basic	Pro	Group	CI
Big Script / Mix String[1]		Y	Y	Y	Y
BCC / RFT / FLY mode[2]			Y	Y	Y
Offline build[3]				Y	
Maximum build devices[4]		100	100	200	0
Unlimited local dockers[5]				Y	
Work in CI/CD pipeline[6]	Y	Y			Y

Notes

Important: CI license doesn't work in the runner which has its own disk.

If the runner is not docker container, use Pro license instead.

Important: All Pyarmor Licenses are only used to generate the obfuscated scripts in build device

It need neither install Pyarmor nor verify Pyarmor License to execute obfuscated scripts

The obfuscated scripts are generated by Pyarmor, but they are completely independent of Pyarmor

You can take them as normal Python scripts, so what the obfuscated scripts do is controlled by you, not by Pyarmor

3.5.1 Terms of Use

1. Only use Pyarmor on your own scripts

   In any case, even you have purchased Pyarmor License, it's not allowed to obfuscate other scripts of which you haven't property. For example, call pyarmor in your app to obfuscate your customer's scripts, provide obfuscation services based on Pyarmor by website etc.

2. No profit no license required

   Free version can be used to obfuscate your scripts which COULD NOT make lot of money for you. Otherwise it need purchase Pyarmor License.

3. One product one license

   One product means one kind of product, not one copy of product. For example, Micorsoft Excel is one product, even it's installed on countless devices

[1]
- Big Script: file size exceeds a certain value.
- Mix Str: obfuscating string constant in script

[2]
- RFT Mode: renaming function/class/method/variable in Python scripts
- BCC Mode: Transforming some Python functions in scripts to c functions, compile them to machine instructions directly

[3] Offline build: the build device need not be online to verify Pyarmor License
[4] Maximum devices could install Pyarmor, each docker run will be taken as one new build device, CI license only works in CI/CD pipeline
[5] Unlimited local dockers: run docker container in local machine, may be offline or in private network
[6] Work in CI/CD pipeline: it need special option to register Pyarmor in CI/CD pipeline.

Each license has an unique number, the format is `pyarmor-vax-xxxxxx`, which x stands for a digital.

4. Pay once

   Except CI License, all the other licenses work forever with Pyarmor version when this license is purchased, but **may not work in future Pyarmor versions**

5. Fair use

   If you have purchased Pyarmor License for one product, but you have another product, and the total revenue of the other project is less than 100 x Pyarmor License fee, you could rent this license in your aother product

   Pyarmor CI License has rate limit in CI/CD pipeline

In details check Pyarmor EULA

See also:

Using Pyarmor Licenses

3.5.2 Privacy

License No. and product name will be embedded into obfuscated scripts, all the other user's information, for example, regname, email are not

For Pyarmor Basic and Pro License, only Pyarmor License file, serial number of hard disk, Ethernet address, IPv4/IPv6 address, and hostname will be sent to Pyarmor License Server for verification

When using Basic or CI License in CI/CD pipeline, some information about docker like docker name, ethernet address, IPv4/IPv6 address, and license information will be sent to Pyarmor License Server for verification

No any user script will be uploaded to Pyarmor License Server

3.5.3 Technical Support

License fees only for unlock features, not include technical supports

Users need to learn Pyarmor features and how to use it by themself. Generally Pyarmor Team won't help to debug users' case and teach them how to use Pyarmor

Pyarmor provides comprehensive learning systems, including but not limited to the following ways:

- *Online documentation*
- *Checklist* and *FAQs* could fix 90% issues reported to Pyarmor Team
- Discussions in Pyarmor project home
- Full examples to show each option usage and common cases by command `pyarmor man`
- Learn Pyarmor concepts by figure and animations in EKE Learning Platform (coming soon)

Rome was not bulit in a day. Pyarmor Team keeps improving documentation and learning systems according to users feedback to make it easy and effects

Report bugs and request new features in Pyarmor project home, Email to pyarmor@163.com is only for security and private issues, there may no reply for common technical issues.

