PyArmor Documentation

Release 4.5.0

Jondy Zhao

Mar 26, 2020

Contents

Version PyArmor 4.5

Homepage http://pyarmor.dashingsoft.com/

Contact jondy.zhao@gmail.com

Authors Jondy Zhao

PyArmor is a command line tool used to obfuscate python scripts, bind obfuscated scripts to fixed machine or expire obfuscated scripts. It protects Python scripts by the following ways:

- Obfuscate code object to protect constants and literal strings.
- Obfuscate co_code of each function (code object) in runtime.
- Clear f_locals of frame as soon as code object completed execution.
- Verify the license file of obfuscated scripts while running it.

PyArmor supports Python 2.6, 2.7 and Python 3.

PyArmor is tested against `Windows`, `Mac OS X`, and `Linux`.

PyArmor has been used successfully with `FreeBSD` and embedded platform such as `Raspberry Pi`, `Banana Pi`, `Orange Pi`, `TS-4600 / TS-7600` etc. but is not fullly tested against them.

Contents:

CHAPTER 1

Installation

PyArmor is a normal Python package. You can download the archive from PyPi, but it is easier to install using pip where is is available, for example:

```
pip install pyarmor
```

or upgrade to a newer version:

```
pip install --upgrade pyarmor
```

1.1 Verifying the installation

On all platforms, the command `pyarmor` should now exist on the execution path. To verify this, enter the command:

```
pyarmor --version
```

The result should show `PyArmor Version X.Y.Z` or `PyArmor Trial Version X.Y.Z`.

If the command is not found, make sure the execution path includes the proper directory.

1.2 Installed commands

The complete installation places these commands on the execution path:

- `pyarmor` is the main command. See *Using PyArmor*.
- `pyarmor-webui` is used to open a simple web ui of PyArmor.

If you do not perform a complete installation (installing via `pip`), these commands will not be installed as commands. However, you can still execute all the functions documented below by running Python scripts found in the distribution folder. The equivalent of the `pyarmor` command is *`pyarmor-folder`*`/pyarmor.py`, and of `pyarmor-webui` is *`pyarmor-folder`*`/pyarmor-webui.py`.

CHAPTER 2

Using PyArmor

The syntax of the `pyarmor` command is:

> `pyarmor` [*command*] [*options*]

2.1 Obfuscating Python Scripts

Use command `obfuscate` to obfuscate python scripts. In the most simple case, set the current directory to the location of your program `myscript.py` and execute:

```
pyarmor obfuscate myscript.py
```

PyArmor obfuscates `myscript.py` and all the `*.py` in the same folder:

- Create `.pyarmor_capsule.zip` in the `HOME` folder if it doesn't exists.
- Creates a folder `dist` in the same folder as the script if it does not exist.
- Writes the obfuscated `myscript.py` in the `dist` folder.
- Writes all the obfuscated `*.py` in the same folder as the script in the `dist` folder.
- Copy runtime files used to run obfuscated scripts to the `dist` folder.

In the `dist` folder you find the obfuscated scripts you distribute to your users:

```
myscript.py

pytransform.py
_pytransform.so, or _pytransform.dll in Windows, _pytransform.dylib in MacOS
pyshield.key
pyshield.lic
product.key
license.lic
```

The last 6 files called `Runtime Files`, all of them are required to run the obfuscated script.

Normally you name one script on the command line. It's entry script. The content of `myscript.py` would be like this:

```
from pytransfrom import pyarmor_runtime
pyarmor_runtime()

__pyarmor__(__name__, __file__, b'\x06\x0f...')
```

The first 2 lines called `Bootstrap Code`, are only in the entry script. They must be run before using any obfuscated file. For all the other obfuscated `*.py`, there is only last line:

```
__pyarmor__(__name__, __file__, b'\x0a\x02...')
```

Run the obfuscated script:

```
cd dist
python myscript.py
```

By default, only the `*.py` in the same path as the entry script are obfuscated. To obfuscate all the `*.py` in the sub-folder recursively, execute this command:

```
pyarmor obfuscate --recursive myscript.py
```

2.2 Distributing Obfuscated Scripts

Except the obfuscated scripts, all the *Runtime Files* need to be distributed to the end user.

Note that the dynamic library *_pytransfrom* is plain, in order to protect it, refer to Protect Dynamic Library _pytransform

2.3 Generating License For Obfuscated Scripts

Use command `licenses` to generate new `license.lic` for obfuscated scripts.

By default there is `dist/license.lic` generated by command `obfuscate`. It allows obfuscated scripts run in any machine and never expired.

Generate an expired license for obfuscated script:

```
pyarmor licenses --expired 2019-01-01 code-001
```

PyArmor generates new license file:

- Read data from `.pyarmor_capsule.zip` in the `HOME` folder
- Create `license.lic` in the `licenses/code-001` folder
- Create `license.lic.txt` in the `licenses/code-001` folder

Overwrite default license with new one:

```
cp licenses/code-001/license.lic dist/
```

Run obfuscated script with new license, It will report error after Jan. 1, 2019:

```
cd dist
python myscript.py
```

Generate license to bind obfuscated scripts to fixed machine, first get hardware information:

```
pyarmor hdinfo
```

Then generate new license bind to harddisk serial number and mac address:

```
pyarmor licenses --bind-disk '100304PBN2081SF3NJ5T' --bind-mac '20:c1:d2:2f:a0:96' code-002
```

Run obfuscated script with new license:

```
cp licenses/code-002/license.lic dist/

cd dist/
python myscript.py
```

2.4 Extending License Type

It's easy to extend any other licese type for obfuscated scripts: just add authentication code in the entry script. The script can't be changed any more after it is obfuscated, so write what ever you want by Python. For example, check expired date by NTP server other than local time:

```
import ntplib
from time import mktime, strptime
c = ntplib.NTPClient()
response = c.request('europe.pool.ntp.org', version=3)
if response.tx_time > mktime(strptime('20190202', '%Y%m%d')):
    sys.exit(1)
```

2.5 Packing Obfuscated Scripts

Use command `pack` to pack obfuscated scripts into the bundle.

