PyArmor Documentation

发布 *6.2.0*

Jondy Zhao

2020 年 05 月 15 日

Contents

1 安装 PyArmor **3**

1.1 可用的命令 . . . 3

1.2 完全卸载 . . . 4

2 使用 PyArmor **5**

2.1 加密脚本 . . . 5

2.2 发布加密的脚本 . . . 6

2.3 生成新的许可文件 . . . 7

2.4 扩展其他认证方式 . . . 8

2.5 加密单个模块 . . . 8

2.6 加密整个 Python 包 . . . 8

2.7 打包加密脚本 . . . 8

2.8 进一步提高加密脚本的安全性 . . . 9

3 高级用法 **11**

3.1 使用超级模式加密脚本 . . . 11

3.2 加密和使用多个包 . . . 12

3.3 如何加密能和其他加密包共存的包 . . . 13

3.4 跨平台发布加密脚本 . . . 14

3.5 使用不同版本 Python 加密脚本 . . . 16

3.6 在没有加密的脚本中运行引导代码 . . . 16

3.7 让 Python 自动识别加密脚本 . . . 18

3.8 使用不同的模式来加密脚本 . . . 19

3.9 使用插件扩展认证方式 . . . 20

3.10 打包加密脚本成为一个单独的可执行文件 . . . 20

3.11 使用定制的.spec 文件打包加密脚本 . . . 21

3.12 使用约束模式增加加密脚本安全性 . . . 22

3.13 使用插件来进一步提高安全性 . . . 23
3.14 在 Python 脚本内部调用 *pyarmor* . . . 26
3.15 运行加密脚本的时候周期性的检查许可文件 . . . 27
3.16 使用 Nuitka 发布加密脚本 . . . 27
3.17 使用 Cython 发布加密脚本 . . . 28
3.18 使用 PyUpdater 发布加密脚本 . . . 29
3.19 绑定加密脚本到固定的 Python 解释器 . . . 29
3.20 定制交叉保护脚本 . . . 31
3.21 如何把许可文件 license.lic 存放到任意位置 . . . 32

4 使用实例 **33**
4.1 加密并打包 PyQt 应用 . . . 33
4.2 使用 Apache 的 mod_wsgi 发布加密的 Django 应用 . . . 34

5 使用工程 **37**
5.1 使用工程管理加密脚本 . . . 37
5.2 使用不同加密模式 . . . 38
5.3 使用子工程加密需要特殊处理的脚本 . . . 38
5.4 工程配置文件 . . . 39

6 命令手册 **45**
6.1 obfuscate . . . 46
6.2 licenses . . . 51
6.3 pack . . . 52
6.4 hdinfo . . . 56
6.5 init . . . 56
6.6 config . . . 57
6.7 build . . . 59
6.8 info . . . 60
6.9 check . . . 61
6.10 banchmark . . . 61
6.11 register . . . 62
6.12 download . . . 62
6.13 runtime . . . 63

7 了解加密脚本 **67**
7.1 全局密钥箱 . . . 67
7.2 加密后的脚本 . . . 67
7.3 引导代码 . . . 69
7.4 运行辅助包 . . . 69
7.5 加密脚本的许可文件 . . . 70
7.6 使用加密脚本的基本原则 . . . 70
7.7 加密脚本和原脚本的区别 . . . 71

7.8 第三方解释器的支持 . . . 72

8 PyArmor 的工作原理 73
8.1 如何加密脚本 . . . 74
8.2 如何处理插件 . . . 75
8.3 对主脚本的特殊处理 . . . 77
8.4 如何运行加密脚本 . . . 79
8.5 如何打包加密脚本 . . . 81

9 运行时刻模块 *pytransform* 85
9.1 内容 . . . 85
9.2 示例 . . . 86

10 支持的平台列表 89
10.1 标准平台名称 . . . 90

11 加密模式 95
11.1 超级模式 . . . 95
11.2 高级模式 . . . 96
11.3 代码加密模式 . . . 96
11.4 代码包裹模式 . . . 97
11.5 模块加密模式 . . . 98
11.6 约束模式 . . . 98

12 加密脚本的性能 103

13 PyArmor 的安全性 105
13.1 交叉保护机制 . . . 105

14 常见问题 109
14.1 Segment fault . . . 110
14.2 启动问题 . . . 110
14.3 加密脚本的问题 . . . 112
14.4 运行加密脚本的问题 . . . 113
14.5 打包加密问题 . . . 115
14.6 PyArmor 注册问题 . . . 116
14.7 已知的问题 . . . 116
14.8 其他问题 . . . 116

15 软件许可 117
15.1 购买 . . . 118
15.2 Q & A . . . 119

索引 121

版本 PyArmor 6.2

主页 http://pyarmor.dashingsoft.com/index-zh.html

联系方式 jondy.zhao@gmail.com

作者 赵俊德

PyArmor 是一个用于加密和保护 Python 脚本的工具。它能够在运行时刻保护 Python 脚本的二进制代码不被泄露，设置加密后 Python 源代码的有效期限，绑定加密后的 Python 源代码到硬盘、网卡等硬件设备。它的保障机制主要包括

- 加密编译后的代码块，保护模块中的字符串和常量
- 在脚本运行时候动态加密和解密每一个函数（代码块）的二进制代码
- 代码块执行完成之后清空堆栈局部变量
- 通过授权文件限制加密后脚本的有效期和设备环境

PyArmor 支持 Python 2.6, 2.7 和 Python 3

PyArmor 在下列平台进行了充分测试: `Windows`, `Mac OS X`, and `Linux`

PyArmor 已经成功应用于 `FreeBSD` 和嵌入式系统，例如 `Raspberry Pi`, `Banana Pi`, `Orange Pi`, `TS-4600 / TS-7600` 等，但是这些平台下面没有进行充分测试。

内容:

CHAPTER 1

安装 PyArmor

PyArmor 可以直接从这里 PyPi 下载, 但是更方便的方式是通过 pip ，直接运行下面的命令进行安装:

```
pip install pyarmor
```

需要升级的话，执行下面的命令:

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

另外还有一个图形界面的包，需要的话可以安装:

```
pip install pyarmor-webui
```

一旦成功安装，就可以直接运行命令 `pyarmor` ，例如:

```
pyarmor --version
```

这个命令会显示版本信息 `PyArmor Version X.Y.Z` 或者 `PyArmor Trial Version X.Y.Z`.

如果命令无法执行，请检查环境变量中是否包含可执行文件所在的路径。

1.1 可用的命令

使用 *pip* 安装之后，有两个可用的命令:

- `pyarmor` 这是主要的工具，参考使用 *PyArmor*.

- pyarmor-webui 用来打开网页版的可视化界面

如果没有使用 *pip* 进行安装，上述命令无法使用，需要运行 *Python* 来执行相应的脚本。pyarmor 等价于执行 *pyarmor-folder*/pyarmor.py, pyarmor-webui 等价于执行 *pyarmor-folder*/pyarmor/webui/server.py

1.2 完全卸载

下列文件可能会在 *pyarmor* 运行时被创建:

```
~/.pyarmor/.pyarmor_capsule.zip          (从 v6.2.0 开始)
~/.pyarmor/license.lic                   (从 v5.8.0 开始)
~/.pyarmor/platforms/

{pyarmor-folder}/license.lic             (在 v5.8.0 之前)
~/.pyarmor_capsule.zip                   (在 v6.2.0 之前)
```

执行下面的命令进行完全卸载:

```
pip uninstall pyarmor
rm -rf ~/.pyarmor

rm -rf {pyarmor-folder}                  (在 v5.8.0 之前)
rm -rf ~/.pyarmor_capsule.zip            (在 v6.2.0 之前)
```

CHAPTER 2

使用 PyArmor

命令 `pyarmor` 的基本语法为:

    `pyarmor` [command] [options]

2.1 加密脚本

命令 `obfuscate` 用来加密脚本。最常用的一种情况是切换到脚本 *myscript.py* 所在的路径，然后执行:

```
pyarmor obfuscate myscript.py
```

PyArmor 会加密 `myscript.py` 和相同目录下面的所有 `*.py` 文件:

- 在用户根目录下面创建 `.pyarmor_capsule.zip` （仅当不存在的时候创建）
- 创建输出子目录 `dist`
- 生成加密的主脚本 `myscript.py` 保存在输出目录 `dist`
- 加密相同目录下其他所有 `*.py` 文件，保存到输出目录 `dist`
- 生成运行加密脚本所需要的全部辅助文件，保存到输出目录 `dist`

输出目录 `dist` 包含运行加密脚本所需要的全部文件:

```
dist/
    myscript.py
```

(下页继续)

(续上页)

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

除了加密脚本之外，额外的那个目录 *pytransform* 叫做运行辅助包 ，它是运行加密脚本不可缺少的。

通常情况下第一个脚本叫做主脚本，它加密后的内容如下:

```
from pytransform import pyarmor_runtime
pyarmor_runtime()
__pyarmor__(__name__, __file__, b'\x06\x0f...')
```

其中前两行是引导代码, 它们只在主脚本出现，并且只能被运行一次。对于其他所有加密脚本，只有这样一行:

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

运行加密脚本:

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

默认情况下，只有和主脚本相同目录的其他 `*.py` 会被同时加密。如果想递归加密子目录下的所有 `*.py` 文件，使用下面的命令:

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

2.2 发布加密的脚本

发布加密脚本给客户只需要把输出路径 *dist* 的所有文件拷贝过去即可。需要注意的是运行辅助包 必须也拷贝到运行环境，否则无法运行加密脚本。

运行辅助包 并不是必须和加密脚本在一起，也可以拷贝到运行环境的任意的 Python 路径下面，只要能正确 *import pytransform* 就可以。

关于加密脚本的安全性的说明，参考*PyArmor 的安全性*

注解: 运行加密脚本不需要安装 PyArmor，没有必要在运行环境里面安装 PyArmor

2.3 生成新的许可文件

使用命令*licenses* 为加密脚本生成新的许可文件 `license.lic`

加密脚本的同时会在输出目录下面生成一个默认许可文件 `dist/license.lic` ，它允许加密脚本运行在任何设备上并且永不过期。

如果需要设置加密脚本的使用期限或者限制脚本在特定的机器使用，需要生成新的许可文件，并覆盖默认许可文件。

例如:

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

执行这条命令 PyArmor 会生成一个带有效期的认证文件:

- 从 `.pyarmor_capsule.zip` 读取相关数据
- 创建 `license.lic` ，保存在 `licenses/code-001`
- 创建 `license.lic.txt` ，保存在 `licenses/code-001`

然后，使用新生成的许可文件覆盖默认的许可文件:

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

这样，加密脚本在 2019 年 1 月 1 日之后就无法在运行了。

如果想绑定加密脚本到固定机器上，首先在该机器上面运行下面的命令获取硬件信息:

```
pyarmor hdinfo
```

然后在生成绑定到固定机器的许可文件:

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

同样，覆盖默认许可文件，这样加密脚本就只能在指定机器上运行:

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

cd dist/
python myscript.py
```

注解: 在 v5.7.0 之前，默认许可文件 `license.lic` 没有在 `dist/pytransform` ，而是在 `dist`

2.4 扩展其他认证方式

除了上述认证方式之外，还可以在 Python 脚本中增加其他任何认证代码，因为加密的脚本对于客户来说就是黑盒子。当需要扩展认证方式的时候，那么需要了解 运行时刻模块 pytransform 提供的相关功能。

直接在脚本中添加认证代码可能会影响调试，并且这些代码有时候必须在加密环境下才能运行。所以推荐的方式是使用插件扩展认证方式 。插件是一个普通的脚本，等价于把认证代码单独提取出来，然后在加密的时候注入到被加密的脚本中去，这样不需要对原来的脚本进行任何修改。更多详细的内容参考如何处理插件 。

这里有一些插件的例子可以参考

https://github.com/dashingsoft/pyarmor/tree/master/plugins

2.5 加密单个模块

如果只需要单独加密一个模块，使用选项 `--exact`:

```
pyarmor obfuscate --exact foo.py
```

这样，就只有 `foo.py` 被加密，导入这个加密模块:

```
cd dist
python -c "import foo"
```

2.6 加密整个 Python 包

加密整个 Python 包使用下面的命令:

```
pyarmor obfuscate --recursive --output dist/mypkg mykpg/__init__.py
```

使用这个加密的包:

```
cd dist
python -c "import mypkg"
```

2.7 打包加密脚本

命令 `pack` 用来打包并加密脚本

首先需要安装 *PyInstaller*:

```
pip install pyinstaller
```

然后运行下面的命令:

```
pyarmor pack myscript.py
```

PyArmor 通过以下的步骤将所有需要的文件打包成为一个独立可运行的安装包:

- 执行 `pyarmor obfuscate` 加密脚本 `myscript.py` 和同目录下的所有其他脚本
- 执行 `pyinstaller myscipt.py` 创建 `myscript.spec`
- 修改 `myscript.spec`, 把原来的脚本替换成为加密后的脚本
- 再次执行 `pyinstaller myscript.spec` ，生成最终的安装包

输出的文件在目录 `dist/myscript` ，这里面包含了脱离 Python 环境可以运行的所有文件。

运行打包好的可执行文件:

```
dist/myscript/myscript
```

检查脚本是否加密。如果加密，下面的第二条命令应该执行失败:

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

为加密脚本设置有效期:

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

dist/myscript/myscript
```

对于复杂的应用，例如传入额外的参数到 *Pyinstaller* 等，请参考命令*pack* 和如何打包加密脚本

2.8 进一步提高加密脚本的安全性

下面这些 PyArmor 的特征可以进一步提高加密脚本的安全性:

1. 尽可能的使用超级模式加密脚本 ，如果使用的平台或者 Python 版本不被支持，那么，启用高级模式
2. 如果情况允许的话，尝试绑定加密脚本到固定的 *Python* 解释器 ，通常情况下，超级模式 不需要这种额外的保护。
3. 定制交叉保护脚本 来确保动态库没有被修改
4. 使用相应约束模式 来加密脚本

5. 使用更高安全级别的代码加密模式 *–obf-code=2*
6. 使用插件来进一步提高安全性

关于加密脚本的安全性的说明，请参考*PyArmor 的安全性*

CHAPTER 3

高级用法

3.1 使用超级模式加密脚本

超级模式 是 6.2.0 的新功能，超级模式只需要一个运行辅助文件，没有了所谓的引导代码，所有的加密脚本都长得一个样，大大简化了加密脚本的使用方法，并且极大的提高了安全性，唯一的缺点是目前支持的平台和 Python 版本还不完整。

超级模式在命令行使用 `--advanced 2` 来指定，例如:

```
pyarmor obfuscate --advanced 2 foo.py
```

而发布加密脚本的时候，只需要把扩展模块 `pytransform` ，放在任意的 Python 路径下面，加密后的脚本就可以正常运行。

如果需要对加密脚本进行限制，那么使用首先生成一个包含约束的 `license.lic` ，例如，绑定到网卡:

```
pyarmor licenses --bind-mac xx:xx:xx:xx regcode-01
```

然后在加密脚本的时候使用 `--with-license` 指定这个文件，例如:

```
pyarmor obufscate --with-license licenses/regcode-01/license.lic \
                  --advanced 2 foo.py
```

这样就可以把指定的许可文件嵌入到扩展模块里面，如果不想把许可文件嵌入到扩展模块里面，而是使用外部 `license.lic` ，这样可以方便替换许可文件，那么在加密的时候需要指定 `--with-license outer` ，例如:

```
pyarmor obfuscate --with-license outer --advanced 2 foo.py
```

这样，运行加密脚本的时候就会在外部查找许可文件，查找外部 license.lic 的顺序

1. 如果设定了环境变量 PYARMOR_LICENSE ，直接使用这里指定的文件名
2. 如果没有，那么查找当前路径下面的 license.lic
3. 如果还没有，查找和扩展模块 pytransform 相同路径下面的 license.lic
4. 没有找到就报错

3.2 加密和使用多个包

假定有三个包 *pkg1*, *pkg2*, *pkg2* 需要加密，使用公共的运行辅助文件，然后可以从其他脚本导入这些加密的包。

首先切换到工作路径，创建三个工程:

```
mkdir build
cd build

pyarmor init --src /path/to/pkg1 --entry __init__.py pkg1
pyarmor init --src /path/to/pkg2 --entry __init__.py pkg2
pyarmor init --src /path/to/pkg3 --entry __init__.py pkg3
```

生成公共的运行辅助包 ，保存在 *dist* 目录下面:

```
pyarmor build --output dist --only-runtime pkg1
```

分别加密三个包，也保存到 *dist* 下面:

```
pyarmor build --output dist --no-runtime pkg1
pyarmor build --output dist --no-runtime pkg2
pyarmor build --output dist --no-runtime pkg3
```

查看并使用加密的包:

```
ls dist/

cd dist
python -c 'import pkg1
import pkg2
import pkg3'
```

注解: 输出目录 *dist* 下面的运行辅助包 `pytransform` 可以被拷贝到任何的 Python 可以导入的目录下面。

从 v5.7.2 之后，运行辅助包也可以使用命令*runtime* 单独生成:

```
pyarmor runtime
```

3.3 如何加密能和其他加密包共存的包

注解: New in v5.8.7

假设有两个包分别被两个不同的开发人员进行加密，那么这两个包能不能在同一个 Python 解释器中运行呢?

