**Хабы**: Блог компании Сбер, Python, Анализ и проектирование систем, Программирование, Исследования в IT

**Бинарники из Python-файлов: Nuitka-компилятор, обзор и небольшое исследование**



Здравствуйте, дорогие хабровчане. Сегодняшняя статья — результат моего небольшого исследования. Я хочу показать, как компилировать бинарные модули расширения (.so) из python-файлов, чем они будут отличаться и как с ними работать. Делать это мы будем при помощи компилятора Nuitka. Он наиболее известен тем, что с его помощью можно создавать исполняемые файлы (.exe) для Windows. Однако, кроме того, он позволяет создавать и бинарные модули python. Всех, кому это интересно, прошу под кат.

<cut /> Текст кнопки: «Узнать все подробности!»

## https://habrastorage.org/webt/fa/ge/er/fageerhwrjnof_bq5wi9tg3o_h4.png

## Обзор компилятора Nuitka

Так как [Nuitka](https://github.com/Nuitka/Nuitka) сперва генерирует Cи-код и потом компилирует бинарник с помощью С/С++-компилятора, то перед началом работы нам нужны:

* С/С++-компилятор. Я всё делаю на linux, поэтому в моём случае gcc не ниже версии 5.1;
* Python-версии 2.6, 2.7 или 3.3-3.10, в моём случае python 3.10.

Подробнее см. [зависимости](https://github.com/Nuitka/Nuitka#requirements).

**Интересный факт**: название nuitka получилось из имени жены разработчика пакета (Kay Hayen), её зовут Анна, логика примерно такая: Anna –> Annuitka –> Nuitka, Annuitka, — это вроде должно звучать как Анютка, сам разработчик из Германии.

Почему nuitka это именно компилятор? Дело в том, что python-скрипт действительно компилируется в бинарный код (хотя на самом деле основную роль здесь играет компилятор C/C++).

Самое главное достоинство этого инструмента в том, что бинарный модуль можно импортировать и использовать как обычный python-скрипт. Есть, конечно, и нюансы. Например, невозможность посмотреть исходный код модуля. А ещё — AST-дерево и имя модуля (как будет показано дальше, именно имя модуля, а не название файла).

[Репозиторий с исходным кодом — на GitHub](https://github.com/cdies/nuitka_module).

В качестве примера будет использоваться очень простой python-скрипт (файл [py/bin\_module.py](https://github.com/cdies/nuitka_module/blob/main/py/bin_module.py)):

text = 'Find me!'

class Dummy:

password = 'qweasd123'

def main(self):

print('Hello world!')

Я буду всё делать в окружении [conda](https://www.anaconda.com/), тем более что в нём же можно установить gcc-компилятор:

conda create -n nuitka\_module python=3.10 gcc">=5.1" nuitka=1.3.6 ordered-set">=3.0" -c conda-forge

**Примечание**: не забудьте его активировать conda activate nuitka\_module, также пакет может быть установлен через pip.

В nuitka можно компилировать модуль вместе с зависимостями. На мой взгляд, это не очень удобно, ведь если что-то в импортируемом скрипте поменяется, то нужно будет перекомпилировать все файлы. Куда проще компилировать каждый отдельный модуль независимо (хотя это не всегда так, бывает и строго наоборот). Я запускаю компилятор с такими ключами:

nuitka --module --nofollow-imports --static-libpython=no --remove-output --no-pyi-file --output-dir=so --jobs=4 py/bin\_module.py

Рассмотрим подробнее ключи:

* --module — ключ для компиляции бинарного модуля;
* --nofollow-imports — не компилирует зависимые модули;
* --static-libpython=no — не нужно ничего брать из conda;
* --remove-output — удаляет после работы сгенерированные из python-модуля Си-файлы;
* --no-pyi-file — после компиляции создаётся текстовый файл .pyi, в котором имеется информация о структуре исходного python-модуля, он нам тоже не нужен;
* --output-dir=so — сохраняет бинарный модуль в папке so;
* --jobs=4 — компилирует в 4 потока.

**Важно обратить внимание**, что скомпилированный модуль можно будет запустить только в той же версии python, в какой он и был скомпилирован. Это происходит из-за того, что модуль загружается конкретной версией CPython, в которой был создан. Т. е. если я скомпилировал бинарник в python 3.10, значит, и запускать его надо в python 3.10. В других версиях python будут генерироваться ошибки при попытке его импортировать.

На выходе получаем бинарный модуль [so/bin\_module.cpython-310-x86\_64-linux-gnu.so](https://github.com/cdies/nuitka_module/tree/main/so).