First install *PyInstaller*:

```
pip install pyinstaller
```

Set the current directory to the location of your program `myscript.py` and execute:

```
pyarmor pack myscript.py
```

PyArmor packs `myscript.py`:

- Execute `pyarmor obfuscate` to obfuscate `myscript.py`
- Execute `pyinstaller myscipt.py` to create `myscript.spec`
- Update `myscript.spec`, replace original scripts with obfuscated ones
- Execute `pyinstaller myscript.spec` to bundle the obfuscated scripts

In the `dist/myscript` folder you find the bundled app you distribute to your users.

Run the final executeable file:

```
dist/myscript/myscript
```

Check the scripts have been obfuscated. It should return error:

```
rm dist/myscript/license.lic
dist/myscript/myscript
```

Generate an expired license for the bundle:

```
pyarmor licenses --expired 2019-01-01 code-003
cp licenses/code-003/license.lic dist/myscript

dist/myscript/myscript
```

Note that command `pack` maybe doesn't work if `.spec` file of PyInstaller has been customed. You need edit `.spec` file to pack obfuscated scripts, See *How To Pack Obfuscated Scripts*.

CHAPTER 3

Runtime Module *pytransform*

If you have realized that the obfuscated scripts are black box for end users, you can do more in your own Python scripts.In these cases, `pytransform` would be useful.

The `pytransform` module is distributed with obfuscated scripts, and must be imported before running any obfuscated scripts. It also can be used in your python scripts.

3.1 Contents

exception PytransformError

This is raised when any pytransform api failed. The argument to the exception is a string indicating the cause of the error.

get_expired_days()

Return how many days left for time limitation license.

0: has been expired

-1: never expired

get_license_info()

Get license information of obfuscated scripts.

It returns a dict with keys *expired*, *CODE*, *IFMAC*.

The value of *expired* is == -1 means no time limitation.

Raise `PytransformError` if license is invalid, for example, it has been expired.

get_hd_info(*hdtype*, *size=256*)

Get hardware information by *hdtype*, *hdtype* could one of

HT_HARDDISK return the serial number of first harddisk

HT_IFMAC return mac address of first network card

Raise `PytransformError` if something is wrong.

HT_HARDDISK, HT_IFMAC
Constant for *hdtype* when calling `get_hd_info()`

3.2 Examples

Show left days of license

```
from pytransform import PytransformError, get_license_info, get_expired_days
try:
    code = get_license_info()['CODE']
    left_days = get_expired_days()
    if left_days == -1:
        print('This license for %s is never expired' % code)
    else:
        print('This license %s will be expired in %d days' % (code, left_days))
except PytransformError as e:
    print(e)
```

Double check harddisk information

```
from pytransform import get_hd_info, HT_IFMAC
expected_mac_address = 'xx:xx:xx:xx:xx'
if get_hd_info(HT_IFMAC) != expected_mac_address:
    sys.exit(1)
```

Check internet time by NTP server, expired on *2019-2-2*

```
from ntplib import NTPClient
from time import mktime, strptime

NTP_SERVER = 'europe.pool.ntp.org'
EXPIRED_DATE = '20190202'

c = NTPClient()
response = c.request(NTP_SERVER, version=3)
if response.tx_time > mktime(strptime(EXPIRED_DATE, '%Y%m%d')):
    sys.exit(1)
```

CHAPTER 4

The Security of PyArmor

PyArmor will obfuscate python module in two levels. First obfucate each function in module, then obfuscate the whole module file. For example, there is a file *foo.py*:

```
def hello():
    print('Hello world!')

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

if __name == '__main__':
    hello()
    print('1 + 1 = %d' % sum(1, 1))
```

PyArmor first obfuscates the function *hello* and *sum*, then obfuscates the whole moudle *foo*. In the runtime, only current called function is restored and it will be obfuscated as soon as code object completed execution. So even trace code in any `c` debugger, only a piece of code object could be got one time.

4.1 Cross Protection for *_pytransform*

The core functions of *PyArmor* are written by *c* in the dynamic library *_pytransform*. *_pytransform* protects itself by JIT technical, and the obfuscated scripts is protected by *_pytransform*. on the other hand, the dynamic library *_pytransform* is checked in the obfuscated script to be sure it's not changed. This is called Cross Protection.

The dynamic library *_pytransform.so* uses JIT technical to achieve two tasks:

- Keep the des key used to encrypt python scripts from tracing by any c debugger
- The code segment can't be changed any more. For example, change instruction *JZ* to *JNZ*, so that *_pytransform.so* can execute even if checking license failed

How JIT works?

First the instruction set based on GNU lightning are defined

Then write some core functions by this instruction set in c file, maybe like this:

```
t_instruction protect_set_key_iv = {
    // function 1
    0x80001,
    0x50020,
    ...

    // function 2
    0x80001,
    0xA0F80,
    ...
}

t_instruction protect_decrypt_buffer = {
    // function 1
    0x80021,
    0x52029,
    ...

    // function 2
    0x80001,
    0xC0901,
    ...
}
```

Build *_pytransform.so*, calculate the codesum of code segment of *_pytransform.so*

Replace some instruction with real codesum got before, and obfuscate all the instructions except "function 1" in c file

Finally build *_pytransform.so* with this changed c file.