Pyarmor team generally will handle submitted issues within 24 hours (working time), but may be extended during holidays or special circumstances

Pyarmor Team doesn't provide any instant technical support by telphone or other similar tools.

3.5.4 Purchasing license

If you have Pyarmor 8.6+ installed, this command also could open shopping cart:

```
$ pyarmor reg --buy
```

Open Pyarmor Shopping Cart in any web browser:

> https://jondy.github.io/paypal/index.html

All of these license are only for Pyarmor 8.0+ with Python 3.7+, if need work with Pyarmor 7.x which supports Python < 3.7, please purchase *Pyarmor Old License*:

> https://jondy.github.io/paypal/obsolete.html

3.5.5 Refund policy

If activation file isn't used, and purchasing date is in 30 days, refund is acceptable.

- If purchasing order from MyCommerce:
  1. Email to Ordersupport@mycommerce.com with order information and ask for refund.
  2. Or click FindMyOrder page to submit refund request
- If purchasing order from reseller, contact your reseller
- For other cases, email to pyarmor@163.com

Out of 30 days, or activation file has been used, refund request will be rejected.

3.5.6 Appendix

What is one product

First of all, if not for sale, all the Python scripts are belong to one product "non-profits".

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 is one product, it includes:

- 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.

Which license type is right for me

All of the following licenses are only for Python 3.7+

Table 16: Table-2. Select Different Licenses

Condition	Free	Basic	Pro	Group	CI
Less than 100 runs per month in CI/CD pipeline	Y	Y	Y		Y
More than 100 runs per month in CI/CD pipeline	Y	Y			Y
Need offline obfuscation	Y			Y	
Need irreversible obfuscation			Y	Y	Y
Less than 100 runs per month in local dockers	Y	Y	Y	Y	Y
More than 100 runs per month in local dockers	Y	Y		Y	Y

How many licenses are required

1. List all the products which are sold separately
   - If the sales revenue of the product is less than 100 x Pyarmor license fee, there is no need to list the product
   - If no more than 2 products left, one license is OK
2. How to tell diffenent product could be taken as one proudct in Pyarmor view

   Suppose there are 2 products in step 1: X and Y

   a. case 1, X and Y could use one license
      - Y includes whole X features
      - The extra features of Y is related to X features

      For example, Pyarmor Basic (X), Pyarmor Pro (Y) could use one license because
      - Pyarmor Pro includes all the features of Pyarmor Basic
      - The extra features of Pyarmor Pro is irreversible obfuscation which is an enhancement of Pyarmor Basic

   b. case 2, X and Y need 2 licenses
      - Y includes whole X features
      - But X features is very small in the Y features

      For example, X is a facial recognition product, B is an attendance management system that uses facial recognition functionality

   c. case 3, X and Y could use one license
      - Y is a functional supplement for product X

      For example, X is CAD Editor, Y is CAD Tool which is used to convert CAD file to PDF file

d. case 4, X and Y need 2 licenses

- The functions of X and Y are almost independent

For example, Microsoft Word (X), Micorsoft Excel (Y) are 2 products, even they're belong to Micorsoft Office Suite

3. Regard to shared backend system

There are 2 product A and B, share one backend engineer C

a. case 1: need obfuscate backend C, but frontend of A and B need not

- one license for backend C is OK

b. case 2: need obfuscate backend C, and the frontend of A and B

- 2 licenses, one for A, another for B
- C need not new license, use any license for A or B

4. Many products need use one license in technical

If there are many products (in Pyarmor view) need use one Pyarmor license in development view, it should

- Purchase many licenses
- Register each product with different license once
- Use one of registeration file to generate the obfuscated scripts for all the products

How to upgrade Pyarmor Licenses

Now it doesn't support to upgrade one Pyarmor License to any other Pyarmor License. For example, upgrade Pyarmor Basic to Pyarmor Pro by paying price difference

There are only the following special cases

- Pyarmor Old License used by Pyarmor <= 7.x could be upgrade to Pyarmor Basic License in some conditions
- If upgrading Pyarmor v8 to v9, Pyarmor License need to be re-activated

Please refer to What need to do after upgrading Pyarmor

In short, if it could be upgraded successfully by the guide in the documentation, it's OK. If something is wrong, this license can't be upgraded. Pyarmor Team doesn't handle this kind of request, there may be no any reply.