如果这两个包都是被试用版加密，那么没有问题。但是如果任何一个是被注册版本的 PyArmor 加密，那么答案是否定的。

从 v5.8.7 开始，使用选项 `--enable-suffix` 来加密时，运行辅助包 的名称不在固定为 `pytransform` ，而是会有一个唯一性的后缀，这样不同的加密包就可以实现共存。例如:

```
pyarmor obfuscate --enable-suffix foo.py
```

加密后的输出目录结构如下:

```
dist/
    foo.py
    pytransform_vax_000001/
        __init__.py
        ...
```

其中后缀 `_vax_000001` 是基于 PyArmor 的注册码生成的具有唯一性的字符串。

对于使用工程加密的方式，则需要使用命令*config* 来设置:

```
pyarmor config --enable-suffix 1
pyarmor build -B
```

使用下面的方式可以禁用后缀模式:

```
pyarmor config --enable-suffix 0
pyarmor build -B
```

3.4 跨平台发布加密脚本

因为加密脚本的运行文件中有平台相关的动态库，所以跨平台发布需要指定目标平台。

首先使用命令*download* 列出所有支持的标准平台名称:

```
pyarmor download
pyarmor download --help-platform
```

使用选项 `--list` 会显示详细的动态库特征信息:

```
pyarmor download --list
pyarmor download --list windows
pyarmor download --list windows.x86_64
```

然后在加密脚本的时候指定目标平台名称:

```
pyarmor obfuscate --platform linux.armv7 foo.py

# For project
pyarmor build --platform linux.armv7
```

3.4.1 使用不同特征的动态库

同一个平台下面，包含多个可用的动态库，不同的动态库具备不同的特征。例如，在相同的平台 `windwos.x86_64` 下面，有两个动态库 `windows.x86_64.0` 和 `windows.x86_64.7` ，其中最后的数字表示动态库的特征:

- 0: 没有反调试、动态代码、高级模式等特征，速度最快
- 7: 包含反调试、动态代码、高级模式等特征，安全性最高

如果不指定特征码，默认是选择安全性最高的动态库，有些不平台只支持部分特征的动态库。在跨平台加密的时候也可以指定平台的特征，例如:

```
pyarmor obfuscate --platform linux.x86_64.7 foo.py
```

需要注意的是不同特征的动态库相互并不兼容，例如，默认情况下 Windows 平台下使用的是高特征的动态库，当使用下面的命令在 Windows 平台下面加密低特征的动态库的时候:

```
pyarmor obfuscate --platform linux.arm.0 foo.py
```

在控制台能看到 PyArmor 会自动重启，使用低特征的库加载之后才对脚本进行加密。

也可以使用环境变量 `PYARMOR_PLATFORM` 直接设置当前平台使用的动态库的特征。例如:

```
PYARMOR_PLATFORM=windows.x86_64.0 pyarmor obfuscate --platform linux.arm.0 foo.py

# 在 Windows 平台
set PYARMOR_PLATFORM=windows.x86_64.0
pyarmor obfuscate --platform linux.arm.0 foo.py
set PYARMOR_PLATFORM=
```

3.4.2 让加密脚本可以在多个平台运行

从 v5.7.5 版本开始，平台名称已经标准化，所有可用名称在这里标准平台名称 ，并且支持运行加密脚本在多个平台。

为了支持加密脚本在多个平台运行，需要把相关平台的动态库都添加到运行辅助包 中，这样就可以在这些平台正常运行加密脚本。例如，使用下面的命令可以加密一个可运行于 Windows/Linux/MacOS 下面的脚本:

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

也可以使用命令*runtime* 单独生成可以运行多个平台的 运行辅助包，这样就不需要每次加密的时候都生成这些辅助文件。例如:

```
pyarmor runtime --platform windows.x86_64,linux.x86_64,darwin.x86_64
pyarmor obfuscate --no-runtime --recursive \
                  --platform windows.x86_64,linux.x86_64,darwin.x86_64 \
                  foo.py
```

即便使用了 `--no-runtime` ，在加密脚本的时候也需要指定运行的平台，因为加密脚本会在启动的时候检查动态库，只有指定的动态库才能通过检查。如果指定了选项 `--no-cross-protection` ，加密脚本就不会在检查动态库，那么加密的时候就不需要指定运行平台，例如:

```
pyarmor obfuscate --no-runtime --recursive --no-cross-protection foo.py
```

注解: 如果指定了平台的特征，例如 `windows.x86_64.7` ，那么需要注意的是所有的平台必须具备相同的特征，不同特征的动态库是无法共存在同一个包里面的。

注解: 如果加密后的脚本无法运行，可以尝试使用下面的命令升级已经下载的动态库:

```
pyarmor download --update
```

也可以直接删除缓存的动态库目录，默认是 `$HOME/.pyarmor/platforms`

3.5 使用不同版本 Python 加密脚本

如果装了多个版本的 Python ，那么使用 *pip* 安装的 *pyarmor* 使用的是默认的 Python 版本。如果需要使用其他版本的 Python 来加密脚本，需要显示指定 Python 解释器。

例如，首先找到 `pyarmor.py` 的位置:

```
find /usr/local/lib -name pyarmor.py
```

通常在大多数 linux 系统，它会在 */usr/local/lib/python2.7/dist-packages/pyarmor*

然后使用下面的方式运行:

```
/usr/bin/python3.6 /usr/local/lib/python2.7/dist-packages/pyarmor/pyarmor.py
```

也可以创建一个便捷脚本 */usr/local/bin/pyarmor3* ，内容如下:

```
/usr/bin/python3.6 /usr/local/lib/python2.7/dist-packages/pyarmor/pyarmor.py "$*"
```

赋予其执行权限:

```
chmod +x /usr/local/bin/pyarmor3
```

然后就可以直接使用 *pyarmor3*

在 Windows 下面就需要创建一个批处理 *pyarmor3.bat* ，内容如下:

```
C:\Python36\python C:\Python27\Lib\site-packages\pyarmor\pyarmor.py %*
```

3.6 在没有加密的脚本中运行引导代码

有时候需要在普通脚本中运行引导代码，这样就可以正常的导入其他加密脚本，而不需要在每一个加密脚本中到插入引导代码。在 v5.7.0 之前，可以直接把两行*引导代码* 插入到普通脚本中，但是之后的版本，为了提高安全性，已经不允许在普通脚本中直接运行引导代码，必须使用一种折衷的方式。

首先使用命令*runtime* 生成一个引导辅助包 `pytransform_bootstrap`:

```
pyarmor runtime -i
```

然后把生成的辅助包拷贝到脚本所在的目录:

```
mv dist/pytransform_bootstrap /path/to/script
```

也可以把这个包拷贝到 Python 的库目录，例如:

```
mv dist/pytransform_bootstrap /usr/lib/python3.5/ (For Linux)
mv dist/pytransform_bootstrap C:/Python35/Lib/ (For Windows)
```

最后修改普通脚本，在其中插入一条语句:

```
import pytransform_bootstrap
```

这样就可以在其后导入并使用其他加密模块。

注解: 在 v5.8.1 之前，需要使用人工的方式生成这个引导辅助包:

```
echo "" > __init__.py
pyarmor obfuscate -O dist/pytransform_bootstrap --exact __init__.py
```

下面是一个实际的例子，运行加密脚本的单元测试用例。

3.6.1 运行加密脚本的单元测试

因为大部分的加密脚本都没有引导代码，所以在运行单元测试之前，必须首先运行引导代码。

假设单元测试脚本为 `/path/to/tests/test_foo.py`，那么首先修改这个单元测试脚本，参考在没有加密的脚本中运行引导代码

这样，这个测试脚本就可以导入被加密的模块进行正常的测试:

```
cd /path/to/tests
python test_foo.py
```

还有一种方式就是直接修改系统包 `unittest`，首先要把引导辅助包拷贝到 Python 的系统库路径下面，参考在没有加密的脚本中运行引导代码

然后在修改 `/path/to/unittest/__init__.py`，插入语句:

```
import pytransform_bootstrap
```

这样，所有的单元测试脚本就都可以直接来测试加密后的模块了。如果有很多单元测试脚本，这种方式会更方便一些。

3.7 让 Python 自动识别加密脚本

下面有几种情况可能会需要让 Python 自动识别加密脚本:

- 几乎所有的脚本都会被作为主脚本来运行
- 在加密脚本中使用模块 *multiprocessing* 创建新进程
- 使用到 *Popen* 或者 *os.exec* 等调用加密后的脚本
- 其他任何需要在很多脚本里面插入引导代码的情况

一种解决方案就是为每一个相关的加密脚本添加引导代码，但是这会有些麻烦。另外一种比较简单的解决方案就是让 Python 能够自动识别加密脚本，这样任何一个加密脚本不需要引导代码就可以正常运行。

下面是基本操作步骤:

1. 首先生成引导辅助包 `pytransform_bootstrap`:

   ```
   pyarmor runtime -i
   ```

   在 v5.8.1 之前，需要通过加密一个空脚本的方式生成引导辅助包:

   ```
   echo "" > __init__.py
   pyarmor obfuscate -O dist/pytransform_bootstrap --exact __init__.py
   ```

2. 其次需要建立一个运行加密脚本的虚拟环境，把引导辅助包拷贝到虚拟环境的库路径，例如:

   ```
   # For windows
   mv dist/pytransform_bootstrap venv/Lib/

   # For linux
   mv dist/pytransform_bootstrap venv/lib/python3.5/
   ```

4. 最后修改 `venv/lib/pythonX.Y/site.py` 或者 `venv/lib/site.py` , 插入一条导入语句:

   ```
   import pytransform_bootstrap

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

也可以把这行代码添加到 `main` 函数里面，总之，只要能得到执行就可以。

这样就可以使用虚拟环境中 `python` 直接运行加密脚本了。这主要使用到了 Python 在启动过程中默认会自动导入模块 `site` 的特性来实现，参考

https://docs.python.org/3/library/site.html

注解： 这里配置的是运行加密脚本的环境，在这里 *pyarmor* 是无法运行的。

注解： 在 v5.7.0 之前，需要根据运行辅助文件 人工创建引导辅助包

3.8 使用不同的模式来加密脚本

高级模式 是从 PyArmor 5.5.0 引入的新特性，默认情况下是没有启用的。如果需要使用高级模式来加密脚本，额外指定选项 `--advanced`:

```
pyarmor obfuscate --advanced 1 foo.py
```

从 PyArmor 5.2 开始, 约束模式 是默认设置。

使用选项 `--restrict` 指定其他约束模式，例如:

```
pyarmor obfuscate --restrict=2 foo.py
pyarmor obfuscate --restrict=3 foo.py

# For project
cd /path/to/project
pyarmor config --restrict 4
pyarmor build -B
```

如果需要禁用各种约束，那么使用下面的命令加密脚本:

```
pyarmor obfuscate --restrict=0 foo.py

# For project
pyarmor config --restrict=0
pyarmor build -B
```

指定代码加密模式, 代码包裹模式, 模块加密模式 需要使用工程 来加密脚本，直接使用命令*obfuscate* 无法改变这些加密模式。例如:

```
pyarmor init --src=src --entry=main.py .
pyarmor config --obf-mod=1 --obf-code=1 --wrap-mode=0
pyarmor build
```

3.9 使用插件扩展认证方式

PyArmor 可以通过插件来扩展加密脚本的认证方式，例如检查网络时间而不是本地时间来校验有效期。

首先定义插件文件 check_ntp_time.py 插件的主函数是 *check_ntp_time* ，另外一个重要的函数是 *__get_license_data* ，用来从加密脚本的 *license.lic* 许可文件中获取自定义的数据信息。

然后在主脚本 foo.py 插入下列两行注释:

```
# {PyArmor Plugins}
# PyArmor Plugin: check_ntp_time()
```

执行下面的命令进行加密:

```
pyarmor obfuscate --plugin check_ntp_time foo.py
```

插件对应的文件一般存放在当前目录，如果存放在其他目录的话，可以指定绝对路径，例如:

```
pyarmor obfuscate --plugin /usr/share/pyarmor/check_ntp_time foo.py
```

最后为加密脚本生成许可文件，使用 `-x` 把自定义的有效期存储到认证文件，这样就可以通过插件脚本中定义的函数 *__get_license_data* 来读取这个数据:

```
pyarmor licenses -x 20190501 rcode-001
cp licenses/rcode-001/license.lic dist/
```

更多插件实例，参考 https://github.com/dashingsoft/pyarmor/tree/master/plugins

关于插件的工作原理，参考如何处理插件

3.10 打包加密脚本成为一个单独的可执行文件

使用下面的命令可以把脚本 *foo.py* 加密之后并打包成为一个单独的可执行文件:

```
pyarmor pack -e " --onefile" foo.py
```

其中 `--onefile` 是 PyInstaller 的选项，使用 `-e` 可以传递任何 Pyinstaller 支持的选项，例如，指定可执行文件的图标:

```
pyarmor pack -e " --onefile --icon logo.ico" foo.py
```

如果不想把加密脚本的许可文件 license.lic 打包到可执行文件，而是和可执行文件放在一起，这样方便为不同的用户生成不同的许可文件。那么需要使用 PyInstaller 提供的 --runtime-hook 功能在加密脚本运行之前把许可文件拷贝到指定目录，下面是具体的操作步骤：

1. 新建一个文件 *copy_license.py*:

   ```
   import sys
   from os.path import join, dirname
   with open(join(dirname(sys.executable), 'license.lic'), 'rb') as src:
       with open(join(sys._MEIPASS, 'license.lic'), 'wb') as dst:
           dst.write(src.read())
   ```

2. 运行下面的命令打包加密脚本:

   ```
   pyarmor pack --clean --without-license -x " --exclude copy_license.py" \
           -e " --onefile --icon logo.ico --runtime-hook copy_license.py" foo.py
   ```

   选项 --without-license 告诉*pack* 不要把加密脚本的许可文件打包进去，使用 PyInstaller 的选项 --runtime-hook 可以让打包好的可执行文件，在启动的时候首先去调用 copy_licesen.py ，把许可文件拷贝到相应的目录。

   命令执行成功之后，会生成一个打包好的文件 dist/foo.exe

   尝试运行这个可执行文件，应该会报错。

3. 使用命令*licenses* 生成新的许可文件，并拷贝到 dist/ 下面:

   ```
   pyarmor licenses -e 2020-01-01 tom
   cp license/tom/license.lic dist/
   ```

4. 这时候在双击运行 dist/foo.exe ，在 2020-01-01 之前应该就可以正常运行

3.11 使用定制的.spec 文件打包加密脚本

如果已经有写好的 *.spec* 文件能够成功打包，例如:

```
pyinstaller myscript.spec
```

那么，现在就可以直接使用*pack* 加密并打包 *myscript.py*:

```
pyarmor pack -s myscript.spec myscript.py
```

如果打包的时候出现下面的错误:

```
Unsupport .spec file, no XXX found
```

那么请检查 *.spec* 文件，确保下列两行在文件中存在（没有缩进）：

```
a = Analysis(...

pyz = PYZ(...
```

并且在创建 *Analysis* 对象的时候下列三个命名参数也存在，例如:

```
a = Analysis(
    ...
    pathex=...,
    hiddenimports=...,
    hookspath=...,
    ...
)
```

PyArmor 会自动把需要的相关参数添加到这些行。但是在 v5.9.6 之前，需要人工进行添加:

- 增加模块 `pytransform` 到 *hiddenimports*
- 增加额外的路径 `DISTPATH/obf/temp` 到 *pathex* 和 *hookspath*

修改后的文件大概会是这样子的:

```
a = Analysis(['myscript.py'],
             pathex=[os.path.join(DISTPATH, 'obf', 'temp'), ...],
             binaries=[],
             datas=[],
             hiddenimports=['pytransform', ...],
             hookspath=[os.path.join(DISTPATH, 'obf', 'temp'), ...],
             ...
```

注解: 这个功能是在 v5.8.0 新增加的

在 v5.8.2 之前，额外的路径是 `DISTPATH/obf` 而不是 `DISTPATH/obf/temp`

3.12 使用约束模式增加加密脚本安全性

默认约束模式仅限制不能修改加密脚本，为了提高安全性，可以使用约束模式 2 来加密 Python 应用程序，例如:

```
pyarmor obfuscate --restrict 2 foo.py
```

约束模式 2 不允许从没有加密的脚本中导入加密的脚本，从而更高程度的保护了加密脚本的安全性。

如果对安全性要求更高，可以使用约束模式 3 ，例如:

```
pyarmor obfuscate --restrict 3 foo.py
```

约束模式 3 会检查每一个加密函数的调用，不允许加密的函数被非加密的脚本调用。

上述两种模式并不适用于 Python 包的加密，因为对于 Python 包来说，必须允许加密的脚本被其他非加密的脚本导入和调用。为了提高 Python 包的安全性，可以采取下面的方案:

- 把需要供外部使用的函数集中到包的某一个或者几个文件
- 使用约束模式 1 加密这些需要被外部调用的文件
- 使用约束模式 4 加密其他的脚本文件

例如:

```
cd /path/to/mypkg
pyarmor obfuscate --exact __init__.py exported_func.py
pyarmor obfuscate --restrict 4 --recursive \
        --exclude __init__.py --exclude exported_func.py .
```

关于约束模式的详细说明，请参考约束模式

3.13 使用插件来进一步提高安全性

使用插件可以自由的加入自己的私有检查代码到被加密后的脚本，但是又不影响原来脚本的执行。因为这些代码一般情况下在没有加密的脚本中是无法正常运行的，如果直接把这些代码写到脚本里，原来的脚本就无法正常调试。

通过增加私有的检查代码，可以很大程度上提高加密脚本的安全性，因为没有人知道你的检查逻辑，并且你可以随时更改这些检查逻辑。

3.13.1 使用内联插件检查动态库没有被修改

虽然 *PyArmor* 自身提供了交叉保护功能来保护动态库，但是还可以增加自己的私有检查点。下面这个例子使用内联插件来再次检查动态库的修改时间，首先在 `main.py` 中增加下面的注释

```
# PyArmor Plugin: import os
# PyArmor Plugin: libname = os.path.join( os.path.dirname( __file__ ), '_pytransform.so'␣
↪)
```

(下页继续)

(续上页)

```
# PyArmor Plugin: if not os.stat( libname ).st_mtime_ns == 102839284238:
# PyArmor Plugin:     raise RuntimeError('Invalid Library')
```

然后调用下面的加密命令来启用这个内联插件:

```
pyarmor obfuscate --plugin on main.py
```

这样，加密脚本在运行的时候就会首先运行下面的插件代码

```
import os
libname = os.path.join( os.path.dirname( __file__ ), '_pytransform.so' )
if not os.stat( libname ).st_mtime_ns == 102839284238:
    raise RuntimeError('Invalid Library')
```

3.13.2 使用插件检查被调用的函数是否经过加密

假设主脚本为 *main.py*, 需要调用模块 *foo.py* 里面的方法 *connect*, 并且需要传递敏感数据作为参数。脚本的相关代码如下

```
#
# This is main.py
#

import foo

def start_server():
    foo.connect('root', 'root password')
    foo.connect2('user', 'user password')

#
# This is foo.py
#

def connect(username, password):
    mysql.dbconnect(username, password)

def connect2(username, password):
    db2.dbconnect(username, password)
```

虽然两个脚本都已经被加密，但是用户可以自己写一个简单的脚本来代替加密的 *foo.py*

```
def connect(username, password):
    print('password is %s', password)
```

然后调用加密的主脚本 *main.py* ，虽然功能不能正常完成，但是敏感数据却被泄露。

为了避免这种情况发生，需要使用插件为函数 *stare_server* 提供修饰函数，在每次运行 *start_server* 之前，检查被调用的函数是否是自己定义的函数，如果不是，直接抛出异常。

从 v6.0.2 开始，运行辅助包 pytransform 提供一个修饰函数 assert_armored ，可以用来检查参数列表中函数是否被加密。现在，编辑脚本 *main.py* ，如下所示的方式增加两个内联插件桩

```
import foo

# PyArmor Plugin: from pytransform import assert_armored

# PyArmor Plugin: @assert_armored(foo.connect, foo.connect2)
def start_server():
    foo.connect('root', 'root password')
```

然后在启用插件模式加密脚本:

```
pyarmor obfuscate --plugin on main.py
```

加密后的脚本相当于下面的代码

```
import foo

from pytransform import assert_armored

@assert_armored(foo.connect, foo.connect2)
def start_server():
    foo.connect('root', 'root password')
```

这样，在调用 start_server 之前，修饰函数 assert_assert 会检查两个 connect 函数，如果它们没有被加密，就会抛出异常。

为了进一步提高安全性，可以把使用外部插件脚本的方式，这样插件函数本身就不需要从外部导入。首先在当前目录下创建一个插件脚本 asser_armored.py

```
from pytransform import _pytransform, PYFUNCTYPE, py_object

def assert_armored(*names):
    prototype = PYFUNCTYPE(py_object, py_object)
    dlfunc = prototype(('assert_armored', _pytransform))
```

(下页继续)

(续上页)

```
    def wrapper(func):
        def _execute(*args, **kwargs):

            # Call check point provide by PyArmor
            dlfunc(names)

            # Add your private check code
            for s in names:
                if s.__name__ == 'connect':
                    if s.__code__.co_code[10:12] != b'\x90\xA2':
                        raise RuntimeError('Access violate')

            return func(*args, **kwargs)
        return _execute
    return wrapper
```

然后修改 *main.py*, 增加插件定义桩和插件调用桩，修改后的代码如下

```
import foo

# {PyArmor Plugins}

# PyArmor Plugin: @assert_armored(foo.connect, foo.connect2)
def start_server():
    foo.connect('root', 'root password')
```

在加密脚本的时候指定插件名称:

```
pyarmor obfuscate --plugin assert_armored main.py
```

注解: 从 v6.2.0 开始，使用超级模式 加密的脚本不需要也无法使用外部插件脚本，直接使用扩展模块 pytransform 提供的修饰函数 assert_armored 即可。

3.14 在 Python 脚本内部调用 *pyarmor*

在 Python 脚本内部，也可以直接调用 *pyarmor* ，不需要使用 *os.exec* 或者 *subprocess.Popen* 等命令行的方式。例如