## AST-дерево

Абстрактное синтаксическое дерево, или [AST-дерево](https://docs.python.org/3/library/ast.html), — это, в сущности, полное описание внутренней логики python-модуля. Как ни странно, но тут объяснение самое простое — конечный бинарный модуль в nuitka компилируется из Си-файлов, т. е. питоновского абстрактного дерева в нём нет по определению. Но давайте проверим это (файл [ast\_tree.py](https://github.com/cdies/nuitka_module/blob/main/ast_tree.py)):

import ast

from pathlib import Path

def pretty\_print(text):

print(f'{text[:160]} \n...\n {text[-100:]}')

so\_file\_module = Path('so/bin\_module.cpython-310-x86\_64-linux-gnu.so').read\_bytes()

so\_file\_module = str(so\_file\_module)

so\_dump = ast.dump(ast.parse(so\_file\_module), indent=2)

py\_file\_module = Path('py/bin\_module.py').read\_text()

py\_dump = ast.dump(ast.parse(py\_file\_module), indent=2)

В бинарном модуле, скомпилированном C/C++-компилятором, нет абстрактного дерева в том виде, в каком мы его ожидаем увидеть. Вместо него там чистый байт-код:

pretty\_print(so\_dump)

**показать...**

Module(

body=[

Expr(

value=Constant(value=b'\x7fELF\x02\x01\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x03\x00>\x00\x01\x00\x00\x00\x00\x00\x00\x00\x00

...

\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'))],

type\_ignores=[])

А вот в исходном питоновском файле содержится полноценное абстрактное дерево:

pretty\_print(py\_dump)

**показать...**

Module(

body=[

Assign(

targets=[

Name(id='text', ctx=Store())],

value=Constant(value='Find me!')),

ClassDef(

name='Dummy',

...

keywords=[]))],

decorator\_list=[])],

decorator\_list=[])],

type\_ignores=[])

## Что в имени тебе моём? Название бинарного модуля

В большинстве случаев не стоит менять название бинарного модуля, т. к. если его переназвать, то можно получить ошибку на этапе импорта.

**Примечание**: изменить название можно, если удалить текст между точками, нельзя менять первое слово перед точкой и расширение. Вот, например, вполне валидное изменение:

* bin\_module.cpython-310-x86\_64-linux-gnu.so — исходное название;
* bin\_module.so — валидное изменение.

Имя модуля завязано на название бинарного файла при обычном импорте. Однако нельзя менять именно имя модуля, название самого файла, как будет показано ниже, изменить вполне можно. Для этого поменяем название бинарника на [custom\_name\_module.so](https://github.com/cdies/nuitka_module/tree/main/so) и воспользуемся библиотекой [importlib](https://docs.python.org/3/library/importlib.html) (файл [import\_lib.py](https://github.com/cdies/nuitka_module/blob/main/import_lib.py)):

import importlib.util

import sys

from pathlib import Path

module\_name = 'bin\_module' # Название модуля без изменений

file\_path = Path('so/custom\_name\_module.so') # А вот имя файла изменено

spec = importlib.util.spec\_from\_file\_location(module\_name, file\_path)

module = importlib.util.module\_from\_spec(spec)

sys.modules[module\_name] = module

spec.loader.exec\_module(module)

Как видно по атрибутам модуля, он вполне себе загрузился:

print(dir(module))

**показать...**

['Dummy', '\_\_builtins\_\_', '\_\_cached\_\_', '\_\_compiled\_\_', '\_\_doc\_\_', '\_\_file\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_', 'text']

А вот подобная попытка приведёт к ошибке:

from so import custom\_name\_module

**показать...**

ImportError Traceback (most recent call last)

line 1

----> 1 from so import custom\_name\_module

ImportError: dynamic module does not define module export function (PyInit\_custom\_name\_module)

То, что можно менять название питоновского файла как угодно, это и так ясно, в этом случае можно примеров не приводить.

## Декомпиляция

Рассмотрим подробнее, что находится внутри бинарника и так ли он хорошо всё скрывает с точки зрения безопасности. Автор, конечно, не мастер реверс-инжиниринга, да и статья не про то, но всё же посмотрим, что можно сделать. Для начала воспользуемся декомпилятором (или любым hex-редактором, я буду использовать [ghidra](https://github.com/NationalSecurityAgency/ghidra)) в самом простом виде. Найдём место, где лежит переменная text, посмотрим, что там ещё есть, и попробуем поменять вывод с Hello world! на Some change!.