When running obfuscated script, *_pytransform.so* loaded. Once a proected function is called, it will

1. Generate code from *function 1*
2. **Run *function 1*:**
    - check codesum of code segment, if not expected, quit
    - check tickcount, if too long, quit
    - check there is any debugger, if found, quit
    - clear hardware breakpoints if possible
    - restore next function *function 2*
3. Generate code from *function 2*
4. Run *function 2*, do same thing as *function 1*

After repeat some times, the real code is called.

In order to protect *_pytransform* in Python script, some extra code will be inserted into the entry script before the line *if __name__ == '__main__'* when obfuscating scripts:

```
def protect_pytransform():

    import pytransform

    def check_obfuscated_script():
        CO_SIZES = 46, 36
        CO_NAMES = set(['pytransform', 'pyarmor_runtime', '__pyarmor__',
```

(continues on next page)

(continued from previous page)

```
                        '__name__', '__file__'])
        co = pytransform.sys._getframe(3).f_code
        if not ((set(co.co_names) <= CO_NAMES)
                and (len(co.co_code) in CO_SIZES)):
            raise RuntimeError('Unexpected obfuscated script')

    def check_mod_pytransform():
        CO_NAMES = set(['Exception', 'LoadLibrary', 'None', 'PYFUNCTYPE',
                        'PytransformError', '__file__', '_debug_mode',
                        '_get_error_msg', '_handle', '_load_library',
                        '_pytransform', 'abspath', 'basename', 'byteorder',
                        'c_char_p', 'c_int', 'c_void_p', 'calcsize', 'cdll',
                        'dirname', 'encode', 'exists', 'exit',
                        'format_platname', 'get_error_msg', 'init_pytransform',
                        'init_runtime', 'int', 'isinstance', 'join', 'lower',
                        'normpath', 'os', 'path', 'platform', 'print',
                        'pyarmor_init', 'pythonapi', 'restype', 'set_option',
                        'str', 'struct', 'sys', 'system', 'version_info'])

        colist = []

        for name in ('dllmethod', 'init_pytransform', 'init_runtime',
                     '_load_library', 'pyarmor_init', 'pyarmor_runtime'):
            colist.append(getattr(pytransform, name).{code})

        for name in ('init_pytransform', 'init_runtime'):
            colist.append(getattr(pytransform, name).{closure}[0].cell_contents.{code}
↪)
        colist.append(pytransform.dllmethod.{code}.co_consts[1])

        for co in colist:
            if not (set(co.co_names) < CO_NAMES):
                raise RuntimeError('Unexpected pytransform.py')

    def check_lib_pytransform(filename):
        size = 0x{size:X}
        n = size >> 2
        with open(filename, 'rb') as f:
            buf = f.read(size)
        fmt = 'I' * n
        checksum = sum(pytransform.struct.unpack(fmt, buf)) & 0xFFFFFFFF
        if not checksum == 0x{checksum:X}:
            raise RuntimeError("Unexpected %s" % filename)
    try:
        check_obfuscated_script()
        check_mod_pytransform()
        check_lib_pytransform(pytransform._pytransform._name)
    except Exception as e:
        print("Protection Fault: %s" % e)
        pytransform.sys.exit(1)

protect_pytransform()

if __name__ == '__main__':
    ...
```

If you want to hide the code more thoroughly, try to use any other tool such as ASProtect, VMProtect to protect dynamic library *_pytransform* which is distributed with obfuscatd scripts.

CHAPTER 5

Understanding Obfuscated Scripts

5.1 Global Capsule

The `.pyarmor_capsule.zip` in the `HOME` path called *Global Capsule*. It's created implicitly when executing command `pyarmor obfuscate`. *PyArmor* will read data from *Global Capsule* when obfuscating scripts or generating licenses for obfuscated scripts.

5.2 Obfuscated Scripts

After the scripts are obfuscated by *PyArmor*, in the *dist* folder you find all the files you distribute to your users:

```
myscript.py
mymodule.py

pytransform.py
_pytransform.so, or _pytransform.dll in Windows, _pytransform.dylib in MacOS
pyshield.key
pyshield.lic
product.key
license.lic
```

The obfuscated scripts are normal Python scripts.

The module *dist/mymodule.py* would be like this:

```
__pyarmor__(__name__, __file__, b'\x06\x0f...')
```

The entry script *dist/myscript.py* would be like this:

```
from pytransfrom import pyarmor_runtime
pyarmor_runtime()
```

(continues on next page)

(continued from previous page)

```
__pyarmor__(__name__, __file__, b'\x0a\x02...')
```

5.3 Bootstrap Code

The first 2 lines in the entry script called *Bootstrap Code*. It's only in the entry script:

```
from pytransfrom import pyarmor_runtime
pyarmor_runtime()
```

5.4 Runtime Files

Except obfuscated scripts, all the other files are called *Runtime Files*:

- *pytransform.py*, a normal python module
- *_pytransform.so*, or *_pytransform.dll*, or *_pytransform.dylib* a dynamic library implements core functions
- *pyshield.key*, *pyshield.lic*, *product.key*, *license.lic*, data files

All of them are required to run obfuscated scripts.

5.5 The *license.lic*

There is a special runtime file *license.lic*. The default one, which generated as executing `pyarmor obfuscate`, allows obfuscated scripts run in any machine and never expired.

To change this behaviour, use command `pyarmor licenses` to generate new *license.lic* and overwrite the default one.

5.6 Running Obfuscated Scripts

The obfuscated scripts is a normal python script, it can be run by normal python interpreter:

```
cd dist
python myscript.py
```

Firt *Bootstrap Code* is executed:

- Import *pyarmor_runtime* from *pytransform.py*
- **Execute *pyarmor_runtime***
  - Load dynamic library *_pytransform* by *ctypes*
  - Check *license.lic* in the same path
  - Add there builtin functions *__pyarmor__*, *__enter_armor__*, *__exit_armor__*

After that:

- Call *__pyarmor__* to import the obfuscated module.