```
from pyarmor.pyarmor import main as call_pyarmor
call_pyarmor(['obfuscate', '--recursive', '--output', 'dist', 'foo.py'])
```

使用选项 `--silent` 可以不显示所有输出

```
from pyarmor.pyarmor import main as call_pyarmor
call_pyarmor(['--silent', 'obfuscate', '--recursive', '--output', 'dist', 'foo.py'])
```

从 v5.7.3 开始，如果以这种方式调用 *pyarmor* 出现了错误，会抛出异常，而不是调用 *sys.exit* 直接退出。

3.15 运行加密脚本的时候周期性的检查许可文件

通常情况下只有运行脚本开始启动的时候会检查许可文件，一旦运行之后，就不在检查。从 v5.9.3 之后，实现了在脚本运行过程中对许可文件进行周期性（每小时一次）检查的功能。所需要做的是使用选项 `--enable-period-mode` 生成一个新的许可文件，覆盖默认的许可文件即可。例如:

```
pyarmor obfuscate foo.py
pyarmor licenses --enable-period-mode code-001
cp licenses/code-001/license.lic ./dist
```

3.16 使用 Nuitka 发布加密脚本

因为加密后的脚本也是正常的 Python 脚本（外加运行辅助包 `pyrtransform` ），所以完全可以使用 Nuitka 对加密脚本进行处理，就像正常的 Python 脚本一样。但是加密脚本的时候，需要指定额外的选项 `--restrict 0` 和 `--no-cross-protection` ，否则加密脚本可能会报错。例如，首先加密脚本 `foo.py`:

```
pyarmor obfuscate --restrict 0 --no-cross-protection foo.py
```

然后使用 Nuitka 把加密后的脚本转换成为可执行的文件:

```
cd ./dist
python -m nuitka --include-package pytransform foo.py
./foo.bin
```

需要注意的是一旦脚本被加密之后，Nuitka 无法自动找到该模块导入的所有相关模块（包）。为了解决这个问题，首先调用 Nuitka 转换没有加密的脚本，生成相应的 `.pyi` 文件，然后把这个文件拷贝到加密脚本所在的目录，这样就可以正常的转换加密后的脚本。例如:

```
# 生成 "mymodule.pyi"
python -m nuitka --module mymodule.py
```

(下页继续)

(续上页)

```
pyarmor obfuscate --restrict 0 --no-bootstrap mymodule.py
cp mymodule.pyi dist/

cd dist/
python -m nuitka --module mymodule.py
```

但是这种方式可能基本没有使用 Nuitka 转换脚本的功能，所以在性能上提升应该不会太明显。

注解: 只要 Nuitka 还连接到 CPython 的库来执行转换后的 C 代码，pyarmor 就应该可以和 Nuitka 共存。但是 Nuitka 的官网上有一段对未来特征的描述:

```
It will do this - where possible - without accessing libpython but in C
with its native data types.
```

也就是说，Nuitka 将来不需要 CPython 库，那么在这种情况下，pyarmor 加密的后的脚本将无法在 Nuitka 下面执行。

3.17 使用 Cython 发布加密脚本

下面是一个简单的例子，来说明如何把一个脚本 `foo.py` 使用 PyArmor 进行加密，然后在使用 Cython 把它转换成为扩展模块，脚本的内容如下:

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

首先加密脚本，使用了一些额外的选项，这些选项的作用可参考命令*obfuscate* 中的说明，加密后的脚本和运行辅助文件存放在目录 `dist` 下面:

```
pyarmor obfuscate --package-runtime 0 --no-cross-protection --restrict 0 foo.py
```

接下来使用 *cythonize* 把 *foo.py* 和 *pytransform.py* 转换成为 `.c` 文件:

```
cd dist
cythonize -3 -k --lenient foo.py pytransform.py
```

注意需要使用选项 `-k` 和 `--lenient` ，否则会报错:

```
undeclared name not builtin: __pyarmor__
```

然后把 *foo.c* and *pytransform.c* 编译成为扩展模块，在 MacOS 平台下，直接执行下面的命令就可以；如果是 Linux p 平台，需要把额外的编译选项 `-fPIC` 加入到命令行:

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

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

最后测试一下这些扩展模块，把所有 *.py* 文件删除了，然后导入加密后的脚本:

```
mv foo.py pytransform.py /tmp
python -c 'import foo'
```

像预想的那样会在控制台打印出 *Hello Cython*

3.18 使用 PyUpdater 发布加密脚本

PyArmor 可以使用下面的方式和 PyUpdater 一起工作。假设脚本为 *foo.py*:

1. 首先使用 PyUpdater 生成 *foo.spec*
2. 其次使用 PyArmor 生成 *foo-patched.spec* ，选项 `--debug` 可以保留这个中间文件在命令执行完成之后不被删除:

   ```
   pyarmor pack --debug -s foo.spec foo.py

   # 如果运行可执行文件的时候抛出保护异常，尝试使用下面的命令禁用加密脚本的相关约束
   pyarmor pack --debug -s foo.spec -x " --restrict 0 --no-cross-protection" foo.py
   ```

然后 PyUpdater 就可以使用这个 *foo-patched.spec* 进行构建。

当脚本修改之后，只需要使用命令*obfuscate* 重新加密脚本，加密脚本需要的全部选项可以在上面*pack* 命令执行后的输出中找到。

如果上面生成的包在执行的时候存在问题，可以把 PyArmor 打好的包压缩成 *.zip* 然后放到 `/pyu-data/new` 下面，在这里签名，处理和上传更新包。

更多信息可以查看命令*pack* 和*使用定制的.spec 文件打包加密脚本*

3.19 绑定加密脚本到固定的 Python 解释器

为了提高加密脚本的安全性，也可以把加密脚本绑定到某一个固定的 Python 解释器，如果用户修改了 Python 解释器，那么加密脚本就会直接退出。

当然，在绑定之后，加密脚本的运行环境会有更多约束。有可能目标机器上 Python 的版本号和加密时候使用的 Python 版本号完全一致才能够运行加密脚本。

如果你使用命令*obfuscate* 来加密脚本，那么在加密之后，使用下面的命令生成一个新的许可文件，覆盖原来的许可文件。例如:

```
pyarmor licenses --fixed 1 -O dist/license.lic
```

这样，加密脚本在运行的时候就会检查当前 Python 的动态库文件，根据平台的不同，它可能是 pythonXY.dll，libpythonXY.so 或者 libpythonXY.dylib。一旦发现运行环境的动态库被修改（和加密时候的不一样），加密脚本就会直接退出。

如果你使用的是工程来加密脚本，那么首先生成一个绑定到固定 Python 解释器的许可证文件:

```
cd /path/to/project
pyarmor licenses --fixed 1
```

默认情况下，它会存放在 *licenses/pyarmor/license.lic* ，然后修改工程配置，使用这个许可证文件:

```
pyarmor config --license=licenses/pyarmor/license.lic
```

如果是跨平台发布加密脚本，那么还需要额外的工作，要得到目标平台的 Python 动态库的特征码。首先在目标平台下按照下面的内容创建一个 Python 脚本，然后使用相应的 Python 解释器执行这个脚本

```
import sys

from ctypes import CFUNCTYPE, cdll, pythonapi, string_at, c_void_p, c_char_p


def get_bind_key():
    c = cdll.LoadLibrary(None)

    if sys.platform.startswith('win'):
        from ctypes import windll
        dlsym = windll.kernel32.GetProcAddressA
    else:
        prototype = CFUNCTYPE(c_void_p, c_void_p, c_char_p)
        dlsym = prototype(('dlsym', c))

    refunc1 = dlsym(pythonapi._handle, 'PyEval_EvalCode')
    refunc2 = dlsym(pythonapi._handle, 'PyEval_GetFrame')

    size = refunc2 - refunc1
    code = string_at(refunc1, size)

    checksum = 0
```

(下页继续)

(续上页)

```
    for c in code:
        checksum += ord(c)
    print('Get bind key: %s' % checksum)


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

运行之后会打印出需要绑定的特征码 *xxxxxx* ，然后使用这个特征码生成相应的新的许可证:

```
pyarmor licenses --fixed xxxxxx -O dist/license.lic
```

也可以绑定许可文件到多个 Python 解释器，把需要绑定的特征码使用逗号分开即可:

```
pyarmor licenses --fixed 1,key2,key3 -O dist/license.lic
pyarmor licenses --fixed key1,key2,key3 -O dist/license.lic
```

注意特征码 *1* 可以用来表示当前 Python 解释器。

3.20 定制交叉保护脚本

为了保护 PyArmor 核心动态库在运行加密脚本的时候不会被别人修改，默认情况下加密脚本的时候会在主脚本里面插入保护代码，参考对主脚本的特殊处理 。但是这种公开的保护代码总可能会给别人机会来绕过这种保护，所以为了提高安全性，最好的办法是修改默认的保护代码，使用自己私有的逻辑来保护动态库，这样可以极大的提高安全性。

从 v6.2.0 开始，可以使用命令*runtime* 来生成默认的交叉保护代码的脚本，你可以参考这个脚本来进行修改，使用自己的方法来检查动态库，当然更好的办法是完全写自己的保护代码，只要能达到在 Python 脚本里面检查动态库 *pytransform* 没有被修改的目的就可以。

首先使用下面的命令生成默认的交叉保护脚本 `build/pytransform_protection.py`:

```
pyarmor runtime --super-mode --output build
```

然后修改这个生成的脚本，并在加密的时候使用选项 `--cross-protection` 来指定这个脚本就可以了。例如:

```
pyarmor obfuscate --cross-protection build/pytransform_protection.py \
                  --advanced 2 foo.py
```

需要注意的是超级模式和其他任何模式使用的交叉保护脚本并不一样，所以如果不是使用超级模式进行加密，生成默认脚本的时候就不需要额外选项，例如:

```
pyarmor runtime --output build
```

注解: 使用 `--advanced 1` 加密并不是超级模式，只有 `--advanced 2` 才是超级模式

3.21 如何把许可文件 license.lic 存放到任意位置

默认情况下运行加密脚本需要的许可文件 *license.lic* 是和模块 `pytransform` 存放在一起，但是也可以把这个文件放在任意位置，然后在原来的位置创建一个符号链接，指向真正的文件就可以。

在 linux 下面，使用下面的命令把许可文件放到 `/opt/my_app`:

```
cd /path/to/obfuscated/pytransform
ln -s /opt/my_app/license.lic license.lic
```

在 Windows 下面，使用下面的命令把许可文件放到 `C:/Users/Jondy/my_app`:

```
cd /path/to/obfuscated/pytransform
mklink license.lic C:/Users/Jondy/my_app/license.lic
```

在发布加密包的时候，只要能够设法在安装完成之后，执行下面的脚本就可以放置加密脚本到任意指定的位置:

```
import os

def make_link_to_license_file(package_path, target_license="/opt/mypkg/license.lic"):
    license_file = os.path.join(package_path, 'pytransform', 'license.lic')
    if os.path.exists(license_file):
        os.rename(license_file, target_license)
    os.symlink(target_license, license_file)
```

CHAPTER 4

使用实例

下面是一些使用 PyArmor 的实例。

4.1 加密并打包 PyQt 应用

文字统计工具 easy-han 使用 PyQt 开发，主要文件如下:

```
config.json

main.py
ui_main.py
readers/
    __init__.py
    msexcel.py

tests/
vnev/py36
```

加密打包脚本如下:

```
cd /path/to/src
pyarmor pack --name easy-han \
             -e " --hidden-import comtypes --add-data 'config.json;.'" \
```

(下页继续)

(续上页)

```
            -x " --exclude vnev --exclude tests" main.py
cd dist/easy-han
./easy-han
```

使用 -e 传入额外的参数去运行 PyInstaller ，要确认 PyInstaller 使用这些选项可以正常打包:

```
cd /path/to/src
pyinstaller --name easy-han --hidden-import comtypes --add-data 'config.json;.' main.py

cd dist/easy-han
./easy-han
```

使用 -x 传入额外的参数去加密脚本，因为 *tests* 和 *vnev* 下面也有很多脚本，但是这些不需要加密，所以使用 --exclude 选项把它们排除。要确认可以使用这些选项可以正常加密脚本:

```
cd /path/to/src
pyarmor obfuscate -r --exclude vnev --exclude tests main.py
```

重要: 命令 *pack* 会自动加密脚本，所以不要使用该命令去打包加密后的脚本，打包加密脚本会导致错误，因为脚本加密之后是无法自动找到的其他被引用的模块的。

注解: 从 PyArmor 5.5.0 开始，可以传入选项 --advanced 启用高级模式来更进一步的提高加密脚本的安全性。例如:

```
pyarmor pack -x " --advanced 1 --exclude tests" foo.py
```

4.2 使用 Apache 的 mod_wsgi 发布加密的 Django 应用

下面是一个 Django 应用的目录结构:

```
/path/to/mysite/
    db.sqlite3
    manage.py
    mysite/
        __init__.py
        settings.py
```

(下页继续)

(续上页)

```
        urls.py
        wsgi.py
    polls/
        __init__.py
        admin.py
        apps.py
        migrations/
            __init__.py
        models.py
        tests.py
        urls.py
        views.py
```

首先加密所有脚本:

```
# 创建加密后脚本的存放路径
mkdir -p /var/www/obf_site

# 先把原来的文件都拷贝过去，因为 pyarmor 不会处理数据文件
cp -a /path/to/mysite/* /var/www/obf_site/

cd /path/to/mysite

# 递归加密当前目录下面的所有 .py 文件，并指定主文件为 wsgi.py
# 加密后的脚本保存到 /var/www/obf_site 下面，覆盖原来的 .py 文件
pyarmor obfuscate --src="." -r --output=/var/www/obf_site mysite/wsgi.py
```

然后修改 Apache 的配置文件:

```
WSGIScriptAlias / /var/www/obf_site/mysite/wsgi.py
WSGIPythonHome /path/to/venv

# pyarmor 的运行文件在这个目录下面，所以需要增加到 Python 路径里面
WSGIPythonPath /var/www/obf_site

<Directory /var/www/obf_site/mysite>
    <Files wsgi.py>
        Require all granted
    </Files>
</Directory>
```

最后重新启动 Apache:

```
apachectl restart
```

CHAPTER 5

使用工程

工程是一个包含配置文件的目录，可以用来方便的管理加密脚本。

使用工程管理脚本的有下列优点:

- 可以递增式加密脚本，仅仅加密修改过的脚本，适用于需要加密脚本很多的项目
- 定制选择工程包含的脚本文件，而不是一个目录下全部脚本
- 设置加密模式和定制保护代码
- 更加方便的管理加密脚本

5.1 使用工程管理加密脚本

首先使用命令*init* 创建一个工程:

```
cd examples/pybench
pyarmor init --entry=pybench.py
```

这会在当前目录下面创建一个工程配置文件 `.pyarmor_config` 。也可以在其他目录创建一个工程:

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

新创建的工程配置文件存放在 *projects/pybench* 。

使用工程的方式一般是切换当前路径到工程目录，然后运行工程相关命令:

```
cd projects/pybench
pyarmor info
```

使用命令*build* 加密工程中包含的所有脚本:

```
pyarmor build
```

当某些脚本修改之后，再次运行*build* ，加密这些修改过的脚本:

```
pyarmor build
```

使用命令*config* 来修改工程的配置。

设置工程的 `--manifest` 选项，可以完全定制工程中所需要加密的脚本，例如，把 `dist`, `test` 目录下面的所有 *.py* 排除在工程之外:

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

详细内容请参考后面的章节工程配置文件 中关于属性 *manifest* 的说明。

默认情况下*build* 仅仅加密修改过的文件，强制加密所有脚本:

```
pyarmor build --force
```

运行加密后的脚本:

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

5.2 使用不同加密模式

配置不同的加密模式:

```
pyarmor config --obf-mod=1 --obf-code=0
```

使用新的加密模式重新加密脚本:

```
pyarmor build -B
```

5.3 使用子工程加密需要特殊处理的脚本

假定在一个工程中大部分的脚本使用约束模式 3 进行加密，但是很少的几个需要使用约束模式 2 进行加密，那么这时候使用子工程就很方便。

1. 首先在脚本所在路径创建一个工程:

```
cd /path/to/src
pyarmor init --entry foo.py
pyarmor config --restrict 3
```

2. 接着拷贝工程配置文件为子工程 *.pyarmor_config-1*:

```
cp .pyarmor_config .pyarmor_config-1
```

3. 然后配置这个子工程，没有主脚本，包含几个特殊脚本，并且约束模式为 2:

```
pyarmor config --entry "" \
               --manifest "include a.py other/path/sa*.py" \
               --restrict 2 \
               .pyarmor_config-1
```

4. 最后依次加密工程和子工程:

```
pyarmor build -B
pyarmor build --no-runtime -B .pyarmor_config-1
```

一般情况下我们并不需要在工程中把子工程中的脚本排除在外，因为加密子工程之后，输出目录相同，原来的加密脚本会被新的加密脚本覆盖。

5.4 工程配置文件

每一个工程都有一个 JSON 格式的工程配置文件，它包含的属性如下：

- name

  工程名称

- title

  工程标题

- src

  工程所包含脚本的路径，可以是绝对路径，也可以是相对路径，相对于工程所在的目录。

- manifest

  选择和设置工程包含的脚本，其支持的格式和 Python Distutils 中的 MANIFEST.in 是一样的。默认值为 *src* 下面的所有 *.py* 文件:

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

多个模式使用逗号分开，例如:

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

关于所有支持的模式，参考 https://docs.python.org/2/distutils/sourcedist.html#commands

- is_package

  可用值: 0, 1, None

  主要会影响到加密脚本的保存路径，如果设置为 1，那么输出路径会额外包含包的名称。

- restrict_mode

  加密脚本的约束模式，可用值: 0, 1, 2, 3, 4

  默认值为 1，即加密脚本不能被修改。

  参考约束模式

  注解: 属性 *disable_restrict_mode* 从 v5.5.6 之后不再使用，而是转换为等价的 *restrict_mode* 。

- entry

  工程的主脚本，可以是多个，以逗号分开:

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

  主脚本可以是绝对路径，也可以是相对路径，相对于工程的 *src* 路径。

- output

  输出路径，保存加密后的脚本和运行辅助文件，相对于工程配置文件所在的目录。

- capsule

  警告: 在 v5.9.0 中已经删除

  工程使用的密钥箱，默认是全局密钥箱 。

- obf_code

  是否加密每一个函数（代码块），参考代码加密模式:

  – 0

不加密

– 1 （默认）

加密每一个函数

– 2

使用比模式 1 更复杂的算法来加密每一个函数

- wrap_mode

  是否使用 *try..final* 结构包裹原理的代码块，参考代码包裹模式:

  – 0

  不包裹

  – 1 （默认）

  包裹每一个代码块

- obf_mod

  是否加密整个模块，参考模块加密模式:

  – 0

  不加密

  – 1 （默认）

  加密模块

- cross_protection

  是否在主脚本插入交叉保护代码:

  – 0

  不插入

  – 1

  插入默认的保护代码，参考对主脚本的特殊处理

  – 文件名称

  使用文件指定的自定义模板

- runtime_path

  None 或者任何路径名称

  用来告诉加密脚本去哪里装载动态库 *__pytransform* 。

  默认值为 None ，是指在运行辅助包 里面。

主要用于使用打包工具（例如 py2exe）把加密脚本压缩到一个 *.zip* 文件的时候，无法正确定位动态库，这时候把 *runtime_path* 设置为空字符串可以解决这个问题。