**Примечание**: более подробно можно посмотреть в [этом](https://www.youtube.com/watch?v=2xUqLLQu0NI) примере.

Рассмотрим подробнее часть hex-кода бинарного модуля из декомпилятора ghidra, стрелками <------ отмечены интересующие нас строки (файл [part\_of\_hex\_code\_decompiler.txt](https://github.com/cdies/nuitka_module/blob/main/compare/part_of_hex_code_decompiler.txt)):

**показать...**

001903c9 75 48 65 6c ds "uHello world!" <------

6c 6f 20 77

6f 72 6c 64

001903d7 61 5f 5f 64 ds "a\_\_doc\_\_"

6f 63 5f 5f 00

001903e0 61 5f 5f 66 ds "a\_\_file\_\_"

69 6c 65 5f

5f 00

001903ea 61 5f 5f 73 ds "a\_\_spec\_\_"

70 65 63 5f

5f 00

001903f4 61 6f 72 69 ds "aorigin"

67 69 6e 00

001903fc 61 68 61 73 ds "ahas\_location"

5f 6c 6f 63

61 74 69 6f

0019040a 61 5f 5f 63 ds "a\_\_cached\_\_"

61 63 68 65

64 5f 5f 00

00190416 75 46 69 6e ds "uFind me!" <------

64 20 6d 65

21 00

00190420 61 74 65 78 ds "atext"

74 00

00190426 61 6d 65 74 ds "ametaclass"

61 63 6c 61

73 73 00

00190431 54 PUSH RSP

00190432 00 00 ADD byte ptr [RAX],AL

00190434 00 00 ADD byte ptr [RAX],AL

00190436 61 5f 5f 70 ds "a\_\_prepare\_\_"

72 65 70 61

72 65 5f 5f 00

00190443 54 PUSH RSP

00190444 02 00 ADD AL,byte ptr [RAX]

00190446 00 00 ADD byte ptr [RAX],AL

00190448 61 44 75 6d ds "aDummy"

6d 79 00

0019044f 54 PUSH RSP

00190450 00 00 ADD byte ptr [RAX],AL

00190452 00 00 ADD byte ptr [RAX],AL

00190454 61 5f 5f 67 ds "a\_\_getitem\_\_"

65 74 69 74

65 6d 5f 5f 00

00190461 75 25 73 2e ds "u%s.\_\_prepare\_\_() must return a mapping, not"

5f 5f 70 72

65 70 61 72

00190491 61 5f 5f 6e ds "a\_\_name\_\_"

61 6d 65 5f

5f 00

0019049b 75 3c 6d 65 ds "u<metaclass>"

74 61 63 6c

61 73 73 3e 00

001904a8 61 5f 5f 6d ds "a\_\_module\_\_"

6f 64 75 6c

65 5f 5f 00

001904b4 61 44 75 6d ds "aDummy"

6d 79 00

001904bb 61 5f 5f 71 ds "a\_\_qualname\_\_"

75 61 6c 6e

61 6d 65 5f

001904c9 61 71 77 65 ds "aqweasd123" <------

61 73 64 31

32 33 00

001904d4 61 70 61 73 ds "apassword" <------

73 77 6f 72

64 00

001904de 61 6d 61 69 ds "amain"

6e 00

001904e4 75 44 75 6d ds "uDummy.main"

6d 79 2e 6d

61 69 6e 00

001904f0 75 62 69 6e ds "ubin\_module.py"

5f 6d 6f 64

75 6c 65 2e

001904ff 75 3c 6d 6f ds "u<module bin\_module>"

64 75 6c 65

20 62 69 6e

Здесь можно подметить структуру класса, его функции и их названия, содержимое и имена переменных. Как и ожидалось, по тексту Find me! удалось найти содержимое переменной password. Впрочем, методом перебора вполне можно было бы найти подходящую версию python и исследовать этот загруженный модуль на структуру и содержимое переменных.

Попробуем пропатчить, т. е. изменить вывод бинарного модуля с Hello world! на Some change! (для простоты длина строк совпадает). Это достаточно просто, необходимо лишь изменить несколько байтов, например:

hex-код текстов из онлайн-[конвертера](https://codebeautify.org/string-hex-converter):

Hello world! -> 48 65 6c 6c 6f 20 77 6f 72 6c 64 21

Some change! -> 53 6f 6d 65 20 63 68 61 6e 67 65 21

Исходный код (файл [so/bin\_module.cpython-310-x86\_64-linux-gnu.so](https://github.com/cdies/nuitka_module/tree/main/so)):

001903c9 75 48 65 6c ds "uHello world!"

6c 6f 20 77

6f 72 6c 64

Изменённый код (файл [compare/bin\_module.cpython-310-x86\_64-linux-gnu.so](https://github.com/cdies/nuitka_module/tree/main/compare)):

001903c9 75 53 6f 6d ds "uSome change!"

