Investigations Into Hardening Python Source Code

- Investigations Into Hardening Python Source Code
- 1. Background
  - 1.1 Pyarmor
  - 1.2 Nuitka
  - 1.3 Cython
- 2. Installation
  - 2.1 Install pyarmor
  - 2.2 Install Nuitka
- 3. Setup
- 4. Usage
  - 4.1 Pyarmor
  - 4.2 Nuitka
  - 4.3 Pyarmor with Nuitka
  - 4.5 Cython
  - 4.6 Pyarmor With Cython
- 5. Galaxy Brain Approach

1. Background

During discussions involving the distribution of our services into a production environment, it was discovered that some sort of protection was needed for those services developed in Python. By default, Python source code is exposed to the outside world which, from a security standpoint, is very dangerous. Investigating into how we could harden our source code led to several different solutions using existing Python modules/utilities. The ideal solution would obfuscate the source code, be hardened against decompilers, portable, and debuggable. Pyarmor, Cython, and Nuitka were selected as possible tools to make this possible.

1.1 Pyarmor

Overview

Pyarmor is a utility for obfuscating python source code.

Basic Technical Overview

Pyarmor obfuscates code in a 2-step process, but getting into the weeds of how that works isn't important to us. A better way is to show its power is through a simple example. Here we have a small one-liner 'Hello World' program in python with the following code:

```
print('Hello, World!)
```

When obfuscated with pyarmor it would look something like this:

```
from pytransform import pyarmor_runtime
pyarmor_runtime()
__pyarmor__(__name__, __file__,
b'\x50\x59\x41\x52\x4d\x4f\x52\x00\x00\x03\x08\x00\x55\x0d\x0d\
x0a\x04\x00\x00\x00\x00\x00\x00\x00\x01\x00\x00\x00\x40\x00\x00\x00\xd5\x00\x00\x0
0\x00\x00\x00
\x18\xf4\x63\x79\xf6\xaa\xd7\xbd\xc8\x85\x25\x4e\x4f\xa6\x80\x72\x9f\x00\x00\x00\x
00\x00\x00\x0
0\x00\xec\x50\x8c\x64\x26\x42\xd6\x01\x10\x54\xca\x9c\xb6\x30\x82\x05\xb8\x63\x3f\
xb0\x96\xb1\x
97\x0b\xc1\x49\xc9\x47\x86\x55\x61\x93\x75\xa2\xc2\x8c\xb7\x13\x87\xff\x31\x46\xa5
\x29\x41\x9d\
xdf\x32\xed\x7a\xb9\xa0\xe1\x9a\x50\x4a\x65\x25\xdb\xbe\x1b\xb6\xcd\xd4\xe7\xc2\x9
7\x35\xd3\x3e
\xd3\xd0\x74\xb8\xd5\xab\x48\xd3\x05\x29\x5e\x31\xcf\x3f\xd3\x51\x78\x13\xbc\xb3\x
3e\x63\x62\xc
a\x05\xfb\xac\xed\xfa\xc1\xe3\xb8\xa2\xaa\xfb\xaa\xbb\xb5\x92\x19\x73\xf0\x78\xe4\
x9f\xb0\x1c\x
7a\x1c\x0c\x6a\xa7\x8b\x19\x38\x37\x7f\x16\xe8\x61\x41\x68\xef\x6a\x96\x3f\x68\x2b
\xb7\xec\x60\
x39\x51\xa3\xfc\xbd\x65\xdb\xb8\xff\x39\xfe\xc0\x3d\x16\x51\x7f\xc9\x7f\x8b\xbd\x8
8\x80\x92\xfe
\xe1\x23\x61\xd0\xf1\xd3\xf8\xfa\xce\x86\x92\x6d\x4d\xd7\x69\x50\x8b\xf1\x09\x31\x
cc\x19\x15\xe
f\x37\x12\xd4\xbd\x3d\x0d\x6e\xbb\x28\x3e\xac\xbb\xc4\xdb\x98\xb5\x85\xa6\x19\x11\
x74\xe9\xab\x
df', 1)
```

1.2 Nuitka

Overview

Nuitka is a python module that can compile Python code to an executable. It translates Python code into a C program that then is linked against libpython to execute code the same as CPython. Nuitka can use a variety of C compilers including gcc, clang, MinGW64, Visual Studio 2019+, and clang-cl to convert your Python code to C.

