Python Malware On The Rise

cyborgsecurity.com/cyborg_labs/python-malware-on-the-rise/

July 14, 2020

The vast majority of serious malware over the past 30 years has been written in Assembly or compiled languages such as C, C++, and Delphi. However, ever-increasing over the past decade, a large amount of malware has been written in interpreted languages, such as Python. The low barrier to entry, ease of use, rapid development process, and massive library collection has made Python attractive for millions of developers- including malware authors. Python has quickly become a standard language in which threat actors create Remote Access Trojans (RATs), information stealers, and vulnerability exploit tools.
As Python continues to grow radically in popularity and the C malware monoculture continues to be challenged, it would seem only certain that Python will be increasingly utilized as malware in cyber attacks.

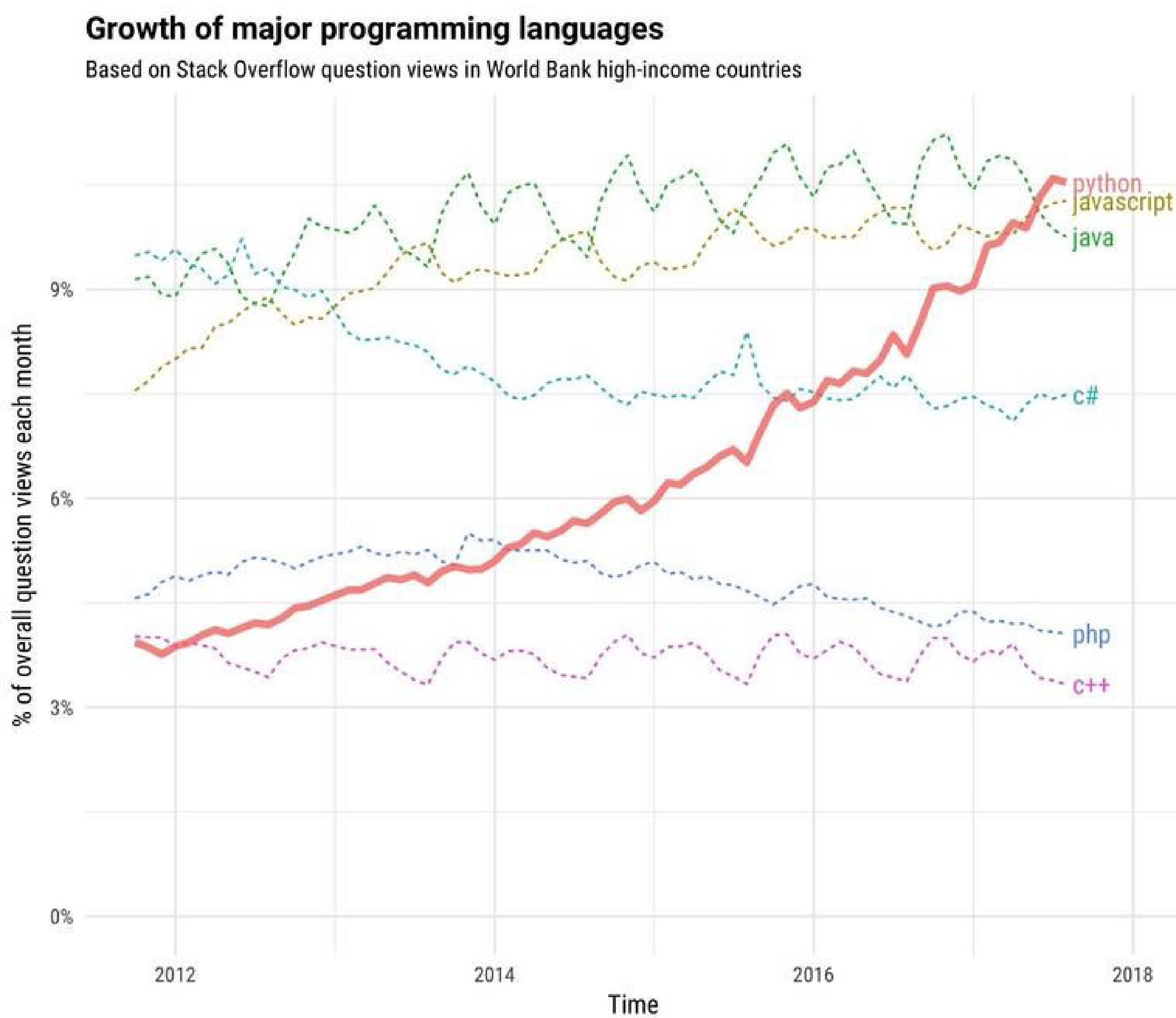


Image Source: Stack Overflow
⟶ Click here to download our free white paper with solutions to the industry's growing content problem.

THE TIMES THEY ARE A-CHANGIN'

In comparison to a standard compiled language like C, writing malware in Python comes with a whole host of difficulties. The first being that Python is required to be installed on the operating system in order to interpret and execute Python code. However, as we'll see in the next section, a Python program can easily be converted into a native executable using a variety of different methods.

Malware written in Python will also have adverse effects on file size, memory footprint, and processing power. Serious malware is often designed to be small, stealthy, have low memory footprint, and use limited processing power. A compiled malware sample written in C might be 200 KB, while a comparable malware sample written in Python might be 20 MB after converted into an executable. Both the CPU & RAM usage will also be significantly higher when using an interpreted language.

However, it's 2020 and the digital landscape isn't what it once was. The internet is faster than it's ever been, our computers have more memory & storage capacity than ever, and CPUs get faster every year. Python is also more ubiquitous than ever, coming pre-installed on macOS and most all Linux distributions by default.

NO INTERPRETER? NO PROBLEM!