- Call *__enter_armor__* to restore code object of function before executing each function
- Call *__exit_armor__* to obfuscate code object of function after each function return

More information, refer to *How to Obfuscate Python Scripts* and *How to Run Obfuscated Script*

5.7 Key Points to Use Obfuscated Scripts

- The obfuscated script is a normal python script, so it can be seamless to replace original script.
- There is only one thing changed, the following code must be run before using any obfuscated script:

  ```
  from pytransform import pyarmor_runtime
  pyarmor_runtime()
  ```

- It can be put in any script anywhere, only if it run in the same Python interpreter. It will create some builtin function to deal with obfuscated code.
- The extra runtime file *pytransform.py* must be in any Python path in target machine. *pytransform.py* need load dynamic library *_pytransform* by *ctypes*. It may be
  - *_pytransform.so* in Linux
  - *_pytransform.dll* in Windows
  - *_pytransform.dylib* in MacOS

  This file is dependent-platform, download the right one to the same path of *pytransform.py* according to target platform. All the prebuilt dynamic libraries list here

  http://pyarmor.dashingsoft.com/downloads/platforms/

- By default *pytransform.py* search dynamic library *_pytransform* in the same path. Check *pytransform._load_library* to find the details.
- All the other *Runtime Files* should in the same path as dynamic library *_pytransform*
- If *Runtime Files* locate in some other path, change *Bootstrap Code*:

  ```
  from pytransform import pyarmor_runtime
  pyarmor_runtime('/path/to/runtime-files')
  ```

CHAPTER 6

How *PyArmor* Does It

Look at what happened after `foo.py` is obfuscated by PyArmor. Here are the files list in the output path `dist`:

```
foo.py

pytransform.py
_pytransform.so, or _pytransform.dll in Windows, _pytransform.dylib in MacOS

pyshield.key
pyshield.lic
product.key
license.lic
```

`dist/foo.py` is obfuscated script, the content is:

```
from pytransfrom import pyarmor_runtime
pyarmor_runtime()

__pyarmor__(__name__, __file__, b'\x06\x0f...')
```

All the other extra files called *Runtime Files*, which are required to run or import obfuscated scripts. So long as runtime files are in any Python path, obfuscated script *dist/foo.py* can be used as normal Python script. That is to say:

The original python scripts can be replaced with obfuscated scripts seamlessly.

6.1 How to Obfuscate Python Scripts

How to obfuscate python scripts by PyArmor?

First compile python script to code object:

```
char *filename = "foo.py";
char *source = read_file( filename );
PyCodeObject *co = Py_CompileString( source, "<frozen foo>", Py_file_input );
```

Then change code object as the following way

- Wrap byte code `co_code` within a `try...finally` block:

```
wrap header:

        LOAD_GLOBALS    N (__armor_enter__)     N = length of co_consts
        CALL_FUNCTION   0
        POP_TOP
        SETUP_FINALLY   X (jump to wrap footer) X = size of original byte code

changed original byte code:

        Increase oparg of each absolute jump instruction by the size of wrap header

        Obfuscate original byte code

        ...

wrap footer:

        LOAD_GLOBALS    N + 1 (__armor_exit__)
        CALL_FUNCTION   0
        POP_TOP
        END_FINALLY
```

- Append function names `__armor_enter`, `__armor_exit__` to `co_consts`
- Increase `co_stacksize` by 2
- Set CO_OBFUSCAED (0x80000000) flag in `co_flags`
- Change all code objects in the `co_consts` recursively

Next serializing reformed code object and obfuscate it to protect constants and literal strings:

```
char *string_code = marshal.dumps( co );
char *obfuscated_code = obfuscate_algorithm( string_code  );
```

Finally generate obfuscated script:

```
sprintf( buf, "__pyarmor__(__name__, __file__, b'%s')", obfuscated_code );
save_file( "dist/foo.py", buf );
```

The obfuscated script is a normal Python script, it looks like this:

```
__pyarmor__(__name__, __file__, b'\x01\x0a...')
```

6.2 How to Run Obfuscated Script

How to run obfuscated script `dist/foo.py` by Python Interpreter?

The first 2 lines, which called `Bootstrap Code`:

```
from pytransfrom import pyarmor_runtime
pyarmor_runtime()
```

It will fulfil the following tasks

- Load dynamic library `_pytransform` by `ctypes`
- Check `dist/license.lic` is valid or not
- Add 3 cfunctions to module `builtins`: `__pyarmor__`, `__armor_enter__`, `__armor_exit__`

The next code line in `dist/foo.py` is:

```
__pyarmor__(__name__, __file__, b'\x01\x0a...')
```

`__pyarmor__` is called, it will import original module from obfuscated code:

```
static PyObject *
__pyarmor__(char *name, char *pathname, unsigned char *obfuscated_code)
{
    char *string_code = restore_obfuscated_code( obfuscated_code );
    PyCodeObject *co = marshal.loads( string_code );
    return PyImport_ExecCodeModuleEx( name, co, pathname );
}
```

After that, in the runtime of this python interpreter

- `__armor_enter__` is called as soon as code object is executed, it will restore byte-code of this code object:

```
static PyObject *
__armor_enter__(PyObject *self, PyObject *args)
{
    // Got code object
    PyFrameObject *frame = PyEval_GetFrame();
    PyCodeObject *f_code = frame->f_code;

    // Increase refcalls of this code object
    // Borrow co_names->ob_refcnt as call counter
    // Generally it will not increased  by Python Interpreter
    PyObject *refcalls = f_code->co_names;
    refcalls->ob_refcnt ++;

    // Restore byte code if it's obfuscated
    if (IS_OBFUSCATED(f_code->co_flags)) {
        restore_byte_code(f_code->co_code);
        clear_obfuscated_flag(f_code);
    }

    Py_RETURN_NONE;
}
```