1.3 Cython

Overview

- Cython helps programmers to boost the performance of code with C like performance. Developers can load and use the extension modules directly in the Python code through the `Import` statement. Cython is two closely related things. It is a programming language that blends Python with the static type system of C and C++ and it is a compiler that translates Python source code into efficient C or C++ source code. This source can then be compiled into a Python extension module or a standalone executable Cython's power comes from the way it combines Python and C. It feels like Python while providing easy access to C. Cython is situated between high-level Python and low-level C. One might call it a Creol Programming Language. Both languages are mainstream but they are typically used in different domains. Given their differences, Cython's beauty is this: It combines Python's expressiveness and dynamism with C's bare metal performance while still feeling like Python.

Features

- Cython has expensive goals, first and foremost being full Python compatibility. It has also acquired features that are specific to its unique position between Python and C making Cython easier to use, more efficient, and more expressive. Some of these Cython only features are the following:
    1. Conversion between C types and Python types
    2. Specialized syntax to ease wrapping and interfacing with C and C++
    3. Automatic static type inference for certain code parts
    4. Buffer specific syntax for first class buffer support
    5. Typed memory views
    6. Thread based parallelism using the p-range function

Basic Technical Overview

- Let's talk about some of the technical things involved in the understanding of Cython. Cython source file names consist of the name of the module followed by a `.pyx` extension. For example, a module called `primes` would have a source file named `primes.pyx` Cython to a `.c` file and in the second stage, the `.c` file is compiled by a C compiler to a `.so` file or a `.pyd` file on Windows. Once you have written your `.pyx` file, there are several ways to turn it into an extension module. Commonly, there are two ways of compiling from the command line. The first is by using the Cython command which takes a `.py/.pyx` file and compiles it into a C/C++ file. The second method uses the Cythonize command which also takes a `.py/.pyx` file and compiles it into a C/C++ file however, it also compiles the C and C++ files into an extension module which is directly importable from Python.

2. Installation

2.1 Install pyarmor

Pyarmor can be downloaded from sources but I found that installing it through `pip/pip3` was what worked for me. I did however, have to specify the full path to pyarmor to execute it on the command line as shown below.

```
python3 -m pip3 install --user pyarmor
```

The `--user` flag was necessary in my case in order to have the correct permissions to install pyarmor without using `sudo` which is discouraged by pip.

2.2 Install Nuitka

```
python3 -m pip3 install --user nuitka
```

3. Setup

For my test, I created a directory on my Desktop called `Nuitka_test_obfusc\` with the following structure:

```
Nuitka_test_obfusc/
    main.py
    math_funcs/
        __init__.py
        math_funcs.py
```

I left `__init__.py` empty and placed the following code inside of `math_funcs.py`

```
class Math:
    def __init__(self):
        pass

    def add(self, x, y):
        return x + y

    def subtract(self, x, y):
        return x - y

    def multiply(self, x, y):
        return x * y

    def divide(self, x, y):
        return x / y

    def exponent(self, x, y=2):
        return x ** y

    def absVal(self, x):
        return x if x >=0 else x * -1

    def factorial(self, n):
        if n == 0 or n == 1:
            return 1

        retval = 1
        for x in range(1, n+1, 1):
            retval *= x
        return retval
```

Then, I added the following code to `main.py`

```
from math_funcs.math_funcs import Math

a = Math()

add_test = a.add(5, 6)
print(add_test)
```

```
sub_test = a.subtract(7, 2)
print(sub_test)

print(a.multiply(5, 5))

print(a.divide(5,2))

print(a.absVal(-10))

print(a.exponent(2,7))

print(a.factorial(5))
```

4. Usage

4.1 Pyarmor

To obfuscate the entire directory I ran the following: `pyarmor obfuscate --src="." -r main.py`

Pros

- Source code is obfuscated which provides some intermediate layer of hardening.
- Running tools such as `strings` or `hexdump` against any of the files does not decode the source or provide anything meaningful.
- All other immediate methods of attempting to reverse engineer the file did not seem to work.

Cons

- Debugging seems to be nullified for files obfuscated by pyarmor.
- There are tools that when combined, can be used to deobfuscate the files back to their source. One example is using unpyarmor to take the encrypted bytes and compile it to a `.pyc` file, then use uncompyle6 to reverse engineer the `.pyc` file.

4.2 Nuitka

Using Nuitka to compile my example project into an executable: `python3 -m nuitka --follow-imports --unstripped main.py`

Pros

- Reverse engineering with a tool like Ghidra would require a higher degree of skill. Pulling apart the executable to reveal the source code would not be as obvious as something like a C/C++ binary would reveal.
- Portable and executable.