- plugins

  None 或者列表

  用来扩展加密脚本的认证方式，支持多个插件，例如:

```
plugins: ["check_ntp_time", "show_license_info"]
```

  关于插件的使用实例，请参考使用插件扩展认证方式

- package_runtime

  保存运行辅助文件的方式:

  – 0

  和加密脚本存放在相同目录，这也是 v5.7.0 之前的存放方式

  – 1 （默认值）

  把运行辅助文件作为包存放在子目录 *pytransform* ，这样目录结构更清晰

- enable_suffix

  注解: 在 v5.8.7 中新增

  是否生成带有后缀名称的运行辅助包，当需要导入其他用户加密的模块时候，需要生成一个唯一后缀的运行辅助包。可用的值:

  – 0 (默认值)

  运行辅助包（模块）的名称没有后缀，固定为 `pytransform`

  – 1

  运行辅助包（模块）有后缀，例如，`pytransform_vax_00001`

- platform

  注解: 在 v5.9.0 新增

  仅仅在跨平台加密的时候需要，指定加密脚本的运行平台，多个平台使用逗号分开，平台名称必须是标准名称。

- license_file

注解: 在 v5.9.0 新增

使用这个许可文件来替换默认的许可文件，如果为空则使用默认的许可文件。

如果非空则文件名称必须是 *license.lic* ，相对路径则是基于当前工程文件的路径。

- bootstrap_code

注解: 在 v5.9.0 新增

控制如何在加密后的主脚本中生成引导代码:

– 0

不要在主脚本插入引导代码

– 1 (默认)

在主脚本中插入引导代码，如果主脚本的名称是 `__init__.py` ，那么使用包含前置的 `.` 的相对导入方式，否则使用绝对导入方式。

– 2

在主脚本插入引导代码的时候总是使用绝对导入的方式。

– 3

在主脚本插入引导代码的时候总是使用相对导入的方式，会有一个前置的 `.`

CHAPTER 6

命令手册

PyArmor 是一个命令行工具，用来加密脚本，绑定加密脚本到固定机器或者设置加密脚本的有效期。

pyarmor 的语法格式:

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

常用的命令包括:

```
obfuscate     加密脚本
licenses      为加密脚本生成新的许可文件
pack          打包加密脚本
hdinfo        获取硬件信息
```

和工程相关的命令:

```
init          创建一个工程，用于管理需要加密的脚本
config        修改工程配置信息
build         加密工程里面的脚本

info          显示工程信息
check         检查工程配置信息是否正确
```

其他不常使用的命令:

```
benchmark     测试加密脚本的性能
register      生效注册文件
download      查看和下载预编译的动态库
runtime       创建运行辅助包
```

可以运行 *pyarmor <command> -h* 查看各个命令的详细使用方法。

注解: 从 v5.7.1 开始，下面这些命令的首字母可以作为别名直接使用:

```
obfuscate, licenses, pack, init, config, build
```

例如:

```
pyarmor o 等价于 pyarmor obfuscate
```

6.1 obfuscate

加密 Python 脚本。

语法:

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

选项

-O, --output PATH 输出路径，默认是 *dist*

-r, --recursive 递归模式加密所有的脚本

-s, --src PATH 当主脚本不在顶层目录的时候指定搜索脚本的路径

--exclude PATH 在递归模式下排除某些目录，多个目录使用逗号分开，或者使用该选项多次

--exact 只加密命令行中列出的脚本

--no-bootstrap 在主脚本中不要插入引导代码

--no-cross-protection 在主脚本中不要插入交叉保护代码

--plugin NAME 在加密之前，向主脚本中插入代码。这个选项可以使用多次。

--platform NAME 指定运行加密脚本的平台

--advanced <0,1,2> 使用高级模式 *1* 或者超级模式 *2* 加密脚本

--restrict <0,1,2,3,4> 设置约束模式

--package-runtime <0,1> 是否把运行文件保存为包的形式

--no-runtime 不生成任何运行辅助文件，只加密脚本

--bootstrap <0,1,2,3> 如何生成引导代码

--enable-suffix 生成带有后缀名称的运行辅助包

--obf-code <0,1,2> 指定代码加密模式

--obf-mod <0,1> 指定模块加密模式

--wrap-mode <0,1> 指定包裹加密模式

--with-license FILENAME 使用指定的许可文件，特殊值 *outer* 表示使用外部许可文件

--cross-protection FILENAME 使用定制的交叉保护脚本

描述

PyArmor 首先检查用户根目录下面是否存在 `.pyarmor_capsule.zip` ，如果不存在，那么创建一个新的。

接着搜索需要加密的脚本，共有三种搜索模式：

- 默认模式：搜索和主脚本相同目录下面的所有 *.py* 文件
- 递归模式：递归搜索和主脚本相同目录下面的所有 *.py* 文件
- 精准模式：仅仅加密命令行中列出的脚本

加密命令只处理 *.py* 文件，不会拷贝数据文件到输出目录。如果被加密的包有很多数据文件，可以先把整个包拷贝到输出目录，然后在进行加密，这样加密后的 *.py* 文件就会覆盖输出目录中 *.py* 文件。

PyArmor 首先会修改主脚本，在其中插入交叉保护代码，详细处理过程请参考对主脚本的特殊处理

如果在命令行中指定了插件，那么 PyArmor 会扫描全部的脚本，根据规则注入相关插件代码，详细处理过程参考如何处理插件 。

接着把搜索到脚本全部加密，保存到输出目录 *dist* 。

然后创建运行辅助包 ，保存到输出目录 *dist* 。

最后插入引导代码 到主脚本。

只有精准模式 *–exact* 被指定时候，命令行列出的所有脚本都被作为主脚本，在其他模式下面，只有第一个脚本是主脚本，不会在其他脚本中插入引导代码和交叉保护代码。

当主脚本没有在顶层目录的时候，需要使用选项 `--src` 用来指定搜索 `.py` 文件的路径。例如:

```
# 没有 --src 的话，"./mysite" 是搜索脚本的路径
pyarmor obfuscate --src "." --recursive mysite/wsgi.py
```

选项 `--plugin` 主要用于扩展加密脚本的授权方式，例如检查网络时间来校验有效期等，这个选项指定的插件名称会在加密之前插入到主脚本中。插件对应的文件为当前目录下面 *名称.py* ，如果插件不在当前路径，可以使用绝对路径指定插件名称，更多详细的内容参考如何处理插件 。关于插件的使用实例，请参考使用插件扩展认证方式

选项 `--platform` 用于指定加密脚本的运行平台，仅用于跨平台发布。因为加密脚本的运行文件中包括平台相关的动态库，所以跨平台发布需要指定该选项。这个选项可以使用多次，以支持加密脚本运行于不同平台。从 v5.7.5 开始，所有平台名称已经标准化，可用的平台名称可以使用命令*download* 查看。

选项 `--restrict` 用于指定加密脚本的约束模式，关于约束模式的详细说明，参考约束模式

如果指定了超级加密模式 `--advanced 2` ，这是一种和以前有很大区别的模式，下面的运行辅助文件和引导代码都不存在，只有一个运行需要的扩展模块:

```
pytransform.so or pytransform.dll
```

运行辅助文件

默认情况下，所有运行时刻文件会作为包保存在一个单独的目录 *pytransform* 下面:

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

只有当选项 `--package-runtime` 设置为 *0* 的时候，生成的运行辅助文件和加密脚本存放在相同的目录下面:

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

如果指定了选项 `--enable-suffix` ，那么运行辅助包（模块）的名称会包含一个后缀，例如，`pytransform_xxxx` 。这里 `xxxx` 是根据 PyArmor 注册码得到的具有唯一性的字符串。

引导代码

默认情况下，下面的引导代码 会被插入到加密后的主脚本中:

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

当主脚本是 `__init__.py` 的时候，会使用包含一个 `.` 的相对导入的方式:

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

如果选项 `--bootstrap` 是 2, 那么引导代码 总是使用绝对导入的方式；如果它被设置为 `3` ，那么总是使用包含前置 `.` 的相对导入方式。

另外如果指定了选项 `--enable-suffix` 的话，引导代码 可能会是这样子的:

```
from pytransform_vax_000001 import pyarmor_runtime
pyarmor_runtime(suffix='vax_000001')
```

如果设置了 `--no-bootstrap` 或者 `--bootstrap` 为 *0* ，那么就不会插入引导代码到主脚本中。

示例

- 加密当前目录下的所有 *.py* 脚本，保存到 *dist* 目录:

```
pyarmor obfuscate foo.py
```

- 递归加密当前目录下面的所有 *.py* 脚本，保存到 *dist* 目录:

```
pyarmor obfuscate --recursive foo.py
```

- 递归加密当前目录下面的所有脚本，主脚本在子目录 *mysite/* 下面:

```
pyarmor obfuscate --src "." --recursive mysite/wsgi.py
```

- 仅仅递归加密指定目录下面的所有 *.py* 脚本，没有主脚本，也不生成运行辅助文件:

```
pyarmor obfuscate --recursive --no-runtime .
pyarmor obfuscate --recursive --no-runtime src/
```

- 除了 *build* 和 *dist* 之外，递归加密当前目录下面的所有 *.py* 脚本，保存到 *dist* 目录:

```
pyarmor obfuscate --recursive --exclude build,dist foo.py
pyarmor obfuscate --recursive --exclude build --exclude tests foo.py
```

- 仅仅加密两个脚本 *foo.py*, *moda.py*:

```
pyarmor obfuscate --exact foo.py moda.py
```

- 只加密主脚本 *foo.py* ， 不要生成其他任何运行文件:

```
pyarmor obfuscate --no-runtime --exact foo.py
```

- 加密包 *mypkg* 所在目录下面的所有 *.py* 文件:

```
pyarmor obfuscate --output dist/mypkg mypkg/__init__.py
```

- 加密当前目录下面所有的 *.py* 文件，但是不要插入交叉保护代码到主脚本 `dist/foo.py`:

```
pyarmor obfuscate --no-cross-protection foo.py
```

- 加密当前目录下面所有的 *.py* 文件，但是不要插入引导代码到主脚本 `dist/foo.py`:

```
pyarmor obfuscate --no-bootstrap foo.py
```

- 在加密 *foo.py* 之前，把当前目录下面的 *check_ntp_time.py* 的内容插入到 *foo.py* 中:

```
pyarmor obfuscate --plugin check_ntp_time foo.py
```

- 仅当插件中的函数 *assert_armored* 被调用的时候导入插件 *assert_armored*:

```
pyarmor obfuscate --plugin @assert_armored foo.py
```

- 在 MacOS 平台下加密脚本，这些加密脚本将在 Ubuntu 下面运行，使用下面的命令进行加密:

```
pyarmor obfuscate --platform linux.x86_64 foo.py
```

- 使用高级模式加密脚本:

```
pyarmor obfuscate --advanced 1 foo.py
```

- 使用约束模式 2 加密脚本:

```
pyarmor obfuscate --restrict 2 foo.py
```

- 使用约束模式 4 加密当前目录下面除了 *__init__.py* 之外的所有 *.py* 文件:

```
pyarmor obfuscate --restrict 4 --exclude __init__.py --recursive .
```

- 生成一个可以被其他加密模块导入的包:

```
cd /path/to/mypkg
pyarmor obfuscate -r --enable-suffix --output dist/mypkg __init__.py
```

- 使用超级模式进行加密，并使用有限制期限的许可文件:

```
pyarmor licenses -e 2020-10-05 regcode-01
pyarmor obfuscate --with-license licenses/regcode-01/license.lic \
                  --advanced 2 foo.py
```

- 使用超级模式进行加密，并使用定制的交叉保护脚本，同时使用外部的许可文件，不要把 `license.lic` 嵌入到扩展模块中:

```
pyarmor obfuscate --cross-protection build/pytransform_protection.py \
                  --with-license outer --advanced 2 foo.py
```

6.2 licenses

为加密脚本生成新的许可文件

语法:

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

选项

-O OUTPUT, --output OUTPUT 输出路径，可以为 stdout 或者 stderr

-e YYYY-MM-DD, --expired YYYY-MM-DD 加密脚本的有效期

-d SN, --bind-disk SN 绑定加密脚本到硬盘序列号

-4 IPV4, --bind-ipv4 IPV4 绑定加密脚本到指定 IP 地址

-m MACADDR, --bind-mac MACADDR 绑定加密脚本到网卡的 Mac 地址

-x, --bind-data DATA 用于扩展认证类型的时候传递认证数据信息

--disable-restrict-mode 运行加密脚本时候禁用所有约束模式

--enable-period-mode 运行加密脚本时候周期性（每小时）检查许可文件

–fixed key,... 绑定加密脚本到特定的 Python 解释器

描述

运行加密脚本必须有一个认证文件 `license.lic` 。一般在加密脚本的同时，会自动生成一个缺省的认证文件。但是这个缺省的认证文件允许加密脚本运行在任何机器并且永不过期。如果你需要对加密脚本进行限制，那么需要使用该命令生成新的许可文件，并覆盖原来的许可文件。

例如，下面的命令生成一个有使用期限的认证文件:

```
pyarmor licenses --expired 2019-10-10 mycode
```

生成的新的认证文件保存在默认输出路径和注册码组合路径 *licenses/mycode* 下面，使用这个新的许可文件覆盖默认的许可文件:

```
cp licenses/mycode/license.lic dist/pytransorm/
```

另外一个例子，限制加密脚本在固定 Mac 地址，同时设置使用期限:

```
pyarmor licenses --expired 2019-10-10 --bind-mac 2a:33:50:46:8f tom
cp licenses/tom/license.lic dist/
```

在这之前，一般需要运行命令*hdinfo* 得到硬件的相关信息:

```
pyarmor hdinfo
```

选项 *-x* 可以把任意字符串数据存放到许可文件里面，主要用于自定义认证类型的时候，传递参数给自定义认证函数。例如:

```
pyarmor licenses -x "2019-02-15" tom
```

然后在加密脚本中，可以从认证文件的信息中查询到传入的数据。例如:

```
from pytransfrom import get_license_info
info = get_license_info()
print(info['DATA'])
```

也可以输出生成的许可文件到标准输出，例如:

```
pyarmor --silent licenses --output stdout -x "2019-05-20" reg-0001
```

有时候我们需要把加密脚本绑定到某一个 Python 解释器上，例如，不允许用户使用自定义的 Python 解释器来执行加密脚本，必须使用官方版本。这时候可以使用选项 `--fixed` 来指定解释器的特征码。例如，使用特殊的特征码 *1* 绑定到当前 Python 解释器:

```
pyarmor licenses --fixed 1
```

也可以绑定到多个 Python 解释器，只需要把多个特征码使用逗号分开即可:

```
pyarmor licenses --fixed 4265050,5386060
```

如何获得解释器的特征码，请参考*绑定加密脚本到固定的 Python 解释器*

注解: 这里有一个实际使用的例子*使用插件扩展认证方式*

6.3 pack

加密并打包脚本或者工程。

语法:

```
pyarmor pack <options> SCRIPT | PROJECT
```

选项

-O, --output PATH 输出路径

-e, --options OPTIONS 传递额外的参数到 PyInstaller

-x, --xoptions OPTIONS 传递额外的参数到*obfuscate* 去加密脚本

-s FILE 使用外部的.spec 文件来打包加密脚本

--clean 打包之前删除缓存的文件

--without-license 不要将加密脚本的许可文件打包进去

--with-license FILE 使用指定的许可文件替换加密脚本默认的许可文件

--debug 不要删除打包过程生成的中间文件

--name 指定最终生成的包的名称，默认是脚本的名称

描述

命令*pack* 首先调用 PyInstaller 生成一个和主脚本同名的 *.spec* 文件，选项 `--e` 的值会原封不动的被传递给 PyInstaller ，但是不能传递这些选项 `-y` ， `--noconfirm` ， `-n` ， `--name` ， `--distpath` ， `--specpath` ，因为这些会被*pack* 命令内部使用。

当*pack* 命令失败的时候，首先要确认脚本文件可以直接用 PyInstaller 打包成功，例如:

```
pyinstaller foo.py
```

通常情况下，只要 PyInstaller 能打包成功，然后额外的参数通过 `-e` 传递过去，命令*pack* 也不会有问题。

接下来*pack* 会递归加密主脚本所在目录下面的所有 *.py* 文件。它会使用 `-x` 中指定的额外选项来调用命令*obfuscate* ，但是不可以在 `-x` 传入选项 `-r` ， `--output` ， `--package-runtime` 等，这些会被*pack* 内部使用。当打包一个工程的时候，*pack* 会直接调用命令*build* 加密工程，选项 `-x` 的值会被忽略，这时候加密选项的控制是通过配置工程来实现。

然后*pack* 会基于原来的 *.spec* 文件，创建一个新的 *.spec* 文件，增加一些语句用于把原来的脚本替换为加密后的脚本。

最后*pack* 再次调用 PyInstaller 使用这个打过补丁的 *.spec* 文件来创建最终的输出。更多详细说明，请参考*如何打包加密脚本*.