65 20 63 68

61 6e 67 65

При попытке его импортировать получаем ошибку:

from compare import bin\_module

**показать...**

Error, corrupted constants object Aborted (core dumped)

Выходит, что напрямую пропатчить модуль не получится. Ну что ж, у нас есть исходник, изменим его, скомпилируем в nuitka и посмотрим на отличия.

text = 'Find me!'

class Dummy:

password = 'qweasd123'

def main(self):

print('Some change!')

Для корректного сравнения (чтобы не плодить лишние изменения) заменим только текст в коде, названия файлов остаются прежними. Hex-код исходного модуля с Hello world! сохраним в файл [compare/hex\_code\_hello\_world.txt](https://github.com/cdies/nuitka_module/blob/main/compare/hex_code_hello_world.txt), а hex-код с текстом Some change! в [compare/hex\_code\_some\_change.txt](https://github.com/cdies/nuitka_module/blob/main/compare/hex_code_some_change.txt). Для сравнения двух файлов можно использовать git, например:

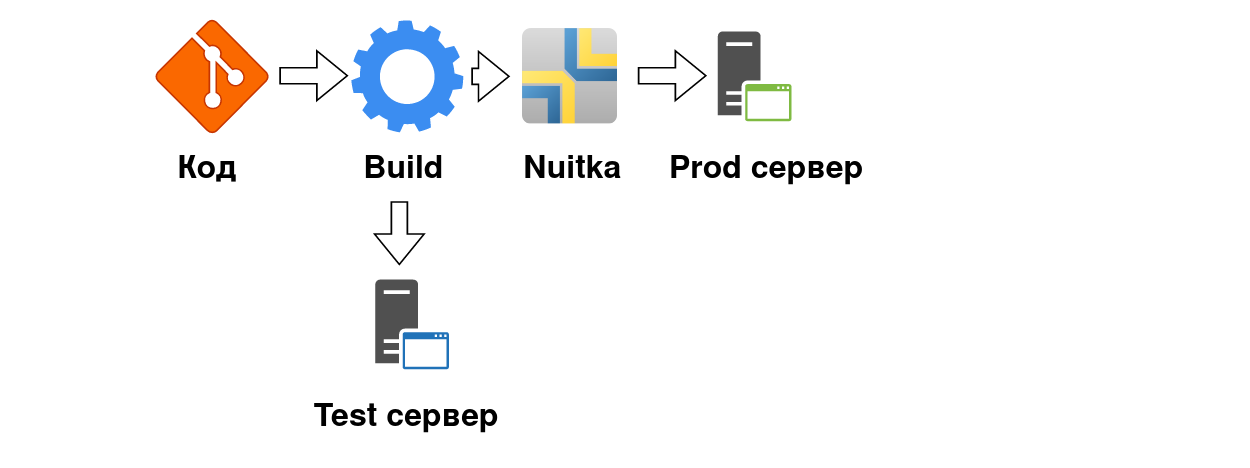
git diff --no-index compare/hex\_code\_hello\_world.txt compare/hex\_code\_some\_change.txt

В итоге получим отличия в **двух** местах (!), а не в одном ([diff.txt](https://github.com/cdies/nuitka_module/blob/main/compare/diff.txt)):

|  |  |  |
| --- | --- | --- |
|  | **hex\_code\_hello\_world.txt** | **hex\_code\_some\_change.txt** |
| Какая-то хэш-сумма | 10d1 8c48 | 33d8 240e |
| hex-код текста | 536f 6d65 2063 6861 6e67 6521 (Some change!) | 4865 6c6c 6f20 776f 726c 6421 (Hello world!) |

Как видно из таблицы, в первый раз, когда мы делали патч, мы поменяли только текст. Но, судя по всему, для значений переменных рассчитывается ещё какая-то хэш-сумма, в итоге из-за этого и получилась ошибка при импорте.

## Заключение



На диаграмме выше показан CI/CD-процесс в самом обобщённом виде. На нём компиляция бинарных модулей расположена перед prod-сервером, т. к. на тестовом сервере критически важен исходный код, а вот на проде его лучше бы скрыть (это зависит от требований безопасности к проекту).

Текущая версия nuitka на данный момент — 1.3, т. е. проект достаточно хорошо работает и уже вышел на релизы. К тому же, как сказано на сайте разработчика, nuitka способна [ускорить код до 3-x раз](https://www.nuitka.net/pages/overview.html#now) — вполне себе неплохой бонус.

В качестве итога скажу, что разобраться в том, как работает бинарный модуль, можно. Но для этого совершенно точно потребуется значительно больше квалификаций и навыков, чем если бы просто можно было поменять чистый python-код.

На этом всё, спасибо за внимание!