- `__armor_exit__` is called so long as code object completed execution, it will obfuscate byte-code again:

```
static PyObject *
__armor_exit__(PyObject *self, PyObject *args)
{
    // Got code object
    PyFrameObject *frame = PyEval_GetFrame();
    PyCodeObject *f_code = frame->f_code;

    // Decrease refcalls of this code object
    PyObject *refcalls = f_code->co_names;
    refcalls->ob_refcnt --;
```

(continues on next page)

(continued from previous page)

```
    // Obfuscate byte code only if this code object isn't used by any function
    // In multi-threads or recursive call, one code object may be referenced
    // by many functions at the same time
    if (refcalls->ob_refcnt == 1) {
        obfuscate_byte_code(f_code->co_code);
        set_obfuscated_flag(f_code);
    }

    // Clear f_locals in this frame
    clear_frame_locals(frame);

    Py_RETURN_NONE;
}
```

6.3 Special Handling of Entry Script

Before obfuscating entry scipt, if there is line like this:

```
if __name__ == '__main__':
```

PyArmor will patch this entry script, insert some code to protect dynamic library before this line:

```
def protect_pytransform():

    import pytransform

    def check_obfuscated_script():
        CO_SIZES = 46, 36
        CO_NAMES = set(['pytransform', 'pyarmor_runtime', '__pyarmor__',
                        '__name__', '__file__'])
        co = pytransform.sys._getframe(3).f_code
        if not ((set(co.co_names) <= CO_NAMES)
                and (len(co.co_code) in CO_SIZES)):
            raise RuntimeError('Unexpected obfuscated script')

    def check_mod_pytransform():
        CO_NAMES = set(['Exception', 'LoadLibrary', 'None', 'PYFUNCTYPE',
                        'PytransformError', '__file__', '_debug_mode',
                        '_get_error_msg', '_handle', '_load_library',
                        '_pytransform', 'abspath', 'basename', 'byteorder',
                        'c_char_p', 'c_int', 'c_void_p', 'calcsize', 'cdll',
                        'dirname', 'encode', 'exists', 'exit',
                        'format_platname', 'get_error_msg', 'init_pytransform',
                        'init_runtime', 'int', 'isinstance', 'join', 'lower',
                        'normpath', 'os', 'path', 'platform', 'print',
                        'pyarmor_init', 'pythonapi', 'restype', 'set_option',
                        'str', 'struct', 'sys', 'system', 'version_info'])

        colist = []

        for name in ('dllmethod', 'init_pytransform', 'init_runtime',
                     '_load_library', 'pyarmor_init', 'pyarmor_runtime'):
            colist.append(getattr(pytransform, name).{code})
```

(continues on next page)

(continued from previous page)

```
        for name in ('init_pytransform', 'init_runtime'):
            colist.append(getattr(pytransform, name).{closure}[0].cell_contents.{code}
↪)
        colist.append(pytransform.dllmethod.{code}.co_consts[1])

        for co in colist:
            if not (set(co.co_names) < CO_NAMES):
                raise RuntimeError('Unexpected pytransform.py')

    def check_lib_pytransform(filename):
        size = 0x{size:X}
        n = size >> 2
        with open(filename, 'rb') as f:
            buf = f.read(size)
        fmt = 'I' * n
        checksum = sum(pytransform.struct.unpack(fmt, buf)) & 0xFFFFFFFF
        if not checksum == 0x{checksum:X}:
            raise RuntimeError("Unexpected %s" % filename)
    try:
        check_obfuscated_script()
        check_mod_pytransform()
        check_lib_pytransform(pytransform._pytransform._name)
    except Exception as e:
        print("Protection Fault: %s" % e)
        pytransform.sys.exit(1)

protect_pytransform()

if __name__ == '__main__':
    ...
```

After the entry script is obfuscated, the *Bootstrap Code* will be inserted at the beginning of the obfuscated script.

CHAPTER 7

How To Pack Obfuscated Scripts

The obfuscated scripts generated by PyArmor can replace Python scripts seamlessly, but there is an issue when packing them into one bundle by PyInstaller, py2exe, py2app, cx_Freeze:

All the dependencies of obfuscated scripts CAN NOT be found at all

To solve this problem, the common solution is

1. Find all the dependenices by original scripts.
2. Add runtimes files required by obfuscated scripts to the bundle
3. Replace original scipts with obfuscated in the bundle
4. Replace entry scrirpt with obfuscated one

Depend on what tool used, there are different ways.

First obfuscate scripts to `dist/obf`:

```
pyarmor obfuscate --output dist/obf hello.py
```

7.1 Work with PyInstaller

Install `pyinstaller`:

```
pip install pyinstaller
```

Generate specfile, add the obfuscated entry script and data files required by obfuscated scripts:

```
pyinstaller --add-data dist/obf/*.lic
            --add-data dist/obf/*.key
            --add-data dist/obf/_pytransform.*
            hello.py dist/obf/hello.py
```

Update specfile `hello.spec`, insert the following lines after the `Analysis` object. The purpose is to replace all the original scripts with obfuscated ones:

```
a.scripts[0] = 'hello', 'dist/obf/hello.py', 'PYSOURCE'
for i in range(len(a.pure)):
    if a.pure[i][1].startswith(a.pathex[0]):
        a.pure[i] = a.pure[i][0], a.pure[i][1].replace(a.pathex[0], os.path.abspath(
↪'dist/obf'), a.pure[i][2]
```