Cons

- Debugging is difficult using standard tools like gdb/pdb.
- Some very basic info about the source such as function names can be determined using `ltrace` and `strings`.

4.3 Pyarmor with Nuitka

To obfuscate with pyarmor when using Nuitka, the documentation suggests the following execution: `pyarmor obfuscate --src="." -r --restrict 0 --no-cross-protection main.py`

Obfuscated files are output to a newly created directory called `dist/`

To pack the obfuscated files using Nuitka first navigate to the `dist/` directory `cd dist/`

Finally, run nuitka using the following command. `python3 -m nuitka --follow-imports --unstripped --include-package pytransform main.py`

The `--follow-imports` flag will recursively trace imports from `main.py` and pack them into the executable for you. `--include-package pytransform` will build a package for `pytransform` with a shared library and pack it into the executable for you as well. Pytransform is essential for pyarmor to work. By default, I believe this is a python file `pytransform.py` that is placed inside the directory every module your project uses. Obviously that is a waste of space for large projects. This flag will build a package with `pytransform.so` inside of it and link it for you, saving space and speed with dynamic linking. The `--unstripped` flag keeps debug symbols in place. By default, the executable has its debug symbols stripped.

Pros

- Combines the advantages of a packaged executable built by nuitka with the added level of security provided through obfuscation with pyarmor.
- Added difficult to reverse engineer using common tools such as IDA or Ghidra.

Cons

- Still carries the disadvantage of not being debuggable.

4.5 Cython

To "cythonize" your python project, you can either convert all of you `.py` files to `.pyx` files by changing the extension or you can build them as is, depending on what your goal with Cython is. For my purposes, I don't really care about the speed advantages of Cython, but am instead focused more on building them into C executables. To do this from the command line, it is best to have a build script. I wrote mine in a file called `compile.py` with the following code:

```
from setuptools import setup, find_packages, Extension
from Cython.Distutils import build_ext
from Cython.Build import cythonize

ext_modules = [
    Extension("math_funcs.math_funcs",  ["math_funcs/math_funcs.py"]), # output
dir of resulting .so file
```

```
    Extension("main",  ["main.py"]),
    # ...all your modules that need be compiled...
]

setup(
    name = 'math_funcs',
    packages=find_packages(),
    cmdclass = {'build_ext': build_ext},
    ext_modules = cythonize(
        ext_modules,
        gdb_debug=True
    ),
)
```

Then run the program with the following flags: `python3 compile.py build_ext --inplace`

Pros

- Cython compiles the python files given to it into a `.so` shared library which can be called the exact same as a normal python module/package.
- Provides a decent layer of protection to source code.

Cons

- Difficult to debug. While the documentation does provide a method to debug using an included tool called `cygdb`, it is difficult to setup even by the developer's own admission.
- Requires some type of entry script that is not compiled with Cython. Whether that be the main that already exists or another python script that simply calls main, there must be some script that is the entry into the project.

4.6 Pyarmor With Cython

Pros

- Similar to Pyarmor + Nuitka minus the fact that Cython does not compile into an executable. You get the advantage of your python code compiled to C in a shared library with the added bonus of knowing your source code is obfuscated.

Cons

- Again, debugging is an issue. Even with debug symbols unstripped, there is no easy way to debug with common debuggers.
- Using pdb to debug the example project and starting in `main.py`, pdb would not enter the `.so` shared object that contains the functions in `math_funcs/math_funcs.py` being called. It seems that `pdb` is not capable of debugging anything besides standard python code.

5. Galaxy Brain Approach

Modify the Python source code (which is written in C) and compile a special build for your purpose.

The way Python works is fully documented and open-source. For instance, consider the `.pyc` file format. Much of the code which deals with reading/writing pyc's can be found in `marshal.c`. If you modify the code and reorder the sequence in which the objects are read/written you can easily throw off decompilers. To further strengthen the protection, you can encrypt the bytecode which will be decrypted only at run-time.

Another technique commonly used is opcode remapping. The set of bytecode instructions in Python can be found in `opcode.h`.

Consider these two instructions:

```
#define DUP_TOP_TWO               5

#define BINARY_TRUE_DIVIDE       27
```

You can interchange the opcodes for these instructions. As a result the bytecode will only be understood by your custom Python build.

Pros

- It would be nearly impossible to reverse engineer your program without the special build of Python you created.

Cons

- Workload -> payoff likely not worth it. It would be a lot of work to maintain such a build and you would have to document every detail in such a high degree for future developers who may need to modify your code.
- Any upgrades to newer Python versions would require the same modifications in order to maintain backwards compatibility with older modified versions.