如果设置了选项 `--debug` ，例如:

```
pyarmor pack --debug foo.py
```

下面这些中间文件会被保留下来，正常情况下它们在打包成功之后会被删除:

```
foo.spec
foo-patched.spec
dist/obf/temp/hook-pytransform.py
dist/obf/*.py                          # All the obfuscated scripts
```

这个打过补丁的 *foo-patched.spec* 可以被 PyInstaller 直接用来打包加密脚本，例如:

```
pyinstaller -y --clean foo-patched.spec
```

当有些脚本被修改之后，只需要重新加密，然后调用这个命令快速打包，而加密命令*obfuscate* 需要的全部选项可以从命令*pack* 在控制台的输出中找到。

如果需要更改输出的可执行文件的名称，直接设置选项 `--name` ，不要使用 `-e` 把这个选项传递给 PyInstaller ，因为这个选项需要一些特殊处理。

如果已经有一个能够打包的 *.spec* 文件，只需要通过选项 `-s` 指定这个文件（这种情况下选项 `-e` 的值会被忽略），例如:

```
pyarmor pack -s foo.spec foo.py
```

主脚本（这里是 *foo.py*) 需要在命令行列出，否则*pack* 就不知道那些脚本需要加密，详细说明请参考使用定制的*.spec* 文件打包加密脚本 。

如果有很多数据文件或者隐含模块，最好的方式是使用 Hook 脚本来自动发现它们。首先创建一个文件 `hook-sys.py` ，下面是一些示例代码:

```
from PyInstaller.utils.hooks import collect_data_files, collect_all
datas, binaries, hiddenimports = collect_all('my_module_name')
datas += collect_data_files('submodule')
hiddenimports += ['_gdbm', 'socket', 'h5py.defs']
datas += [ ('/usr/share/icons/education_*.png', 'icons') ]
```

接下来使用额外选项 `--additional-hooks-dir .` 来调用*pack* ，告诉 PyInstaller 在当前路径下面搜索 Hook 脚本:

```
pyarmor pack -e " --additional-hooks-dir ." foo.py
```

更多关于 Hook 脚本的信息，参考 https://pyinstaller.readthedocs.io/en/stable/hooks.html#understanding-pyinstaller-hooks

最后，如果打包过程出现问题，打开 PyArmor 调试标志来输出详细的错误信息:

```
pyarmor -d pack ...
```

示例

- 加密脚本 foo.py 并打包到 *dist/foo* 下面:

  ```
  pyarmor pack foo.py
  ```

- 删除缓存的 *foo.spec* 和其他中间文件，开始一个全新的打包:

```
pyarmor pack --clean foo.py
```

- 使用已经写好的 *myfoo.spec* 来打包加密脚本:

```
pyarmor pack -s myfoo.spec foo.py
```

- 传递额外的参数运行 *PyInstaller*:

```
pyarmor pack --options '-w --icon app.ico' foo.py
```

- 打包的时候不要加密目录 *venv* 和 *test* 下面的所有文件:

```
pyarmor pack -x " --exclude venv --exclude test" foo.py
```

- 使用高级模式加密脚本，然后打包成为一个可执行文件:

```
pyarmor pack -e " --onefile" -x " --advanced 1" foo.py
```

- 使用有时间限制的许可证打包一个可执行文件:

```
pyarmor licenses -e 2020-12-25 cy2020
pyarmor pack --with-license licenses/cy2020/license.lic foo.py
```

- 生成名称为 my_app 的安装包:

```
pyarmor pack --name my_app foo.py
```

- 使用高级模式加密脚本并打包一个工程:

```
pyarmor init --entry main.py
pyarmor config --advanced 1
pyarmor pack .
```

注解: 从 v5.9.0 开始，也可以直接打包一个工程，只需要最后一个参数指定工程所在的路径，就可以加密工程中的文件，并使用工程主脚本进行打包。例如:

```
pyarmor init --entry main.py
pyarmor pack .
```

首先在当前目录创建一个工程，然后直接打包该工程，使用工程的好处是可以完全定制加密选项。

重要: 命令 *pack* 会自动加密脚本，所以不要使用该命令去打包加密后的脚本，打包加密脚本会导致错误，

因为脚本加密之后是无法自动找到的其他被引用的模块的。

6.4 hdinfo

显示当前机器的硬件信息，例如硬盘序列号，网卡 Mac 地址等。

这些信息主要用来为加密脚本生成许可文件的时候使用。

语法:

```
pyarmor hdinfo
```

如果没有装 *pyarmor*, 也可以在这里下载获取硬件信息的小工具 *hdinfo*

https://github.com/dashingsoft/pyarmor-core/tree/master/#hdinfo

然后直接运行:

```
hdinfo
```

获取得到的硬件信息和这里显示的是一样的。

6.5 init

创建管理加密脚本的工程文件。

语法:

```
pyarmor init <options> PATH
```

选项:

-t, --type <auto,app,pkg> 工程类型，默认是 *auto*

-s, --src SRC 脚本所在路径，默认是当前路径

-e, --entry ENTRY 主脚本名称

描述

这个命令会在 *PATH* 指定的路径创建一个工程配置文件 `.pyarmor_config` ，这个一个 JSON 格式的文件。

如果选项 `--type` 是 *auto* (也是默认情况),那么工程类型根据主脚本命令来判断。如果主脚本是 *__init__.py* , 那么工程类型就是 *pkg* , 否则就是 *app* 。

如果新的工程类型为 *pkg* ，不管是自动判断还是选项指定，*init* 命令都会设置工程属性 *is_package* 为 *1* ，这个属性的默认值是 *0* 。

工程创建之后，可以使用命令*config* 进行修改和配置。

示例

- 在当前路径创建一个工程:

```
pyarmor init --entry foo.py
```

- 创建一个工程在构建路径 *obf*:

```
pyarmor init --entry foo.py obf
```

- 创建一个 *pkg* 类型的工程:

```
pyarmor init --entry __init__.py
```

- 在构建路径 *obf* 创建一个工程，管理在 */path/to/src* 处的脚本:

```
pyarmor init --src /path/to/src --entry foo.py obf
```

6.6 config

修改工程配置。

语法:

```
pyarmor config <options> [PATH]
```

选项

--name NAME 工程名称

--title TITLE 显示标题

--src SRC 工程脚本所在的路径，用于匹配模板命令

--output OUTPUT 保存加密脚本的输出路径

--manifest TEMPLATE 过滤脚本的模板语句

--entry SCRIPT 工程主脚本，可以多个，使用逗号分开

--is-package <0,1> 管理的脚本是一个 Python 包类型

--restrict <0,1,2,3,4> 设置约束模式

--obf-mod <0,1> 是否加密整个模块对象

--obf-code <0,1,2> 是否加密每一个函数

--wrap-mode <0,1> 是否启用包裹模式加密函数

--advanced <0,1,2> 使用高级模式 *1* 或者超级模式 *2* 加密脚本

--cross-protection <0,1> 是否插入交叉保护代码到主脚本，也可以直接指定脚本名称

--runtime-path RPATH 设置运行文件所在路径

--plugin NAME 设置需要插入到主脚本的代码文件，这个选项可以使用多次

--package-runtime <0,1> 是否保存运行文件为包的形式

--bootstrap <0,1,2,3> 如何生成引导代码

--with-license FILENAME 使用指定的许可文件，特殊值 *outer* 表示使用外部许可文件

描述

在工程所在路径运行该命令，修改一个或者多个工程属性:

```
pyarmor config --option new-value
```

或者在命令的最后面指定工程所在的路径:

```
pyarmor config --option new-value /path/to/project
```

选项 `--entry` 用来指定工程主脚本，可以是多个，以逗号分开:

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

主脚本可以是绝对路径，也可以是相对路径，相对于 *src* 指定的路径。

选项 `--manifest` 用来选择和设置工程包含的脚本。默认值为 *src* 下面的所有 *.py* 文件:

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

多个模式使用逗号分开，例如:

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

关于所有支持的模式，参考 https://docs.python.org/2/distutils/sourcedist.html#commands

选项 `--plugin` 为空字符串有特殊作用，用来清除所有的插件。

所有选项的作用，请参考 工程配置文件

示例

- 修改工程名称和标题:

  ```
  pyarmor config --name "project-1"  --title "My PyArmor Project"
  ```

- 修改工程主脚本:

```
pyarmor config --entry foo.py,hello.py
```

- 排除路径 *build* 和 *dist* ，下面的的所有 *.py* 文件会被忽略:

```
pyarmor config --manifest "global-include *.py, prune build, prune dist"
```

- 使用非包裹模式加密脚本，这样可以提高加密脚本运行速度，但是会降低安全性:

```
pyarmor config --wrap-mode 0
```

- 配置主脚本的插件，下面的例子中会把 *check_ntp_time.py* 的内容插入到主脚本，这个脚本会检查网络时间，超过有效期会自动退出:

```
pyarmor config --plugin check_ntp_time.py
```

- 清除所有插件:

```
pyarmor config --plugin ''
```

6.7 build

加密工程中的所有脚本。

语法:

```
pyarmor build <options> [PATH]
```

选项

-B, --force 强制加密所有脚本，默认情况只加密上次构建之后修改过的脚本

-r, --only-runtime 只生成运行依赖文件

-n, --no-runtime 只加密脚本，不要生成运行依赖文件

-O, --output OUTPUT 输出路径，如果设置，那么工程属性里面的输出路径就无效

--platform NAME 指定加密脚本的运行平台，仅用于跨平台发布

--package-runtime <0,1> 是否保存运行文件为包的形式

描述

可以直接在工程所在路径运行该命令:

```
pyarmor build
```

或者在命令行指定工程所在路径:

```
pyarmor build /path/to/project
```

选择 --no-runtime 可能会影响引导代码 的生成方式，设置之后在主脚本中引导代码总是会使用绝对导入的方式。

选项 --platform 和 --package-runtime 的使用，请参考命令*obfuscate*

示例

- 递增式加密工程中的脚本，上次运行该命令之后，没有修改过的脚本不会再被加密:

```
pyarmor build
```

- 强制加密工程中的所有脚本，即便是没有修改:

```
pyarmor build -B
```

- 仅仅生成运行加密脚本需要的依赖文件，不要加密脚本:

```
pyarmor build -r
```

- 只加密脚本，不要生成其他的依赖文件:

```
pyarmor build -n
```

- 忽略工程中设置的输出路径，保存加密脚本到新路径:

```
pyarmor build -B -O /path/to/other
```

- 在 MacOS 平台下加密脚本，这些加密脚本将在 Ubuntu 下面运行，使用下面的命令进行加密:

```
pyarmor build -B --platform linux.x86_64
```

6.8 info

显示工程配置信息。

语法:

```
pyarmor info [PATH]
```

描述

可以直接在工程所在路径运行该命令:

```
pyarmor info
```

或者在命令行指定工程所在路径:

```
pyarmor info /path/to/project
```

6.9 check

检查工程文件的配置是否正确。

语法:

```
pyarmor check [PATH]
```

描述

可以直接在工程所在路径运行该命令:

```
pyarmor check
```

或者在命令行指定工程所在路径:

```
pyarmor check /path/to/project
```

6.10 banchmark

测试加密脚本的性能。

语法:

```
pyarmor benchmark <options>
```

选项:

-m, --obf-mode <0,1> 是否加密模块

-c, --obf-code <0,1> 是否单独加密每一个函数

-w, --wrap-mode <0,1> 是否使用包裹模式加密函数

--debug 不要清理生成的测试脚本

描述

主要用来检查加密脚本的性能，命令输出包括初始化加密脚本运行环境需要的额外时间，以及不同加密模式下面导入模块、运行不同大小的代码块需要消耗的额外时间。

示例

- 测试默认加密模式的性能:

```
pyarmor benchmark
```

- 测试不使用包裹模式的性能:

```
pyarmor benchmark --wrap-mode 0
```

- 查看测试过程中使用的脚本，保存在 *.benchtest* 目录下面:

```
pyarmor benchmark --debug
```

6.11 register

生效注册文件，显示注册信息。

语法:

```
pyarmor register [KEYFILE]
```

描述

生效购买的 PyArmor 注册文件:

```
pyarmor register /path/to/pyarmor-regfile-1.zip
```

查看注册信息:

```
pyarmor register
```

6.12 download

查看和下载不同平台下面的预编译的动态库。

语法:

```
pyarmor download <options> NAME
```

选项:

--help-platform 显示所有支持的规范化平台名称

--list PATTERN 查看所有可用的预编译动态库

-O, --output PATH 下载之后保存的路径

--update 更新已经下载的动态库

描述

这个命令主要是用来下载其他平台的动态库，一般用于交叉平台的发布。

列出所有可用平台的规范化名称，例如:

```
pyarmor download
pyarmor download --help-platform
pyarmor download --help-platform windows
pyarmor download --help-platform linux.x86_64
```

下载其中的一个。例如:

```
pyarmor download linux.armv7
pyarmor download linux.x86_64
```

默认情况下，下载的文件是保存在目录 `~/.pyarmor/platforms` 下面，使用相应的路径来存放不同平台的动态库。

选项 `--list` 会显示详细的动态库信息，同时也可以过滤平台，搜索名称、CPU 架构、动态库特征等。例如:

```
pyarmor download --list
pyarmor download --list windows
pyarmor download --list windows.x86_64
pyarmor download --list JIT
pyarmor download --list armv7
```

当 *pyarmor* 升级之后，已经下载的动态库不会自动更新，可以使用 `--update` 更新全部已经下载的动态库。例如:

```
pyarmor download --update
```

6.13 runtime

创建运行辅助包

SYNOPSIS:

```
pyarmor runtime <options>
```

OPTIONS:

-O, --output PATH 输出路径，默认是 *dist*

-n, --no-package 不要使用包的形式来存放生成运行文件

-i, --inside 创建包含引导脚本的包 *pytransform_bootstrap*

-L, --with-license FILE 使用这个文件替换默认的加密脚本许可文件，特殊值 *outer* 表示使用外部许可证

--platform NAME 生成其他平台下的运行辅助包

--enable-suffix 生成带有后缀名称的运行辅助包

--super-mode 为超级模式生成运行辅助文件

DESCRIPTION

这个命令主要用来创建运行辅助包

因为使用相同的全局密钥箱 加密的脚本可以共享运行辅助包 ，所以单独创建运行辅助包之后，在加密脚本的时候就不需要每一次都重新生成运行辅助文件。

它同时也会在输出目录下面创建一个引导脚本 `pytransform_bootstrap.py` ，这是对一个空脚本进行加密之后生成的，包含有引导代码 。它最主要用途就是可以让没有加密的脚本能够运行引导代码 。例如，在脚本中导入这个模块之后，其他加密模块就都可以被正常导入了:

```
import pytransform_bootstrap
import obf_mod
```

如果选项 `--inside` 被指定，那么将在输出目录使用包 `pytransform_bootstrap` 的形式来保存引导脚本。

从 v6.2.0 开始，这个命令还会另外创建辅助脚本 `pytransform_protection.py` ，这是默认的交叉保护脚本，这个脚本在运行时刻并不需要，它主要是作为模版来定制自己的交叉保护脚本，参考定制交叉保护脚本

选项 `--super-mode` 用来生成超级模式 的运行辅助文件，需要注意的是超级模式的运行辅助文件和其他模式是完全不一样的。

选项 `--platform` 和 `--enable-suffix` 的使用，请参考命令*obfuscate*

EXAMPLES

- 在默认输出路径 *dist* 下面创建运行辅助包 `pytransform`:

  ```
  pyarmor runtime
  ```

- 创建独立的运行辅助文件 ，但是不使用包的形式存放:

  ```
  pyarmor runtime -n
  ```

- 创建引导脚本在一个单独的包 `pytransform_bootstrap`:

```
pyarmor runtime -i
```

- 为 *armv7* 平台创建运行辅助包，并且设置加密脚本的使用期限:

```
pyarmor licenses --expired 2020-01-01 code-001
pyarmor runtime --with-license licenses/code-001/license.lic --platform linux.armv7
```

- 为超级模式创建运行辅助包:

```
pyarmor runtime --super-mode
pyarmor runtime --advanced 2
```

- 为超级模式创建运行辅助包，同时使用外部许可文件，也就是说，不要把许可文件嵌入到扩展模块里面:

```
pyarmor runtime --super-mode --with-license outer
```

CHAPTER 7

了解加密脚本

7.1 全局密钥箱

全局密钥箱是存放在用户主目录的一个文件 `.pyarmor_capsule.zip` 。当 PyArmor 加密脚本或者生成加密脚本的许可文件的时候，都需要从这个文件中读取数据。

所有的试用版本使用同一个密钥箱 *公共密钥箱*，这个密钥箱使用 1024 位 RSA 密钥对。

而对于正式版本，每一个用户都会收到一个专用的 *私有密钥箱*，这种密钥箱使用 2048 位 RSA 密钥对。

通过密钥箱是无法恢复加密后的脚本，所以即便都使用 *公共密钥箱*，加密后的脚本也是安全的，但是使用相同的密钥箱为加密脚本生成的许可是通用的。使用 *公共密钥箱*是无法真正的限制加密脚本的使用期限或者绑定到某一台指定设备上，因为别人同样可以使用 *公共密钥箱*为你的加密脚本生成合法的许可。

运行加密脚本并不需要密钥箱，只有在加密脚本和为加密脚本生成许可的时候才会用到。

7.2 加密后的脚本

和原来的脚本相比，被 PyArmor 加密后的脚本需要额外的运行辅助文件，下面是加密后在输出目录 *dist* 下的所有文件清单:

```
myscript.py
mymodule.py

pytransform/
```

(下页继续)

(续上页)

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

被加密的脚本也是一个普通的 Python 脚本，模块 *dist/mymodule.py* 加密后会是这样:

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

而主脚本 *dist/myscript.py* 被加密后则会是这样:

```
from pytransform import pyarmor_runtime
pyarmor_runtime()
__pyarmor__(__name__, __file__, b'\x0a\x02...')
```

7.2.1 超级加密脚本

使用超级模式 加密后的脚本和上面的有所不同，运行辅助文件只需要一个扩展文件 `pytransform` ，加密后输出目录 *dist* 下的所有文件清单:

```
myscript.py
mymodule.py

pytransform.so or pytransform.dll
```

而被加密的脚本也是一个普通的 Python 脚本，加密后会是这样:

```
from pytransform import pyarmor
pyarmor(__name__, __file__, b'\x0a\x02...', 1)
```

扩展模块也可能有一个后缀，例如:

```
from pytransform_vax_000001 import pyarmor
pyarmor(__name__, __file__, b'\x0a\x02...', 1)
```

注解: 下面章节中引导代码 ，运行辅助包 ，运行辅助文件 在超级模式加密后的脚本并不存在。

7.2.2 主脚本

在 PyArmor 中，主脚本并不一定是启动脚本，而是在运行 Python 解释器之后，第一个被执行（导入）的加密脚本。例如，如果只有一个 Python 包的被加密，那么这个包的 *__init__.py* 就是主脚本。

7.3 引导代码

主脚本的前两行就是 *引导代码*，它一般出现在主脚本中:

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

对于被加密的 Python 包，其主脚本是 *__init__.py* ，那么引导代码会使用相对导入的方式:

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

如果加密脚本的时候指定了运行时刻路径，引导代码会是这样的形式:

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

从 v5.8.7 开始，运行辅助包（模块）也可能会有个后缀，引导代码会像这样:

```
from pytransform_vax_000001 import pyarmor_runtime
pyarmor_runtime(suffix='_vax_000001')
```

7.4 运行辅助包

和加密脚本一起生成的目录 *pytransform* 是一个 Python 的包，叫做 *运行辅助包*。它是运行加密脚本的唯一依赖，只要它在任何一个 Python 路径下面，加密脚本就可以像普通脚本一样被正常执行。

通常情况下面运行辅助包和加密脚本在相同目录下，但是也可以在其他任何路径，只要可以使用 *import pytransform* 能够正常导入就可以。并且使用相同*全局密钥箱* 加密的脚本都可以共用这个包。你完全可以把这个包放到 Python 的系统库路径下面，这样加密后的脚本目录结构就和原来完全一样。

运行辅助包里面有四个文件:

```
pytransform/
    __init__.py                 Python 模块文件
    _pytransform.so/.dll/.lib   动态链接库，核心功能的实现
```

(下页继续)

(续上页)

pytransform.key	数据文件
license.lic	加密脚本的许可文件

在 v5.7.0 之前, 运行辅助包是另外一种存放形式 运行辅助文件

7.4.1 运行辅助文件

它们不是以 Python 包的形式存在，而是四个单独文件:

pytransform.py	Python 模块文件
_pytransform.so/.dll/.lib	动态链接库，核心功能的实现
pytransform.key	数据文件
license.lic	加密脚本的许可文件

很明显，使用运行辅助包的形式使得加密后的脚本目录结构更清晰。

从 v5.8.7 开始，运行辅助包（模块）也可能会有个后缀，例如:

```
pytransform_vax_000001/
    __init__.py
    ...

pytransform_vax_000001.py
...
```

7.5 加密脚本的许可文件

运行辅助文件中的 *license.lic* 作用比较特殊，它包含着对加密脚本的运行许可信息。在加密脚本的同时会在输出目录下面生成一个默认许可文件，该文件允许加密脚本运行在任何机器并且永不过期。

如果需要为加密脚本设置新的许可, 例如设置有效期, 限制加密脚本在特定机器上运行, 需要运行命令*licenses* 生成新的相应的许可文件，然后用新生成的 *license.lic* 覆盖原来的许可文件。

注解: PyArmor 的安装目录下面也有一个 *license.lic* ，这个文件主要是设置 PyArmor 自身的许可，这个许可是由我来发布的, :)

7.6 使用加密脚本的基本原则

- 加密后的脚本也是一个正常的 Python 脚本，它可以无缝替换原来的脚本

- 唯一的改变是，引导代码 必须被首先执行，加密脚本才能正常运行，否则会报错。
- 运行辅助包 必须在任何 Python 路径下面，确保引导代码 能被正确导入。
- 引导代码 会使用 *ctypes* 装载动态库 *__pytransform.so/.dll/.dylib* 。动态库是平台相关的，所有预编译的动态库列表在这里支持的平台列表
- 默认情况下，引导代码 会在运行辅助包 里面搜索动态库 *__pytransform* ，具体装载过程可以查看函数 *pytransform.__load_library* 的源代码。
- 如果动态库 *__pytransform* 没有在默认位置，需要修改引导代码 ，设置运行时刻路径:

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

  运行辅助文件 *license.lic* 和 *pytransform.key* 必须也在这个目录下面

- 如果在代码中动态创建新的执行环境，例如 *multiprocssing.Process*, *os.exec*, *subprocess.Popen* 等, 要确保引导代码 在新的执行环境被首先执行，否则加密脚本会报错。

更多详细的信息，可以参考如何加密脚本 和如何运行加密脚本

7.7 加密脚本和原脚本的区别

加密脚本和原来的脚本相比，存在下列一些的不同:

- 运行加密脚本的 Python 主版本和加密脚本使用的 Python 主版本应该要一致，因为加密的脚本实际上已经是 *.pyc* 文件，如果主版本不一致，有些指令无法识别或者会出错。尤其是 Python3.6，在这个版本引入了新的指令系统，所以和 Python3.5 以及之前的版本完全不同。
- 执行加密角本的 Python 不能是调试版，准确的说，不能是设置了 Py_TRACE_REFS 或者 Py_DEBUG 生成的 Python
- 使用 `sys.settrace`, `sys.setprofile`, `threading.settrace` 和 `threading.setprofile` 设置的回调函数在加密脚本中将被忽略
- 模块 `inspect` 的部分函数可能无法工作，并且其他任何模块如果试图访问加密脚本的源代码或者 Byte Code ，也会失败或者得到错误的数据
- 使用高级模式进行加密的脚本直接访问代码块的属性 `co_const` 可能会导致奔溃
- 代码块的属性 `__file__` 在加密脚本是 `<frozen name>` ，而不是文件名称，在异常信息中会看到文件名的显示是 `<frozen name>`

  需要注意的是模块的属性 `__file__` 还和原来的一样，还是文件名称。加密下面的脚本并运行，就可以看到输出结果的不同:

```
def hello(msg):
    print(msg)
```

(下页继续)

(续上页)

```
# The output will be 'foo.py'
print(__file__)

# The output will be '<frozen foo>'
print(hello.__file__)
```

7.8 第三方解释器的支持

对于第三方的解释器（例如 Jython 等）以及通过嵌入 Python C/C++ 代码调用加密脚本，需要满足下列条件:

- 第三方解释器或者嵌入的 Python 代码必须装载 Python 官方的动态库，动态库的源代码在 https://github.com/python/cpython ，并且核心代码不能被修改，修改后的代码可能会导致加密脚本无法执行。
- 在 Linux 下面装载 Python 动态库 *libpythonXY.so* 的时候 *dlopen* 必须设置 *RTLD_GLOBAL* ，否则加密脚本无法运行。

注解: Boost::python，默认装载 Python 动态库是没有设置 *RTLD_GLOAL* 的，运行加密脚本的时候会报错"No PyCode_Type found" 。解决方法就是在初始化的调用方法 *sys.setdlopenflags(os.RTLD_GLOBAL)* ，这样就可以共享动态库输出的函数和变量。

- 模块 *ctypes* 必须存在并且 *ctypes.pythonapi._handle* 必须被设置为 Python 动态库的句柄，PyArmor 会通过该句柄获取 Python C API 的地址。

CHAPTER 8

PyArmor 的工作原理

让我们看看一个普通的 Python 脚本 *foo.py* 加密之后是什么样子。下面是加密脚本所在的目录 *dist* 下的所有文件列表:

```
foo.py

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

dist/foo.py 是加密后的脚本，它的内容如下:

```
from pytransform import pyarmor_runtime
pyarmor_runtime()
__pyarmor__(__name__, __file__, b'\x06\x0f...')
```

和加密脚本在一起的目录 *pytransform* 叫做运行辅助包 ，它是运行加密脚本所必须的。只要这个包能被正常导入进来，加密脚本 *dist/foo.py* 就可以像正常脚本一样被运行。

这是 PyArmor 的一个重要特征：**加密脚本无缝替换原来的脚本**

8.1 如何加密脚本

PyArmor 是怎么加密 Python 源代码呢?

首先把源代码编译成代码块:

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

接着对这个代码块进行如下处理

- 使用 *try...finally* 语句把代码块的代码段 *co_code* 包裹起来:

```
新添加一个头部，对应于 try 语句:

        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

接着是处理过的原始代码段:

        对于所有的绝对跳转指令，操作数增加头部字节数

        加密修改过的所有指令代码

        ...

追加一个尾部，对应于 finally 块:

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

- 添加字符串名称 *__armor_enter__*, *__armor_exit__* 到 *co_consts*
- 如果 *co_stacksize* 小于 4，那么设置为 4
- 在 *co_flags* 设置自定义的标志位 CO_OBFUSCAED (0x80000000)
- 按照上面的方式递归修改 *co_consts* 中的所有类型为代码块的常量

然后把改装后的代码块转换成为字符串，把字符串进行加密，保护其中的常量和字符串:

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

最后生成加密后的脚本，写入到磁盘文件:

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

单纯加密后的脚本就是一个正常的函数调用语句，长得就像这个样子:

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

8.2 如何处理插件

在 PyArmor 中插件主要用于在加密的过程中向脚本中注入代码，例如，使用插件在加密脚本中检查网络时间:

```
pyarmor obfuscate --plugin check_ntp_time foo.py
```

那么为什么不直接把这些代码插入到脚本中呢？主要是因为插件中用到的一些函数必须在加密脚本中才有效，例如，读取加密脚本中许可文件的有效期等。

每一个插件对应一个 Python 的脚本文件，PyArmor 搜索插件的顺序:

- 如果插件指定了绝对路径，那么直接在这个路径下面查找对应的 *.py* 文件
- **如果是相对路径，那么查找顺序是**
  - 当前目录
  - `$HOME/.pyarmor/plugins`
  - `{pyarmor_folder}/plugins`
- 没有找到就抛出异常

如果在加密脚本的时候指定了插件，PyArmor 在加密脚本之前，会逐行扫描源代码的注释去查找插件桩。插件桩分为两种类型:

- 插件定义桩
- 插件调用桩

插件定义桩的格式如下:

```
# {PyArmor Plugins}
```

插件定义桩必须单独占一行，不能有缩进，必须在其他插件调用桩的前面。而且一个脚本中只能有一个插件定义桩，所有插件对应的脚本文件会被原封不动的插入到下面。需要注意的是，如果脚本中没有插件定义桩，任何插件定义代码都不会被注入到加密脚本中。

插件调用桩有三种格式，只要注释行的前缀和下面的任意模式匹配，就是一个插件调用桩:

```
# PyArmor Plugin:
# pyarmor_
# @pyarmor_
```

插件调用桩可以有缩进，可以在模块的任何地方，但是必须在插件调用桩之后，插件调用桩可以有任意多个。

第一种格式又称为 *内联调用桩*，PyArmor 只是简单的把匹配的部分和紧随其后的一个空格删除，只剩下后半部分的代码。例如，在脚本 `foo.py` 有下列内联调用桩:

```
# PyArmor Plugin: check_ntp_time()
# PyArmor Plugin: print('This is plugin code')
# PyArmor Plugin: if sys.flags.debug:
# PyArmor Plugin:     check_something():
```

在加密后的脚本中 `dist/foo.py` ，它们将被替换为:

```
check_ntp_time()
print('This is plugin code')
if sys.flags.debug:
    check_something()
```

只要在命令行指定了插件，这种替换就会发生。如果没有外部的插件脚本，可以在命令行使用特殊的插件名称 `on` 来生效内联插件桩，例如:

```
pyarmor obfuscate --plugin on foo.py
```

而后面两种格式又称为 *条件调用桩*，它们只用于调用插件中函数，并且函数名称还必须和插件名称一致，否则就会被忽略。例如，在使用下面的命令加密脚本:

```
pyarmor obfuscate --plugin check_ntp_time foo.py
```

在 `foo.py` 中，第一个条件调用桩会生效，会被替换成为函数调用语句，但是第二个条件调用桩则不会被替换，还保留原来的注释前缀，因为它调用的函数 `check_multi_mac` 在命令行中没有对应的插件:

```
# pyarmor_check_ntp_time()
# pyarmor_check_multi_mac()


==>
```

(下页继续)

(续上页)

```
check_ntp_time()
# pyarmor_check_multi_mac()
```

第三种格式和第二种类似，只是把注释前缀替换成为 `@`，主要用于注入修饰函数。例如:

```
# @pyarmor_assert_obfuscated(foo.connect)
def login(user, name):
    foo.connect(user, name)

==>

@assert_obfuscated(foo.connect)
def login(user, name):
    foo.connect(user, name)
```

如果在指定插件名称的时候使用了前缀 `@`，则插件脚本只有在被使用到的情况下，才会被注入加密脚本中；如果没有被用到，则会被忽略。例如:

```
pyarmor obfuscate --plugin @check_ntp_time foo.py
```

在脚本 `foo.py` 中必须使用条件调用桩来调用插件函数:

```
# pyarmor_check_ntp_time()
```

而如果脚本中使用的是下面的内联调用桩，那么插件脚本不会被注入，执行的时候会报错，找不到函数 `check_ntp_time`:

```
# PyArmor Plugin: check_ntp_time()
```

8.3 对主脚本的特殊处理

和其他模块不一样，PyArmor 对主脚本有额外的处理:

- 在加密之前，修改主脚本，插入保护代码
- 在加密之后，修改加密脚本，插入引导代码

在加密主脚本之前，PyArmor 会逐行扫描源代码。如果发现下面的一行:

```
# {PyArmor Protection Code}
```

PyArmor 就会把这一行替换成为保护代码。

如果发现了下面这一行:

```
# {No PyArmor Protection Code}
```

PyArmor 就不会在主脚本中插入保护代码。

如果上面两个特征行都没有，那么在看看有没有这样的行:

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

如果有，插入保护代码到这条语句的前面。如果没有，那么不添加保护代码。

默认的保护代码的模板如下:

```
def protect_pytransform():

    import pytransform

    def check_obfuscated_script():
        CO_SIZES = 49, 46, 38, 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():
        def _check_co_key(co, v):
            return (len(co.co_names), len(co.co_consts), len(co.co_code)) == v
        for k, (v1, v2, v3) in {keylist}:
            co = getattr(pytransform, k).{code}
            if not _check_co_key(co, v1):
                raise RuntimeError('unexpected pytransform.py')
            if v2:
                if not _check_co_key(co.co_consts[1], v2):
                    raise RuntimeError('unexpected pytransform.py')
            if v3:
                if not _check_co_key(co.{closure}[0].cell_contents.{code}, v3):
                    raise RuntimeError('unexpected pytransform.py')

    def check_lib_pytransform():
        filename = pytransform.os.path.join({rpath}, {filename})
```

(下页继续)

(续上页)

```
        size = {size}
        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 == {checksum}:
            raise RuntimeError("Unexpected %s" % filename)
    try:
        check_obfuscated_script()
        check_mod_pytransform()
        check_lib_pytransform()
    except Exception as e:
        print("Protection Fault: %s" % e)
        pytransform.sys.exit(1)

protect_pytransform()
```

在加密脚本的时候，PyArmor 会使用真实的值来替换其中的字符串模板 `{xxx}`

如果不想让 PyArmor 添加保护代码，除了在脚本中添加上面所示的标志行之外，也可以使用命令行选项 `--no-cross-protection` ，例如:

```
pyarmor obfuscate --no-cross-protection foo.py
```

主脚本被加密之后，PyArmor 会在最前面插入*引导代码* 。

8.4 如何运行加密脚本

那么，一个普通的 Python 解释器运行加密脚本 *dist/foo.py* 的过程是什么样呢?

上面我们看到 *dist/foo.py* 的前两行是这个样子:

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

这两行叫做*引导代码* ，在运行任何加密脚本之前，它们必须先要被执行。它们有着重要的使命

- 使用 *ctypes* 来装载动态库 *__pytransform*
- 检查授权文件 *dist/license.lic* 是否合法
- 添加三个内置函数到模块 *builtins*: *___pyarmor___* ， *___armor_enter___* ， *___armor_exit___*

最主要的是添加了三个内置函数，这样 *dist/foo.py* 的下一行代码才不会出错，因为它马上要调用函数 *__pyarmor__*:

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

__pyarmor__ 被调用，它的主要功能是导入加密的模块，实现的伪代码如下:

```
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 );
}
```

从现在开始，在整个 Python 解释器的生命周期中

- 每一个函数（代码块）一旦被调用，首先就会执行函数 *__armor_enter__* ，它负责恢复代码块。其实现原理如下所示:

```
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;
}
```

- 因为每一个代码块都被人为的使用 *try...finally* 块包裹了一下，所以代码块执行完之后，在返回上一级

之前，就会调用 *___armor_exit___* 。它会重新加密代码块，同时清空堆栈内的局部变量:

```
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 --;

    // 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;
}
```

8.5 如何打包加密脚本

虽然加密脚本可以无缝替换原来的脚本，但是打包的时候还是存在一个问题:

加密之后所有的依赖包无法自动获取

解决这个问题的基本思路是

1. 使用没有加密的脚本找到所有的依赖文件
2. 使用加密脚本替换原来的脚本
3. 添加加密脚本需要的运行辅助文件到安装包
4. 替换主脚本，因为主脚本会被编译成为可执行文件

PyArmor 提供了一个命令*pack* 可以用来直接打包脚本，它会首先加密脚本，然后调用 PyInstaller 打包，但是在某些情况下，打包可能会失败。这里详细描述了命令*pack* 的内部工作原理，可以帮助定位问题所在，同

时也可以作为自己直接使用 PyInstaller 打包加密脚本的使用手册。

PyArmor 需要 *PyInstaller* 来完成加密脚本的打包工作，如果没有安装的话，首先执行下面的命令进行安装:

```
pip install pyinstaller
```

pyarmor pack 命令的第一步是加密所有的脚本，保存到 `dist/obf`:

```
pyarmor obfuscate --output dist/obf --runtime-mode 0 hello.py
```

第二步是生成 *.spec* 文件，这是 *PyInstaller* 需要的，把加密脚本需要的运行辅助文件也添加到里面:

```
pyinstaller --add-data dist/obf/license.lic:. \
            --add-data dist/obf/pytransform.key:. \
            --add-data dist/obf/_pytransform.*:. \
            -p dist/obf --hidden-import pytransform
            hello.py
```

在 Windows 平台下，命令行中的 : 应该替换为 ;

第三步是修改 *hello.spec*, 在 *Analysis* 之后插入下面的语句，主要作用是打包的时候使用加密后的脚本，而不是原来的脚本:

```
src = os.path.abspath('.')
obf_src = os.path.abspath('dist/obf')

for i in range(len(a.scripts)):
    if a.scripts[i][1].startswith(src):
        x = a.scripts[i][1].replace(src, obf_src)
        if os.path.exists(x):
            a.scripts[i] = a.scripts[i][0], x, a.scripts[i][2]

for i in range(len(a.pure)):
    if a.pure[i][1].startswith(src):
        x = a.pure[i][1].replace(src, obf_src)
        if os.path.exists(x):
            if hasattr(a.pure, '_code_cache'):
                with open(x) as f:
                    a.pure._code_cache[a.pure[i][0]] = compile(f.read(), a.pure[i][1],
↪'exec')
            a.pure[i] = a.pure[i][0], x, a.pure[i][2]
```

最后运行这个修改过的文件，生成最终的安装包:

```
pyinstaller --clean -y hello.spec
```

注解: 必须要指定选项 `--clean` ，否则不会把原来的脚本替换成为加密脚本。

检查一下安装包中的脚本是否已经加密:

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

rm dist/hello/license.lic

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

CHAPTER 9

运行时刻模块 *pytransform*

如果你意识到加密后的脚本对用户来说就是黑盒子，那么有很多事情都可以在 Python 脚本里来做。在这个时候，模块 `pytransform` 会提供很多有用的函数和功能。

模块 `pytransform` 是和加密脚本一起发布，在运行加密脚本之前必须被导入进来，所以，你可以在你的脚本中直接导入这个模块，使用里面的函数。

9.1 内容

exception PytransformError

任何 PyArmor 的 api 失败都会抛出这个异常，传入的参数是错误发生的原因。

get_expired_days()

返回加密脚本的剩余的有效天数。

>0: 剩余的有效天数

-1: 永不过期

注解: 如果加密脚本已经过期，会直接抛出异常并退出。加密脚本里面的任何代码都不会被执行，所以一般情况下这个函数是不会返回 0 的。

---

get_license_info()

获取加密脚本许可证的相关信息。

返回一个字典，可能的键名有：

- expired: 许可过期的日期
- IFMAC：绑定的网卡 MAC 地址
- HARDDISK：绑定的硬盘序列号
- IPV4：绑定的 IPv4 地址
- DATA：自定义的数据，用于扩展认证方式
- CODE：许可注册码

如果许可证没有包含对应的键名，那么其值为 *None*

如果许可文件非法，例如已经过期，会抛出异常 `Exception`

`get_license_code()`

返回字符串，该字符串是生成许可文件时指定的注册码参数。

如果认证文件非法或者无效，抛出异常 `Exception`

`get_user_data()`

返回字符串，该字符串是生成许可文件时指定的 `-x` 参数。如果没有指定该参数，那么返回 *None*

如果认证文件非法或者无效，抛出异常 `Exception`

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

得到当前机器的硬件信息，通过 *hdtype* 传入需要获取的硬件类型（整型），可用的常量如下：

- HT_HARDDISK 返回硬盘序列号
- HT_IFMAC 返回网卡 Mac 地址
- HT_IPV4 返回网卡的 IPv4 地址
- HT_DOMAIN 返回目标设备的域名

无法获取硬件信息会抛出异常 `Exception`

`HT_HARDDISK, HT_IFMAC, HT_IPV4, HT_DOMAIN`

调用*get_hd_info()* 时候 *hdtype* 的可以使用的常量

`assert_armored(`**args*`)`

必须作为修饰函数来使用，用来检查传入的参数列表中的函数是经过加密的。

抛出异常 `Exception` 如果任何传入的一个函数不是 PyArmor 加密过的。

9.2 示例

下面是一些示例，拷贝这些代码到需要加密的脚本里面，然后加密脚本，运行加密脚本查看效果。

显示加密脚本的剩余的有效天数

```
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 for %s will be expired in %d days' % (code, left_days))
except Exception as e:
    print(e)
```

更多内容，请参考使用插件扩展认证方式

注解： 虽然在运行加密脚本的时候 `pytransform.py` 没有被加密，但是它同样被 *PyArmor* 所保护。如果对它进行任何修改，运行加密脚本同样会抛出保护异常。

参考对主脚本的特殊处理

CHAPTER 10

支持的平台列表

PyArmor 的核心函数使用 C 来实现，对于常用的平台和部分嵌入式系统都已经有编译好的动态库。

最常用平台的动态库已经打包在 PyArmor 的安装包里面，只要安装好之后即可使用，参考预安装的动态库清单 。

其他平台的动态库并没有随着安装包发布，参考其他平台的动态库清单 。在这些平台下面，*pyarmor* 会搜索 `~/.pyarmor/platforms/SYSTEM/ARCH` ，其中 `SYSTEM.ARCH` 是一个标准平台名称 。如果还没有下载，那么会自动从远程服务器下载相应平台的动态库。

从 v6.2.0 开始，新增加的超级模式 和以前的动态库不一样，它是直接使用扩展模块 `pytransform` ，预编译的扩展模块列在超级模式预编译扩展模块表

最新的全部支持的动态库详细列表可以参考 https://github.com/dashingsoft/pyarmor-core/blob/master/platforms/index.json

在同一个平台下面可能有多个可用的动态库，分别具备不同的特征，一般在标准平台名称的后面增加一个数字来标识，组成一个唯一的平台 ID。

每一个特征都有自己的标志为:

- 1: 反调试
- 2: JIT，动态代码
- 4: 高级模式
- 8: 超级模式

例如，动态库为 `windows.x86_64.7` 具备特征反调试 ()1，JIT(2)，高级模式 (4)，而 `windows.x86_64.0` 则表示没有任何额外的特征，相应的性能也最高。

在跨平台发布的时候需要注意，特征为 0 的动态库和其他具备特征的动态库是相互不兼容的，为了提高安全性，没有任何特征的动态库使用的加密算法和有特征的库是不相同的。所以，动态库 `windows.x86_64.7` 是无法和 `linux.armv7.0` 共用相同的加密脚本的。

有些平台 *pyarmor* 无法自动识别，但是在其他平台的动态库清单 中有可用的动态库。可以直接下载下来，保存到这个平台的搜索路径 `~/.pyarmor/platforms/SYSTEM/ARCH` 下面。如果不能确定存放的路径，可以使用命令 `pyarmor -d download` 查看，在开始的时候会显示 *pyarmor* 去那里查找动态库。如果需要让 *pyarmor* 自动识别这个平台，请把下面这个脚本的输出发送到 jondy.zhao@gmail.com

```
from platform import *
print('system name: %s' % system())
print('machine: %s' % machine())
print('processor: %s' % processor())
print('aliased terse platform: %s' % platform(aliased=1, terse=1))

if system().lower().startswith('linux'):
    print('libc: %s' % libc_ver())
    print('distribution: %s' % linux_distribution())
```

如果需要在上面没有列出的平台使用 PyArmor，请发送邮件到 jondy.zhao@gmail.com

10.1 标准平台名称

这些名称可用于命令*obfuscate*, *build*, *runtime*, *download* 中来指定平台名称。

- windows.x86
- windows.x86_64
- linux.x86
- linux.x86_64
- darwin.x86_64
- vs2015.x86
- vs2015.x86_64
- linux.arm
- linux.armv6
- linux.armv7
- linux.aarch32
- linux.aarch64
- android.aarch64

- android.armv7 (从 5.9.3 开始支持)
- uclibc.armv7 (从 5.9.4 开始支持)
- linux.ppc64
- darwin.arm64
- freebsd.x86_64
- alpine.x86_64
- alpine.arm
- poky.x86

表 1: 表-1. 预安装的动态库清单

名称	操作系统	CPU 架构	特征	下载	说明
windows.x86	Windows	i686	反调试、JIT、高级模式	_pytransform.dll	使用 i686-pc-mingw32-gcc 交叉编译
windows.x86_64	Windows	AMD64	反调试、JIT、高级模式	_pytransform.dll	使用 x86_64-w64-mingw32-gcc 交叉编译
linux.x86	Linux	i686	反调试、JIT、高级模式	_pytransform.so	使用 GCC 编译
linux.x86_64	Linux	x86_64	反调试、JIT、高级模式	_pytransform.so	使用 GCC 编译
darwin.x86_64	OSX	x86_64, intel	反调试、JIT、高级模式	_pytransform.dylib	使用 CLang 编译（MacOSX10.11）

表 2: 表-2. 其他平台的动态库清单

名称	操作系统	CPU 架构	特征	下载	说明
vs2015.x86	Windows	x86		_pytransform.dll	使用 VS2015 编译
vs2015.x86_64	Windows	x64		_pytransform.dll	使用 VS2015 编译
linux.arm	Linux	armv5		_pytransform.so	32-bit Armv5 (arm926ej-s)
linxu.armv6	Linux	armv6		_pytransform.so	32-bit Armv6 (-marm -march=armv6 -mfloat-abi=hard)
linux.armv7	Linux	armv7	反调试、JIT	_pytransform.so	32-bit Armv7 Cortex-A, hard-float, little-endian
linux.aarch32	Linux	aarch32	反调试、JIT	_pytransform.so	32-bit Armv8 Cortex-A, hard-float, little-endian
linux.aarch64	Linux	aarch64	反调试、JIT	_pytransform.so	64-bit Armv8 Cortex-A, little-endian
linux.ppc64	Linux	ppc64le		_pytransform.so	适用于 POWER8
darwin.arm64	iOS	arm64		_pytransform.dylib	使用 CLang 编译（iPhoneOS9.3sdk）
freebsd.x86_64	FreeBSD	x86_64		_pytransform.so	不支持获取硬盘序列号
alpine.x86_64	Alpine Linux	x86_64		_pytransform.so	可用于 Docker（musl-1.1.21）
alpine.arm	Alpine Linux	arm		_pytransform.so	可用于 Docker（musl-1.1.21），32 bit Armv5T, hard-float, little-endian
poky.x86	Inel Quark	i586		_pytransform.so	使用 i586-poky-linux 交叉编译
android.aarch64	Android	aarch64		_pytransform.so	Build by android-ndk-r20/toolchains/llvm/prebuilt/linux-x86_64/bin/aarch64-linux-android21-clang
android.armv7	Android	armv7l		_pytransform.so	Build by android-ndk-r20/toolchains/llvm/prebuilt/linux-x86_64/bin/armv7a-linux-android21-clang
uclibc.armv7	Linux	armv7l		_pytransform.so	Build by armv7-buildroot-uclibceabihf-gcc

表 3: Table-3. 超级模式预编译扩展模块表

名称	操作系统	CPU 架构	特征	下载	说明
darwin.x86_64.38.py37	MacOSX	x86_64, intel	Anti-Debug, JIT, SUPER	__pytransform_built_py_cl_37m-darwin.so	Built by CLang with MacOSX10.11
darwin.x86_64.38.py27	MacOSX	x86_64, intel	Anti-Debug, JIT, SUPER	__pytransform_built_	Built by CLang with MacOSX10.11

CHAPTER 11

加密模式

PyArmor 提供多种加密模式，以满足安全和性能方面的平衡。通常情况下，默认的加密模式能够满足绝大多数的需要，一般情况下也无不需要对加密模式有详细的了解。仅当对性能有特别的要求或者默认加密模式无法满足需求的时候，才需要改变加密模式，这就需要理解 PyArmor 的不同加密模式。

11.1 超级模式

超级模式是从 PyArmor 6.2.0 开始增加的新特征。在这种模式下，加密脚本中的代码块结构会被改变，并且会对操作码进行映射，是目前安全级别最高的一种模式。在超级模式下面，运行辅助文件只需要一个扩展模块 `pytransform` ，而不在需要其他任何运行辅助文件，加密后的脚本统一为如下格式:

```
from pytransform import pyarmor
pyarmor(__name__, __file__, b'\x0a\x02...', 1)
```

但是超级模式不是所有的 Python 版本都支持的，目前支持的版本有:

- Python 2.7
- Python 3.7

Python 3.5 以及之后的版本会在随后被支持，但是 Python 3.0 ~ 3.4 不会被支持。

使用下面的命令可以启用超级模式加密脚本:

```
pyarmor obfuscate --advanced 2 foo.py
```

更多使用方法参考使用超级模式加密脚本 。

11.2 高级模式

高级模式是从 PyArmor 5.5.0 开始增加的新功能，在这种模式下，加密脚本中代码块的 *PyCode_Type* 的结构会被修改，同时在加密脚本运行的时候，会在 Python 动态库中注入一个钩子函数，来处理这种非正常的代码块。在注入钩子函数的同时，也会检查 Python 解释器是否被修改，如果解释器行为不正常，加密脚本就不会在继续执行。这个特性需要分析汇编指令，目前只在 X86/X64 系列的 CPU 上实现，并且还依赖于编译器。一些使用低版本 GCC 编译的 Python 解释器可能无法被 PyArmor 正确识别，所以目前高级模式默认是没有启用的。

使用下面的命令可以启用高级模式加密脚本:

```
pyarmor obfuscate --advanced 1 foo.py
```

在下一个主版本中高级模式有可能会默认启用。

对于使用老版本的用户来说，升级之前请确保生产环境的 Python 解释器支持高级模式，参考下面文档来评估 Python 解释器

https://github.com/dashingsoft/pyarmor-core/tree/v5.3.0/tests/advanced_mode/README.md

建议已经在生产环境中使用加密脚本的用户在下一个主版本高级模式稳定之后在升级。

注解: 高级模式在试用版本中的限制是每一个模块中的函数（方法）等代码块总数不能超过大约 30 个左右，超过这个限制将无法被加密（但是依旧可以使用普通模式进行加密）。

注解: 如果在使用过程中出现正常的 Python 解释器无法在高级模式下运行，欢迎报告问题到 https://github.com/dashingsoft/pyarmor/issues 或者发送邮件到 jondy.zhao@gmail.com

---

11.3 代码加密模式

一个 Python 文件中，通常有很多个函数（代码块）

- obf_code == 0

不会加密函数对应的代码块

- obf_code == 1 （默认值）

在这种情况下，会加密每一个函数对应的代码块，根据代码包裹模式 的设置，使用不同的加密方式

- obf_code == 2

和 obf_mode 1 类似，但是使用更为复杂的算法来加密代码块（bytecode），所以比前者要慢一些。

11.4 代码包裹模式

- wrap_mode == 0

当包裹模式关闭，代码块使用下面的方式进行加密:

```
0   JUMP_ABSOLUTE            n = 3 + len(bytecode)

3    ...
     ... 这里是加密后的代码块
     ...

n   LOAD_GLOBAL              ? (__armor__)
n+3 CALL_FUNCTION            0
n+6 POP_TOP
n+7 JUMP_ABSOLUTE            0
```

在开始插入了一条绝对跳转指令，当执行加密后的代码块时

1. 首先执行 JUMP_ABSOLUTE, 直接跳转到偏移 n
2. 偏移 n 处是调用一个 PyCFunction *___armor___* 。这个函数的功能会恢复上面加密后的代码块，并且把代码块移动到最开始（向前移动 3 个字节）
3. 执行完函数之后，跳转到偏移 0，开始执行原来的函数。

这种模式下，除了函数的第一次调用需要额外的恢复之外，随后的函数调用就和原来的代码完全一样。

- wrap_mode == 1 （默认值）

当打开包裹模式之后，代码块会使用 *try...finally* 语句包裹起来:

```
LOAD_GLOBALS    N (__armor_enter__)     N = co_consts 的长度
CALL_FUNCTION   0
POP_TOP
SETUP_FINALLY   X (jump to wrap footer) X = 原来代码块的长度

这里是加密的后的代码块

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

这样，当被加密的函数开始执行的时候

1. 首先会调用 *___armor_enter___* 恢复加密后的代码块
2. 然后执行真正的函数代码
3. 在函数执行完成之后，进入 *final* 块，调用 *___armor_exit___* 重新加密函数对应的代码块

在这种模式下，函数的每一次执行完成都会重新加密，每一次执行都需要恢复。

11.5 模块加密模式

- obf_mod == 1 （默认值）

在这种模式下，最终生成的加密脚本如下:

```
__pyarmor__(__name__, __file__, b'\x02\x0a...', 1)
```

其中第三个参数是代码块转换而成的字符串，它通过下面的伪代码生成:

```
PyObject *co = Py_CompileString( source, filename, Py_file_input );
obfuscate_each_function_in_module( co, obf_mode );
char *original_code = marshal.dumps( co );
char *obfuscated_code = obfuscate_algorithm( original_code  );
sprintf( buffer, "__pyarmor__(__name__, __file__, b'%s', 1)", obfuscated_code );
```

- obf_mod == 0

在这种模式下，最终生成的代码如下（最后一个参数为 0）:

```
__pyarmor__(__name__, __file__, b'\x02\x0a...', 0)
```

第三个参数的生成方式和上面的基本相同，除了最后一条语句替换为:

```
sprintf( buffer, "__pyarmor__(__name__, __file__, b'%s', 0)", original_code );
```

默认情况下以上三种加密模式都是启用的，如果要改变加密模式，必须使用工程来加密脚本。具体使用方法请参考工程的 使用不同加密模式

11.6 约束模式

从 PyArmor 5.7.0 开始，引导代码 必须在加密脚本中，并且加密脚本还必须是主脚本。例如，同一个目录下有两个文件 *foo.py* 和 *test.py* ，使用下面的命令加密:

```
pyarmor obfuscate foo.py
```

如果直接往加密后的脚本 *dist/test.py* 中插入 *引导代码*，这个加密脚本就无法使用，因为这个脚本加密的时候没有被指定为主脚本。只能使用下面的命令来插入 *引导代码*:

```
pyarmor obfuscate --no-runtime --exact test.py
```

如果需要在没有加密的脚本中运行*引导代码* ，可以使用一种变通方式。首先加密一个空脚本:

```
echo "" > pytransform_bootstrap.py
pyarmor obfuscate --no-runtime --exact pytransform_bootstrap.py
```

然后在导入这个加密后的空脚本 *import pytransform_bootstrap* 。

从 PyArmor 5.5.6 开始，约束模式有四种形式。

- 模式 1

在此约束模式下，加密脚本必须是下面的形式之一:

```
__pyarmor__(__name__, __file__, b'...')

Or

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

Or

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

例如，下面的这个加密脚本就无法运行，因为有一条额外的语句 *print*:

```
$ cat b.py
from pytransform import pyarmor_runtime
pyarmor_runtime()
__pyarmor__(__name__, __file__, b'...')
print(__name__)

$ python b.py
```

- 模式 2

在此约束模式下，除了加密脚本不能被修改约束外，主脚本必须是加密脚本，并且加密脚本不能被非加密脚本导入和使用，一般用于提高 Python 开发的独立的应用程序的安全性。

例如，使用下面的方式导入使用约束模式 2 加密后的主脚本 *foo* 会出错:

```
$ python -c'import foo'
```

- 模式 3

在此约束模式下，除了满足约束模式 2 之外，加密脚本里面的函数只能被加密后的模块（函数）调用。

- 模式 4

此约束模式和模式 3 基本相似，只是主脚本不需要是加密脚本。一般用于加密 Python 包的部分脚本，以提高加密脚本安全性。

典型的应用是使用约束模式 1 加密 Python 包中 *__init__.py* 和其他需要被外部使用的脚本，而使用约束模式 4 来加密那些只是在包内部使用的脚本。

例如，加密后 *mypkg/__init__.py* 的可访问性如下

```
# mypkg/
#     __init__.py 使用约束模式 1 加密
#     foo.py 使用约束模式 4 加密

# "foo.hello" 不能被没有加密的脚本直接调用
from .foo import hello

# "open_hello" 可以被其他脚本直接调用
def open_hello(msg):
    print('This is public hello: %s' % msg)

# "proxy_hello" 可以被其他脚本直接调用
def proxy_hello(msg):
    print('This is proxy hello: %s' % msg)
    # "foo.hello" 可以被加密脚本 "__init__.py" 调用
    hello(msg)
```

注解: 约束模式 2 和 3 不能用于加密 Python 包，否则加密后的包是无法被非加密的脚本导入的。

注解: 约束模式是针对单个脚本的，不同的脚本可以有不同的约束模式。

从 PyArmor 5.2 开始, 约束模式 1 是默认设置。

如果需要使用其他约束模式加密脚本，通过选项 `--restrict` 指定。例如:

```
pyarmor obfuscate --restrict=2 foo.py
pyarmor obfuscate --restrict=4 foo.py

# For project
pyarmor config --restrict=2
pyarmor build -B
```

如果需要禁用上面全部的约束, 那么使用下面的命令加密脚本:

```
pyarmor obfuscate --restrict=0 foo.py

# For project
pyarmor config --restrict=0
pyarmor build -B
```

详细示例请参考使用约束模式增加加密脚本安全性

CHAPTER 12

加密脚本的性能

运行命令*banchmark* 可以检查加密脚本的性能:

```
pyarmor benchmark
```

下面是输出结果的示例:

```
INFO     Start benchmark test ...
INFO     Obfuscate module mode: 1
INFO     Obfuscate code mode: 1
INFO     Obfuscate wrap mode: 1
INFO     Benchmark bootstrap ...
INFO     Benchmark bootstrap OK.
INFO     Run benchmark test ...
Test script: bfoo.py
Obfuscated script: obfoo.py
-------------------------------------


load_pytransform: 28.429590911694085 ms
init_pytransform: 10.701080723946758 ms
verify_license: 0.515428636879825 ms
total_extra_init_time: 40.34842417122847 ms

import_no_obfuscated_module: 9.601499631936461 ms
```

(下页继续)

(续上页)

```
import_obfuscated_module: 6.858413569322354 ms

re_import_no_obfuscated_module: 0.007263492985840059 ms
re_import_obfuscated_module: 0.0058666674116400475 ms

run_empty_no_obfuscated_code_object: 0.015085716201360122 ms
run_empty_obfuscated_code_object: 0.0058666674116400475 ms

run_one_thousand_no_obfuscated_bytecode: 0.003911111607760032 ms
run_one_thousand_obfuscated_bytecode: 0.005307937181960043 ms

run_ten_thousand_no_obfuscated_bytecode: 0.003911111607760032 ms
run_ten_thousand_obfuscated_bytecode: 0.005587302296800045 ms

----------------------------------------
INFO     Remove test path: .\.benchtest
INFO     Finish benchmark test.
```

其中额外的初始化时间大约是 *40ms* ，这包括装载动态库、初始化动态库和校验授权文件的总时间。

上面结果中，导入加密模块的时间还少于导入正常模块的时间，这主要是因为加密脚本已经被编译成为字节码文件，而原始文件需要额外的时间来进行编译。

这里执行加密函数需要的额外时间一般在 *0.002ms* 左右，也就是执行 *1000* 个函数，加密脚本额外消耗的时间大约为 *2ms* 。

不同的机器可能结果不同，需要根据实际环境下运行结果来进行评估。

查看所有支持的模式:

```
pyarmor benchmark -h
```

组合上面列出的可用加密模式，测试不同加密模式的性能。例如:

```
pyarmor benchmark --wrap-mode 0
```

查看测试命令使用的脚本，使用选项 `--debug` 保留生成的中间文件，所有的中间文件保存在目录 *.benchtest* 下面:

```
pyarmor benchmark --debug
```

CHAPTER 13

PyArmor 的安全性

PyArmor 使用分片式技术来保护 Python 脚本。所谓分片保护，就是单独加密每一个函数，在运行脚本的时候，只有当前调用的函数被解密，其他函数都没有解密。而一旦函数执行完成，就又会重新加密，这是 PyArmor 的特点之一。

例如，下面这个脚本 *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 会首先加密函数 *hello* 和 *sum* ，然后在加密整个模块，进行两次加密。当运行加密的 *hello* 的时候，*sum* 依旧是加密的。*hello* 执行完成之后，会被重新加密，然后才开始解密并执行 *sum* 。

13.1 交叉保护机制

PyArmor 的核心代码使用 *c* 来编写，所有的加密和解密算法都在动态链接库中实现。首先 *__pytransform* 自身会使用 *JIT* 技术，即动态生成代码的方式来保护自己，加密后的 Python 脚本由动态库 *__pytransform* 来

保护，反过来，在加密的 Python 的脚本里面，也会来校验动态库，确保其没有进行任何修改。这就是交叉保护的原理，Python 代码和 *c* 代码相互进行校验和保护，大大提高了安全性。

动态库保护的核心有两点:

1. 用户不能通过修改代码段指令来获得没有授权的使用。例如，将指令 *JZ* 修改为 *JNZ* ，从而使得认证失败可以继续执行
2. 加密 Python 脚本使用的键值不能通过反向跟踪的方式获取到

那么，*JIT* 是如何来做到的呢?