Run patched specfile to build final distribution:

```
pyinstaller hello.spec
```

Check obfuscated scripts work:

```
# It works
dist/hello/hello.exe

rm dist/hello/license.lic

# It should not work
dist/hello/hello.exe
```

7.2 Work with py2exe

For Python3.3 and later:

```
pip install py2exe
```

Build bundle executable to `dist` with separated library:

```
build_exe --library library.zip hello.py
```

Build bundle executable with the obfuscated entry to `dist/obf/dist`, all the other obfuscated scripts should be include by `-i name` or `-p pkgname`:

```
( cd dist/obf;
  build_exe --library library.zip -i queens hello.py )
```

Update `dist/obf/library.zip`, which only includes the obfuscated scripts, merge all the dependenices files from `dist/library.zip` into it.

Copy all the files to final output:

```
cp -a dist/obf/dist/* dist/
```

Copy runtime files required by obfuscated scripts to finial output:

```
( cd dist/obf;
  cp pyshield.key pyshield.lic product.key license.lic _pytransform.dll ../dist/ )
```

Check obfuscated scripts work:

```
# It works
dist/hello.exe
```

(continues on next page)

(continued from previous page)

```
rm dist/license.lic

# It should not work
dist/hello.exe
```

For Python2, write a `setup.py` and run `py2exe` as the following way:

```
python setup.py py2exe hello.py
```

7.3 Work with cx_Freeze 5

Install `cx_Freeze`:

```
pip install cx_Freeze
```

Build bundle executable to `dist`:

```
cxfreeze --target-dir=dist hello.py
```

Build bundle executable with the obfuscated entry to `dist/obf/dist`, all the other obfuscated scripts should be include by `--include-modules NAMES`:

```
cd dist/obf
cxfreeze --target-dir=dist --include-modules=queens hello.py
```

Update `dist/obf/python34.zip`, which only includes the obfuscated scripts, merge all the dependenices files from `dist/python34.zip` into it.

Copy all the files to final output:

```
cp -a dist/obf/dist/* dist/
```

Copy runtime files required by obfuscated scripts to finial output:

```
( cd dist/obf;
  cp pyshield.key pyshield.lic product.key license.lic _pytransform.dll ../dist/ )
```

Check obfuscated scripts work:

```
# It works
dist/hello.exe

rm dist/license.lic

# It should not work
dist/hello.exe
```

CHAPTER 8

Using Project

Project is a folder include its own capsule and configuration file, which used to manage obfuscated scripts.

There are several advantages to manage obfuscated scripts by Project:

- Increment build, only updated scripts are obfuscated since last build
- Filter obfuscated scripts in the project, exclude some scripts
- More convenient to manage obfuscated scripts

8.1 Managing Obfuscated Scripts With Project

Use command `init` to create a project:

```
pyarmor init --src=examples/pybench --entry=pybench.py projects/pybench
```

The project path *projects/pybench* will be created, and ther are 3 files in it:

```
.pyarmor_config
.pyarmor_capsule.zip
pyarmor.bat or pyarmor
```

`.pyarmor_config` is project configuration of JSON format.

`.pyarmor_capsule.zip` called `Projct Capsule`, is used to obfuscate scripts in this project and generate new license file for those obfuscated scripts.

The last file is shell script to call `pyarmor` in this project.

The common usage for project is to do any thing in the project path:

```
cd projects/pybench
```

Show project information:

```
./pyarmor info
```

Obfuscate all the scripts in this project:

```
./pyarmor build
```

Exclude the `dist`, `test`, the *.py* files in these folder will not be obfuscated:

```
./pyarmor config --manifest "include *.py, prune dist, prune test"
```

Force rebuild:

```
./pyarmor build --force
```

Run obfuscated script:

```
cd dist
python pybench.py
```

After some scripts changed, just run `build` again:

```
cd projects/pybench
./pyarmor build
```

8.2 Project Configuration File

Each project has a configure file. It's a json file named `.pyarmor_config` stored in the project path.

- name

  Project name.

- title

  Project title.

- src

  Base path to match files by manifest template string.

  Generally it's absolute path.

- manifest

  A string specifies files to be obfuscated, same as MANIFEST.in of Python Distutils, default value is:

  ```
  global-include *.py
  ```

  It means all files anywhere in the *src* tree matching.

  Multi manifest template commands are spearated by comma, for example:

  ```
  global-include *.py, exclude __mainfest__.py, prune test
  ```

  Refer to https://docs.python.org/2/distutils/sourcedist.html#commands

- is_package

Available values: 0, 1, None

When it's set to 1, the basename of *src* will be appended to *output* as the final path to save obfuscated scripts, and runtime files are still in the path *output*

When init a project and no *–type* specified, it will be set to 1 if there is *__init__.py* in the path *src*, otherwise it's None.

- disable_restrict_mode

  Available values: 0, 1, None

  When it's None or 0, obfuscated scripts can not be imported from outer scripts. It's more quickly.

  When it's set to 1, it the obfuscated scripts are allowed to be imported by outer scripts. It's more security.

  By default it's set to 1.

  Refer to *Restrict Mode*

- entry

  A string includes one or many entry scripts.

  When build project, insert the following bootstrap code for each entry:

  ```
  from pytransform import pyarmor_runtime
  pyarmor_runtime()
  ```

  The entry name is relative to *src*, or filename with absolute path.

  Multi entries are separated by comma, for example:

  ```
  main.py, another/main.py, /usr/local/myapp/main.py
  ```

  Note that entry may be NOT obfuscated, if *manifest* does not specify this entry. In this case, bootstrap code will be inserted into the header of entry script either. So that it can import other obfuscated modules.