Q & A

I have just started selling my product, do I can use Pyarmor trial version to product it

Before the value of sales exceed 100 x Pyarmor license fee, Pyarmor trial version can be used to obfuscate the product. After the value of sales exceed, it need purchase Pyarmor license.

I noticed that the pricing table lists "0 Maximum build devices," for the CI/CD plan which has left me a bit confused. Could you please clarify what this means?

It means CI License only works in CI/CD pipeline, can't be used in local device.

If we are using the product in our CI/CD environment we can deploy it to unlimited number of kubernetes pods

If the product means the obfuscated scripts, Pyarmor has no any limitation on it.

Pyarmor License is only apply to build machine in which to generate the obfuscated scripts.

3.6 FAQ

Pyarmor provides rich options for different cases, the default option only works for common case. When something is wrong, it may be not bug, but need the right options. Users need spend time learning Pyarmor by documentation or *pyarmor man*, and find the right options for their project. Generally pyarmor team won't learn user's project and tell user which options should be used.

Pyarmor is well document, you needn't read all of them at first, but it's necessary to read *Getting Started* which includes essentials concepts. If you spend 10 minutes reading full of it, it may save your several hours to solve the wrong usage problems.

If using **`pyarmor-7`** or Pyarmor < 8.0, please check Pyarmor 7.x Doc

---

Important: Pyarmor team handles too many wrong usage issues, so one document *Pyarmor Check List* has been arranged to solve this kind of issue quickly. If you aren't sure this issue is wrong usage or not, please check this doc or https://eke.dashingsoft.com/pyarmor/ask/ at first.

Pyarmor team will mark this kind of issue as *invalid* or *documented* and close it immediately.

3.6.1 Asking questions in GitHub

Before ask question, please try the following common solutions in order to avoid duplicated issues:

- If need some feature, first check *the detailed table of contents*
- If there is error message, check *Error Messages*
- If obfuscated scripts don't work as expected, make sure you have read *Understanding Obfuscated Script*
- 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

3.6.2 Reporting bug

A good report should have

- A clear title
- Reproduced steps
- Actual results
- Expected results

It's recommand to report issue by command *pyarmor man*.

Important: If a bug report misses necessary information and not clear, it may be marked as invalid and closed immediately.

Build issues

If there is error message when run pyarmor, please first check *Error Messages* to find solutions

If still no solution, please report issue based on `pyarmor.bug.log` generated by Pyarmor automically. For example:

```
[BUG]: no found input "fooxxx.py"

## Command Line
pyarmor gen fooxxx.py

## Environments
Home /Users/jondy/.pyarmor
Platform darwin.x86_64 (darwin.x86_64)
Python 3.12.0
Pyarmor 9.0.4 (group), 006000, btarmor
```

Pack issues

Check list for pack issues:

- Using PyInstaller to pack the script without obfuscation, make sure the final bundle works
- Without packing, only obfuscate scripts, make sure the obfuscated scripts works

If check list pass, then report bug as the next section guide

Runtime issues

If the obfuscated scripts doesn't work or not as expected, first check *Understanding Obfuscated Script*

If there is error message, also check *Error Messages* to find solutions

If using `--enable-bcc`, try to obfuscate script without it. If make sure this option results in problem, check *Using bccmode pro* to find solutions

If using `--enable-rft`, , try to obfuscate script without it. If make sure this option results in problem, check *Using rftmode pro* to find solutions

Try to use less options to obfuscate script, find the minimum options to reproduce this issue

Report runtime issue with:

- A clear title
- Full command options to obfuscate scripts
- Full command to run the obfuscated scripts and traceback (if any)
- Necessary supplements and explanations

3.6.3 Hot Questions

Is there any tool could broken Pyarmor?

Pyarmor team doesn't care about these kind of tools, but focus on researching CPython source to design obfuscation algorithm. Through several irreversible obfuscation methods, Pyarmor makes sure the obfuscated scripts can't be restored by any way.