PyArmor 定义了一套自己的指令系统（基于 GNU lightning)，然后把核心函数，主要是获取键值的算法，加解密的过程等，使用自己的指令系统生成数据代码。数据代码存放在一个单独的 *c* 文件中，内容如下:

```
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,
    ...
}
```

这是两个受保护的函数，每一个受保护的函数里面会有很多小函数段。随后编译动态库，计算代码段的校验和，使用这个真实的代码段的校验和替换相关的指令，并对数据代码进行混淆，修改后的文件如下:

```
t_instruction protect_set_key_iv = {
    // function 1, 不混淆
    0x80001,
```

(下页继续)

(续上页)

```
    0x50020,
    ...

    // function 2, 混淆下面的数据指令
    0xXXXXX,
    0xXXXXX,
    ...
}

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

    // function 2, 混淆下面的数据指令
    0xXXXXX,
    0xXXXXX,
    ...
}
```

使用修改后的文件重新编译生成动态库，这个动态库会发布给客户。

当加密脚本运行的时候，每一次调用被保护的函数的时候，就会进入 *JIT* 动态代码保护例程:

1. 读取 *functiion 1* 的数据代码，动态生成 *function 1*
2. 执行 *function 1*:

   ```
   检查代码段的校验和，如果不一致，退出
   检查当前是否有调试器，如果发现，退出
   检查执行时间是否太长，如果执行时间太长，退出
   如果可能的话，清除硬件断点寄存器
   恢复下一个函数 `function 2` 的数据代码
   ```

3. 读取 *functiion 2* 的数据代码，动态生成 *function 2*
4. 重复步骤 2 的操作

这样循环有限次之后，真正受保护的代码才被执行。总之，主要达到的目的是开始执行受保护的代码的时候，不能被调试器中断。

为了在 Python 端保护动态库没有被进行任何修改，在加密主脚本的时候，会插入额外的一段代码来检查和保护动态链接库，详细工作原理参考对主脚本的特殊处理

CHAPTER 14

常见问题

当出现问题的时候，先尝试一些方法看能否解决它。

对于运行 *pyarmor* 出现的问题:

- 查看控制台的输出，有没有什么路径错误，和一些有效的错误信息
- 使用调试选项 `-d` 运行，显示执行堆栈和更多的调试信息。例如:

```
pyarmor -d obfuscate --recurisve foo.py
```

- 设置 Python 的调试标志，例如:

```
PYTHONDEBUG=y pyarmor -d obfuscate --recurisve foo.py

# In Windows
set PYTHONDEBUG=y
pyarmor obfuscate --recurisve foo.py
```

对于运行加密脚本出现的问题:

- 尝试打开 Python 的调试选项查看更多的错误信息。例如:

```
python -d obf_foo.py
```

在任何情况下，打开 Python 的调试开关之后，会在当前目录创建一个日志文件 *pytransform.log* ，里面包含有帮助定位问题的更多信息。

14.1 Segment fault

下面的情况都可能会导致导致程序崩溃

- 使用调试版本的 Python 来运行加密脚本
- 使用 Python 2.6 加密脚本，但是却使用 Python 2.7 来运行加密脚本

如果使用的是 PyArmor v5.5.0 之后的版本，有的机器可能因为不支持高级模式而崩溃。一个快速的解决方案是禁用高级模式，直接修改 pyarmor 安装包路径下面的 *pytransform.py* , 找到函数 *_load_library* ，把禁用高级模式的注释去掉，修改成为下面的样子:

```
# Disable advanced mode if required
m.set_option(5, c_char_p(1))
```

14.2 启动问题

14.2.1 Could not find *__pytransform*

通常情况下动态库 *__pytransform* 在运行辅助包 里面（在 v5.7.0 之前，是和加密脚本在相同的目录下）:

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

首先检查这个文件是否存在。如果文件存在:

- 检查文件权限是否正确。如果没有执行权限，在 Windows 系统会报错:

  ```
  [Error 5] Access is denied
  ```

- 检查 *ctypes* 是否可以直接装载 *__pytransform*:

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

- 如果上面的语句执行失败，尝试在引导代码 中设置运行时刻路径:

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

如果还是有问题，那么请报告 issue

14.2.2 ERROR: Unsupport platform linux.xxx

有些平台 *pyarmor* 无法自动识别，如果在**'其他平台的动态库清单 '__** 中有可用的动态库。可以直接下载下来，保存到这个平台的搜索路径 `~/.pyarmor/platforms/SYSTEM/ARCH` 下面。如果不能确定存放的路径，可以使用命令 `pyarmor -d download` 查看，在开始的时候会显示 *pyarmor* 去那里查找动态库。

如果需要在上面没有列出的平台使用 PyArmor，请发送邮件到 jondy.zhao@gmail.com

14.2.3 /lib64/libc.so.6: version 'GLIBC_2.14' not found

在一些没用 *GLIBC_2.14* 的机器上，会报这个错误。

解决方案是对动态库 *__pytransform.so* 打补丁。

首先查看依赖的版本信息:

```
readelf -V /path/to/_pytransform.so
...

Version needs section '.gnu.version_r' contains 2 entries:
 Addr: 0x00000000000056e8  Offset: 0x0056e8  Link: 4 (.dynstr)
  000000: Version: 1  File: libdl.so.2  Cnt: 1
  0x0010:   Name: GLIBC_2.2.5  Flags: none  Version: 7
  0x0020: Version: 1  File: libc.so.6  Cnt: 6
  0x0030:   Name: GLIBC_2.7  Flags: none  Version: 8
  0x0040:   Name: GLIBC_2.14  Flags: none Version: 6
  0x0050:   Name: GLIBC_2.4  Flags: none  Version: 5
  0x0060:   Name: GLIBC_2.3.4  Flags: none  Version: 4
  0x0070:   Name: GLIBC_2.2.5  Flags: none  Version: 3
  0x0080:   Name: GLIBC_2.3  Flags: none  Version: 2
```

然后把版本依赖项 *GLIBC_2.14* 替换为 *GLIBC_2.2.5*:

- 从 0x56e8+0x10=0x56f8 拷贝四个字节到 0x56e8+0x40=0x5728
- 从 0x56e8+0x18=0x5700 拷贝四个字节到 0x56e8+0x48=0x5730

下面是使用 *xxd* 进行打补丁的脚本命令:

```
xxd -s 0x56f8 -l 4 _pytransform.so | sed "s/56f8/5728/" | xxd -r - _pytransform.so
xxd -s 0x5700 -l 4 _pytransform.so | sed "s/5700/5730/" | xxd -r - _pytransform.so
```

注解: 从 v5.7.9 开始，在 linux/x86_64 平台（例如，CentOS6）已经不需要这个补丁就可以工作了。

在交叉发布的时候，可以在使用下面的命令加密脚本来解决这个问题:

```
pyarmor obfuscate --platform centos6.x86_64 foo.py
```

14.3 加密脚本的问题

14.3.1 'GBK' codec can't decode byte 0xXX

在源代码的第一行指定字符编码，例如:

```
# -*- coding: utf-8 -*-
```

关于源文件字符编码，请参考 https://docs.python.org/2.7/tutorial/interpreter.html#source-code-encoding

14.3.2 Warning: code object xxxx isn't wrapped

这是因为函数中包含特殊情况的跳转指令而引起的，例如，某一个函数编译成为 byte code 之后如果有类似这样的一条指令 *JMP 255*

在加密之前，这条指令占用两个字节。而在加密之后，因为在函数头部插入了额外的指令，这个跳转指令的操作数变成了 *267* 。在 Python3.6 之后，这条指令需要 4 个字节:

```
EXTEND 1
JMP 11
```

遇到这种情况，会使用非包裹方式加密这个函数。当然，当前模块中的其他函数还是被使用包裹模式正常加密的。

如果想要避免这个问题，可以尝试在函数里面增加一些冗余的语句，让跳转长度不要在临界值即可。

后面的版本会考虑解决这个问题，因为一旦修改原来的指令长度，代码块所有的相对跳转、绝对跳转指令都需要调整，情况比较复杂，所以遇到这种情况，暂时忽略了。

注解: 在 v5.5.0 之前，遇到这种情况，这个函数不会被加密。

14.3.3 Error: Try to run unauthorized function

试图使用没有授权的功能。出现这个问题一般是当前目录下面存在 *license.lic*‘或者 *‘pytransform.key* 而导致的认证问题，解决方案是一是删除这些不必要文件，或者升级到 PyArmor 5.4.5 以后的版本。

14.3.4 为什么插件不工作

如果加密脚本的时候指定了插件，但是插件却没有像期望的那样工作。那么首先要检查插件是否被正确注入到主脚本中。例如:

```
# In linux
export PYTHONDEBUG=y
# In Windows
set PYTHONDEBUG=y

pyarmor obfuscate --exact --plugin check_ntp_time foo.py
```

这样会生成一个调试文件 `foo.py.pyarmor-patched` ，检查这个文件，确保插件脚本被正确的插入到里面，并且能被调用。

14.4 运行加密脚本的问题

14.4.1 NameError: name '__pyarmor__' is not defined

原因是引导代码 没有被执行。

当使用模块 *subprocess* 或者 *multiprocessing* ，调用 *Popen* 或者 *Process* 创建新的进程的时候，确保引导代码 在新进程中也得到执行。否则新进程是无法使用加密脚本的。

14.4.2 Marshal loads failed when running xxx.py

当出现这个问题，依次进行下面的检查

1. 检查运行加密脚本的 Python 的版本和加密脚本的 Python 版本是否一致
2. 使用 *python -d* 运行加密脚本，查看更多的错误信息
3. 确保生成许可使用的密钥箱和加密脚本使用的密钥箱是相同的（当运行 PyArmor 的命令时，该命令使用的密钥箱的文件名称会显示在控制台）

14.4.3 _pytransform can not be loaded twice

如果引导代码 被执行两次，就会报这个错误。通常的情况下，是因为在加密模块中插入了引导代码 。因为引导代码在主脚本已经执行过，所以导入这样的加密模块就出现了问题。

注解: 这个限制是从 PyArmor 5.1 引入的，并且在 PyArmor 5.3.5 中已经移除，之后的版本都没有这个限制。

14.4.4 Check restrict mode failed

违反了加密脚本的使用约束，默认情况下，加密脚本是不能被进行任何修改的。更多信息参考约束模式

如果使用 *pack* 命令去打包加密后的脚本，也会出现这个错误提示。命令 *pack* 正确的使用方式是直接打包原始的脚本。

14.4.5 Protection Fault: unexpected xxx

违反了加密脚本的使用约束，默认情况下，下列文件不能进行任何修改：

- pytransform.py
- _pytransform.so/.dll/.dylib

更多信息参考对主脚本的特殊处理

14.4.6 运行脚本时候提示: Invalid input packet

如果加密脚本是在不同的平台下进行加密，参考跨平台发布加密脚本 里面的备注。

在 v5.7.0 之前，检查当前目录下面是否有存在文件 *license.lic* 或者 *pytransform.key* ，如果存在的话，确保它们是加密脚本对应的运行时刻文件。

因为加密脚本会首先在当前目录查看是否存在 *license.lic* 和运行时刻文件 *pytransform.key*, 如果存在，那么使用当前目录下面的这些文件。

其次加密脚本会查看运行时刻模块 *pytransform.py* 所在的目录，看看有没有 *license.lic* 和 *pytransform.key*

如果当前目录下面存在任何一个文件，但是和加密脚本对应的运行时刻文件不匹配，就会报这个错误。

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

通常情况下是因为加密脚本使用的密钥箱和生成许可文件时候使用的密钥箱不一样，例如在试用版本加密脚本，但是在正式版本下面生成许可文件。

通用的解决方法就是重新把加密脚本生成一下，然后在重新生成许可文件。

14.4.8 导入 OpenCV 失败: *NEON - NOT AVAILABLE*

在一些 Raspberry Pi 平台上面，在加密脚本中导入 OpenCV 会报错:

```
************************************************ ****************
* FATAL ERROR: *
* This OpenCV build doesn't support current CPU / HW configuration *
* *
```

(下页继续)

(续上页)

```
* Use OPENCV_DUMP_CONFIG = 1 environment variable for details *
************************************************* ***************

Required baseline features:
NEON - NOT AVAILABLE
terminate called after throwing an instance of 'cv :: Exception'
  what (): OpenCV (3.4.6) /home/pi/opencv-python/opencv/modules/core/src/system.cpp:538:␣
↪error:
(-215: Assertion failed) Missing support for required CPU baseline features. Check␣
↪OpenCV build
configuration and required CPU / HW setup. in function 'initialize'
```

当前的解决方案使用选项 `--platform=linux.armv7.0` ，例如:

```
pyarmor obfuscate --platform linux.armv7.0 foo.py
pyarmor build --platform linux.armv7.0
pyarmor runtime --platform linux.armv7.0
```

另一种解决方案是设置环境变量 *PYARMOR_PLATFORM=linux.armv7.0* ，例如:

```
PYARMOR_PLATFORM=linux.armv7.0 pyarmor obfuscate foo.py
PYARMOR_PLATFORM=linux.armv7.0 pyarmor build

或者,

export PYARMOR_PLATFORM=linux.armv7.0
pyarmor obfuscate foo.py
pyarmor build
```

14.5 打包加密问题

14.5.1 No module name pytransform

在使用 *pyarmor pack* 打包的时候报这个错误:

- 确认命令行指定的脚本是没有加密的
- 使用选项 `--clean` 清除缓存的 *myscript.spec*:

  ```
  pyarmor pack --clean foo.py
  ```

14.6 PyArmor 注册问题

14.6.1 购买的私有密钥箱没有起作用

使用私有密钥箱加密的脚本，依然可以在试用版本生成的许可证下运行:

- 确认命令 *pyarmor register* 能显示正确的注册信息
- 确认全局密钥箱文件 *~/.pyarmor_capsule.zip* 和注册文件 *pyarmor-regfile-1.zip* 中的 *.pyarmor_capsule.zip* 是同一个文件
- 重新启动系统

14.7 已知的问题

14.7.1 交叉发布的脚本无法运行

从 v5.6.0 到 v5.7.0 这几个版本，交叉发布功能有一个问题。在 Windows / Ubuntu / MacOS 等上面使用跨平台加密方式加密的脚本，拷贝到下面的任一平台都不能正常运行:

```
armv5, android.aarch64, ppc64le, ios.arm64, freebsd, alpine, alpine.arm, poky-i586
```

14.8 其他问题

14.8.1 在 Linux 下面无法获取硬盘序列号

获取硬盘序列号需要超级用户权限，首先确认有相关权限。

其次检查一下目录 */dev/disk/by-id* ，这里会列出已经挂载的硬盘的接口和序列号。如果这里没有文件，那么是无法获取硬盘序列号信息的。在 Docker 环境里面的话，确保运行 docker 的时候挂载硬盘设备。

目前支持的硬盘接口包括 IDE，SCSI 以及 NVME 固态硬盘，对于其他接口的尚不支持。

CHAPTER 15

软件许可

PyArmor 是一个共享软件。试用版永不过期，试用版的限制是

- 最大可加密的脚本大小（编译成为 *.pyc* 之后）是 **32768** 个字节
- 在试用版中中生成的加密脚本不是私有的，也就是说，其他任何人也可以为这些加密脚本生成新的许可文件。
- 使用高级模式加密脚本的时候，每一个模块最多可以有大约 32 个函数（代码块）
- 任何人都可以使用本软件加密非商业用途的 Python 脚本，未经许可不得用于商业用途。

关于加密脚本的许可文件，可以参考加密脚本的许可文件

生成私有的加密脚本和加密任意大小的脚本需要购买下列任意一种许可证。

PyArmor 有两种类型的许可证:

a. 个人用户许可，适用于产品的所有权为个人所有。个人用户购买一个许可证可以在自己所有的计算机和相关硬件设备上使用。

   个人用户许可证允许使用本软件加密任何属于自己的 Python 脚本，为加密脚本生成私有许可文件，发布加密后的脚本和必要的辅助文件到任何其他设备。

   个人用户许可证不允许加密产权属于法人（公司）的 Python 脚本。

b. 企业用户许可，适用于产品的所有权为法人（公司）所有。企业用户购买一个软件许可证可以在同一个产品系列的各个项目中使用。

   产品系列包括同一个产品升级之后的所有版本。

企业用户许可证允许使用本软件在任何设备上，加密属于该产品系列的 Python 脚本，为加密脚本生成私有许可文件，发布加密后的脚本和必要的辅助文件到任何其他设备。

除非有许可人的许可，否则企业用户许可证不可以用于完全独立的其他产品系列。如果需要在其他产品系列中使用，必须为其他产品单独购买软件许可。

不管那一种许可方式，本软件都只可用于保护产品本身，不允许应用于产权不属于被授权人的 Python 脚本。例如，开发平台工具 PyCharm 购买的 PyArmor 许可证只能用于加密和保护 PyCharm 自身，而不能使用 PyArmor 的许可为其他使用 PyCharm 进行开发的 Python 脚本提供加密功能。

15.1 购买

使用微信或者支付宝通过下面的链接购买许可

https://pyarmor.dashingsoft.com/cart/order.html

使用信用卡或者 PayPal 通过下面的连接购买许可

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

支付成功之后注册文件会自动通过电子邮件发送过去。注册文件是一个压缩文件，里面包含 3 个文件:

- README.txt
- license.lic (注册码)
- .pyarmor_capsule.zip (私有密钥箱)

当收到包含注册文件的邮件之后，保存附件为 *profile-regfile-1.zip* ，然后使用下面的命令生效注册文件:

```
pyarmor register pyarmor-regfile-1.zip
```

运行下面的命令查看注册信息

```
pyarmor register
```

注册码生效之后，使用试用版本加密的脚本需要全部重新加密。

如果你使用的是 PyArmor 5.6 之前的版本，使用下面的方式注册:

1. 解压注册文件
2. 拷贝解压后的"license.lic" 到 PyArmor 的安装目录下面
3. 拷贝解压后的".pyarmor_capsule.zip" 到用户的 HOME 目录

重要: 软件注册码永久有效，可以一直使用，但是不能转接或者租用。另外，注册码也有可能在某一个新版本无法使用，虽然到现在为止，注册码可以应用于所有版本。

15.2 Q & A

1. Single PyArmor license purchased can be used on various machines for obfuscation? or its valid only on one machine? Do we need to install license on single machine and distribute obfuscate code?

   It can be used on various machines, but one license only for one product.

2. Single license can be used to obfuscate Python code that will run various platforms like windows, various Linux flavors?

   For all the features of current version, it's yes. But in future versions,
   I'm not sure one license could be used in all of platforms supported by
   PyArmor.

3. How long the purchased license is valid for? is it life long?

   It's life long. But I can't promise it will work for the future version of PyArmor.

4. Can we use the single license to obfuscate various versions of Python package/modules?

   Yes, only if they're belong to one product.

5. Is there support provided in case of issues encountered?

   Report issue in github or send email to me.

6. Does Pyarmor works on various Python versions?

   Most of features work on Python27, and Python30~Python38, a few features
   may only work for Python27, Python35 later.

7. Are there plans to maintain PyArmor to support future released Python versions?

   Yes. The goal of PyArmor is let Python could be widely used in the commercial softwares.

8. What is the mechanism in PyArmor to identify whether modules belong to same product? how it identifies product?

   PyArmor could not identify it by itself, but I can check the obfuscated scripts to find which registerred user distributes them. So I can find two products are distributed by one same license.

9. If product undergoes revision ie. version changes, can same license be used or need new license?

   Same license is OK.

索引

A

assert_armored() (🄵置函数), 86

G

get_expired_days() (🄵置函数), 85
get_hd_info() (🄵置函数), 86
get_license_code() (🄵置函数), 86
get_license_info() (🄵置函数), 85
get_user_data() (🄵置函数), 86

P

PytransformError, 85