- output

  A path used to save output of build. It's relative to project path.

- capsule

  Filename of project capsule. It's relative to project path if it's not absolute path.

- obf_module_mode

  How to obfuscate whole code object of module:

  - none

    No obfuscate

  - des

    Obfuscate whole code object by DES algorithm

    The default value is *des*

- obf_code_mode

  How to obfuscate byte code of each code object:

  - none

    No obfuscate

  - des

    Obfuscate byte-code by DES algorithm

  - fast

    Obfuscate byte-code by a simple algorithm, it's faster than DES

  - wrap

    The wrap code is different from *des* and *fast*. In this mode, when code object start to execute, byte-code is restored. As soon as code object completed execution, byte-code will be obfuscated again.

  The default value is *wrap*.

- runtime_path

  None or any path.

  When run obfuscated scripts, where to find dynamic library *_pytransform*. The default value is None, it means it's in the same path of `pytransform.py`.

  It's useful when obfuscated scripts are packed into a zip file, for example, use py2exe to package obfuscated scripts. Set runtime_path to an empty string, and copy *Runtime Files* to same path of zip file, will solve this problem.

CHAPTER 9

The Differences of Obfuscated Scripts

There are something changed after Python scripts are obfuscated:

- Python Version in build machine must be same as in target machine. To be exact, the magic string value used to recognize byte-compiled code files (.pyc files) must be same.
- If Python interpreter is compiled with Py_TRACE_REFS or Py_DEBUG, it will crash to run obfuscated scripts.
- The callback function set by `sys.settrace`, `sys.setprofile`, `threading.settrace` and `threading.setprofile` will be ignored by obfuscated scripts.
- The attribute `__file__` of code object in the obfuscated scripts will be `<frozen name>` other than real filename. So in the traceback, the filename is shown as `<frozen name>`.

  Note that `__file__` of moudle 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__)
  ```

CHAPTER 10

Advanced Topics

10.1 Restrict Mode

PyArmor can obfuscate the scripts in restrict mode.

If the script is obfuscated in restrict mode:

- Each function (code object) will not be obfuscated again in runtime
- Any other code can not be insert into the obfuscated scripts
- The obfuscated module can not be imported from other clear python scripts

For examples, obfuscate `foo.py` in restrict mode:

```
pyarmor obfuscate --restrict foo.py
```

Each function is obfuscated before it's called. Once it's executed, it will not be obfuscated again.

And if adding `print` in the obfuscated script:

```
from pytransform import pyarmor_runtime
pyarmor_runtime()
__pyarmor__(__name__, __file__, b'...')
print('Something')
```

It will report error as running:

```
python foo.py
```

In a short word, the obfuscated script in restrict mode runs quickly, but maybe less security.

If the scripts is obfuscated in restrict mode, you should enable restrict mode either as generating new licenses for it:

```
pyarmor licenses --restrict --expired 2019-01-01 mycode
```

10.2 Show License Information

Maybe you'd like to show how many days left when you issue an expired license for obfuscated scripts.

Use `get_license_info` function in the module `pytransform.py` of *Runtime Files* to get license information of obfuscated scripts.

Here it's an example which explains how to do. Suppose there is a script `foo.py` will be obfuscated and distributed to the customer, it will print expired date and license code, then do something.

The content of foo.py:

```
def show_license_code():
    from pytransfrom import get_license_info

    info = get_license_info()
    print('This script is only for %s' % info['CODE]')
    print('This script will expired on %s' % info['expired'])

if __name__ == '__main__':
    show_license_code()
    do_something()
```

Let's obfuscate foo.py at first:

```
pyarmor obfuscate foo.py
pyarmor licenses --expired 2019-01-01 Brave-Tom
cp licenses/Brave-Tom/license.lic dist/license.ic
```

Then run this obfuscated script in the output path `dist`:

```
cd dist/
python foo.py
```

The output will be:

```
This script is only for Brave-Tom
This script will expired on 2019-01-01
```

CHAPTER 11

Man Page

PyArmor is a command line tool used to obfuscate python scripts, bind obfuscated scripts to fixed machine or expire obfuscated scripts.

The syntax of the `pyarmor` command is:

```
pyarmor <command> [options]
```

The most commonly used pyarmor commands are:

```
obfuscate    Obfuscate python scripts
licenses     Generate new licenses for obfuscated scripts
pack         Pack obfuscated scripts to one bundle
hdinfo       Show hardware information
```

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

11.1 obfuscate

Obfuscate python scripts.

SYNOPSIS:

```
pyarmor obfuscate <options> SCRIPT...
```

DESCRIPTION

PyArmor first checks whether `.pyarmor_capsule.zip` exists in the `HOME` path. If not, make it.

Then search all the *.py* files in the path of entry script, and obfuscate them in the default output path *dist*.

Next generate default `license.lic` for obfuscated scripts and make all the other *Runtime Files* in the *dist* path.

Finally insert *Bootstrap Code* into each entry script.

OPTIONS

-O PATH, --output PATH Output path

-r, --recursive Match files recursively

--capsule CAPSULE Use this capsule to obfuscate scripts

11.2 licenses

Generate new licenses for obfuscated scripts.

SYNOPSIS:

```
pyarmor licenses <options> CODE
```

OPTIONS:

-C CAPSULE, --capsule CAPSULE Use this capsule to generate new licenses

-O OUTPUT, --output OUTPUT Output path

-e YYYY-MM-DD, --expired YYYY-MM-DD Expired date for this license

-d SN, --bind-disk SN Bind license to serial number of harddisk

-4 IPV4, --bind-ipv4 IPV4 Bind license to ipv4 addr

-m MACADDR, --bind-mac MACADDR Bind license to mac addr

11.3 pack

Obfuscate the scripts and pack them into one bundle.

SYNOPSIS:

```
pyarmor pack <options> SCRIPT
```

OPTIONS:

-t TYPE, --type TYPE cx_Freeze, py2exe, py2app, PyInstaller(default).

-O OUTPUT, --output OUTPUT Directory to put final built distributions in.

11.4 hdinfo

Show hardware information of this machine, such as serial number of hard disk, mac address of network card etc. The information got here could be as input data to generate license file for obfuscated scripts.

SYNOPSIS:

```
pyarmor hdinfo
```

CHAPTER 12

When Things Go Wrong

Turn on debugging output to get more error information:

```
python -d pyarmor.py ...
PYTHONDEBUG=y pyarmor ...
```

12.1 Segment fault

In the following cases, obfuscated scripts will crash

- Running obfuscated script by the debug version Python
- Obfuscating scripts by Python 2.6 but running the obfuscated scripts by Python 2.7

12.2 Could not load *_pytransform*

Generally, the dynamic library *_pytransform* is in the same path of obfuscated scripts. It may be:

- *_pytransform.so* in Linux
- *_pytransform.dll* in Windows
- *_pytransform.dylib* in MacOS

First check whether the file exists. If it exists:

- Check the permissions of dynamic library

  If there is no execute permissions in Windows, it will complain: *[Error 5] Access is denied*

- Check whether *ctypes* could load *_pytransform*:

  ```
  from pytransform import _load_library
  m = _load_library(path='/path/to/dist')
  ```

- Try to set the runtime path in the *Bootstrap Code* of entry script:

  ```
  from pytransform import pyarmor_runtime
  pyarmor_runtime('/path/to/dist')
  ```

Still doesn't work, report an issue

12.3 The *license.lic* generated doesn't work

The key is that the capsule used to obfuscate scripts must be same as the capsule used to generate licenses.

If obfuscate scripts by command *pyarmor obfuscate*, *Global Capsule* is used implicitly. If obfuscate scripts by command *pyarmor build*, the project capsule in the project path is used.

When generating new licenses for obfuscated scripts, if run command *pyarmor licenses* in project path, the project capsule is used implicitly, otherwise *Global Capsule*.

The *Global Capsule* will be changed if the trial license file of *PyArmor* is replaced with normal one, or it's deleted occasionally (which will be generated implicitly as running command *pyarmor obfuscate* next time).

The project capsule is overwrited when running command *pyarmor init* in the project path created before.

In any cases, generating new license file with the different capsule will not work for the obfuscated scripts before. If the old capsule is gone, one solution is to obfuscate these scripts by the new capsule again.

12.4 Two types of *license.lic*

In pyarmor, there are 2 types of *license.lic*

- *license.lic* of *PyArmor*, which locates in the source path of *PyArmor*. It's required to run *pyarmor*
- *license.lic* of Obfuscated Scripts, which is generated as obfuscating scripts or generating new licenses. It's required to run obfuscated scripts.

Each project has its own capsule *.pyarmor_capsule.zip* in project path. This capsule is generated when run command *pyarmor init* to create a project. And *license.lic* of *PyArmor* will be as an input file to make this capsule.

When runing command *pyarmor build* or *pyarmor licenses*, it will generate a *license.lic* in project output path for obfuscated scripts. Here the project capsule *.pyarmor_capsule.zip* will be as input file to generate this *license.lic* of Obfuscated Scripts.

So the relation between 2 *license.lic* is:

```
license.lic of PyArmor --> .pyarmor_capsule.zip --> license.lic of Obfuscated Scripts
```

If the scripts are obfuscated by command *pyarmor obfuscate* other than by project, *Global Capsule* is used implicitly.

12.5 Work with subprocess and multiprocessing

When creating new process by *Popen* or *Process*, note that *Bootstrap Code* must be called before importing any obfuscated code in sub-process. Otherwise it will report:

```
NameError: name '__pyarmor__' is not defined
```

12.6 Marshal loads failed when running xxx.py

- The version Python to running scripts is different from the version Python to obfuscated script
- The capsule used to generated license for obfuscated scripts is different from the capsule used to obfuscate the scripts.

CHAPTER 13

License

PyArmor is published as shareware. Free trial version never expires, the limitation is

- *Global Capsule* in trial version is fixed.

There are 2 basic types of licenses issued for the software. These are:

- A natural person usage license for home users. The user purchases one license to use the software on his own computer.

  Home users may use their natural person usage license on all computers and embedded devices which are property of the license owner.

- A juridical person usage license for business users. The user purchases one license to use the software for one product or one project of an organization.

  Business users may use their juridical person usage license on all computers and embedded devices for one product or project. But they require another license for different product or project.

13.1 Purchase

To buy a license, please visit the following url

https://order.shareit.com/cart/add?vendorid=200089125&PRODUCT{[}300871197{]}=1

A registration code will be sent to your immediately after payment is completed successfully.

After you receive the email which includes registration code, copy registration code only (no newline), then replace the content of `*pyarmor-folder*/license.lic` with it.

Note that there are 2 types of `license.lic`, this one locates in the source path of *PyArmor*. It's used by *PyArmor*. The other locates in the same path as obfuscated scripts, It's used by obfuscated scripts.

Check new license works, execute this command:

```
pyarmor --version
```

The result should show `PyArmor Version X.Y.Z` and registration code.

After new license takes effect, you need obfuscate the scripts again, and a random *Global Capsule* will be generated implicitly when you run command `pyarmor obfuscate`

The registration code is valid forever, it can be used permanently.

CHAPTER 14

Indices and tables

- genindex
- modindex
- search

Index