Refer to *Highest security and performance*, use the highest security options available for you to obfuscate this script

```
import sys

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

print('python version:', sys.version_info[:2])
print('this is fib(10)', fib(10))
```

And then try any tool to broken it.

If it's broken, please send Python version, Pyarmor version, Platform, Obfuscation options, sample script and broken steps to pyarmor@163.com

Do not publish any pyarmor hack link in Pyarmor project

Pyarmor is good at protecting Python scripts, but not 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. More information check *Protecting Runtime Memory Data*

3.6.4 License

Will Pyarmor Pro license upload my scripts to remote server to verify license?

No. For Pyarmor Basic and Pro License, only Pyarmor License file, serial number of hard disk, Ethernet address, IPv4/IPv6 address, and hostname will be sent to remote server for verification.

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?

Please refer to *Using Pyarmor in CI Pipeline*

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"

Which PyArmor 8.0 license for CI, more than 100 runs / day

It's recommend to upgrade to Pyarmor 9, and use *Pyarmor CI* License or *Pyarmor Basic* License. See also *Using Pyarmor in CI Pipeline*

We should be able to assume that the CI regfile will keep working as long as it is within the license limits. Otherwise, builds might break at a moment notice. Could you confirm whether it is safe to assume that the CI license will keep working?

Fix Pyarmor version in the CI/CD pipeline, CI regfile works within the validity period.

About CI License, is there an option to allow offline usage in the CI/CD pipeline?

No. There is no offline option for CI License.

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 *Using Pyarmor License* 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 2 times because the core libraries are rewritten:

- 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

Does the license include access to support and software updates? If so, what is the duration of support and how are updates delivered?

Generally the license could be used in the next versions, until there are big changes in one major version, but I have no plan in details. Just run *pip install* to upgrade Pyarmor to latest version, the license will keep work.

But there is no more technical supports about how to use Pyarmor, Pyarmor is a command line tool, and all the options are full documentations. Pyarmor users need spend some time to learn Pyarmor by himself.

Is there an option for custom licensing arrangements to accommodate specific project or organizational needs?

At this time, the answer is no.

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

Using PyArmor 9.x newer version, we cannot generated licenses compatible with "Software" created using older PyArmor (7.x versions), mainly using license.lic. Correct?

You're right. At this time one possible solution is still using Pyarmor 7 obfuscated script to verify old runtime key, in Pyarmor 9 obfuscated script check old runtime key by calling Pyarmor 7 obfuscated script indirectly (IPC)

3.6.5 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

- If purchasing order from MyCommerce:
  1. Email to Ordersupport@mycommerce.com with order information and ask for refund.
  2. Or click FindMyOrder page to submit refund request
- If purchasing order from reseller, contact your reseller
- For other cases, email to pyarmor@163.com

3.6.6 Misc.

What is the ECCN or rating of Pyarmor (EAR99,5D99S,5D002 or other type ECCN)?

EAR99

Does Pyarmor contain any encryption capabilities?

Pyarmor uses AES/RSA etc., but it hasn't its own encryption algorithms.

What is the country of origin of this package?

China

Where is the final built for Pyarmor?

All of Pyarmor packages are published in the PyPI, refer to *Pyarmor Package* and section *Installation in offline device* in the chapter *Installation*

Pyarmor Pro checks the internet, what is the output IP or DNS? I need to release it on my client's firewall, placing an outbound rule for yours IP.

Now it's *119.23.58.77* (March 20, 2024)

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[cygcheck] How to get *cygcheck.exe*

Download https://pyarmor.dashingsoft.com/downloads/tools/cygcheck.zip and unzip it

Or get it from official website

- Open https://cygwin.com/mirrors.html
- Select one mirror, enter paths *x86_64/release/cygwin/*
- Download the latest package *cygwin-3.5.3-1.tar.xz*
- Extract *cygcheck.exe* from this file by *tar xJf cygwin-3.5.3-1.tar.xz*

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

E

G

H

J

L

M

O

P

R

T