Microsoft Windows is still the primary target for most malicious campaigns, and it does not come with Python installed by default. Therefore, for threat actors to distribute their malware effectively they must convert their Python code into an executable format. There are many methods to "compile Python" into a native executable. Let's take a look at the few most popular methods…

PyInstaller

PyInstaller is capable of building Python applications into stand-alone executables for Windows, Linux, macOS and more by "freezing" Python code. It is one of the most popular methods to convert Python code into executable format and has been used widely for both legitimate and malicious purposes.

Let's create a simple "Hello, world!" program in Python and freeze it into a stand-alone executable using PyInstaller:

```
$ cat hello.py
print('Hello, world!')

$ pyinstaller --onefile hello.py
...

$ ./dist/hello
Hello, world!

$ file dist/hello
dist/hello: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter
/lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32,
BuildID[sha1]=294d1f19a085a730da19a6c55788ec0
8c2187039, stripped

$ du -sh dist/hello
7.0M    dist/hello
```

This process created a portable, stand-alone Linux ELF (Executable and Linkable Format) which is the equivalent to an EXE on Windows. Now let's create and compile a "Hello, world!" program in C on Linux for comparison:

```
$ cat hello.c
#include
int main() {
    printf("Hello, world!");
}

$ gcc hello.c -o hello

$ ./hello
Hello, world!

$ file hello
hello: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter
/lib64/ld-linux-x86-64.so.2, BuildID[sha1]=480c7c75e09c169ab25d1b81bd28f66fde08da7c,
for GNU/Li
nux 3.2.0, not stripped

$ du -sh hello
20K     hello
```

Notice how much larger the file size is: 7 MB (Python) vs 20 KB (C)! This demonstrates the major drawback we discussed previously about file size and memory usage. The Python executable is so much larger due to the fact it must bundle the Python interpreter (as a shared object file on Linux) inside the executable itself in order to run.

py2exe

Py2exe is another popular method to convert Python code into Windows EXE (executable) format that can be run natively. Similar to PyInstaller, it bundles the Python interpreter with your Python code to make a portable executable. Py2exe is likely to fall out of style with time as it has not been supported past Python 3.4, this is due to the bytecode in CPython being heavily changed in Python 3.6 and beyond.

Py2exe utilizes distutils and requires a small `setup.py` script to be created to produce an executable. Let's create an example "Hello, world!" executable using py2exe:

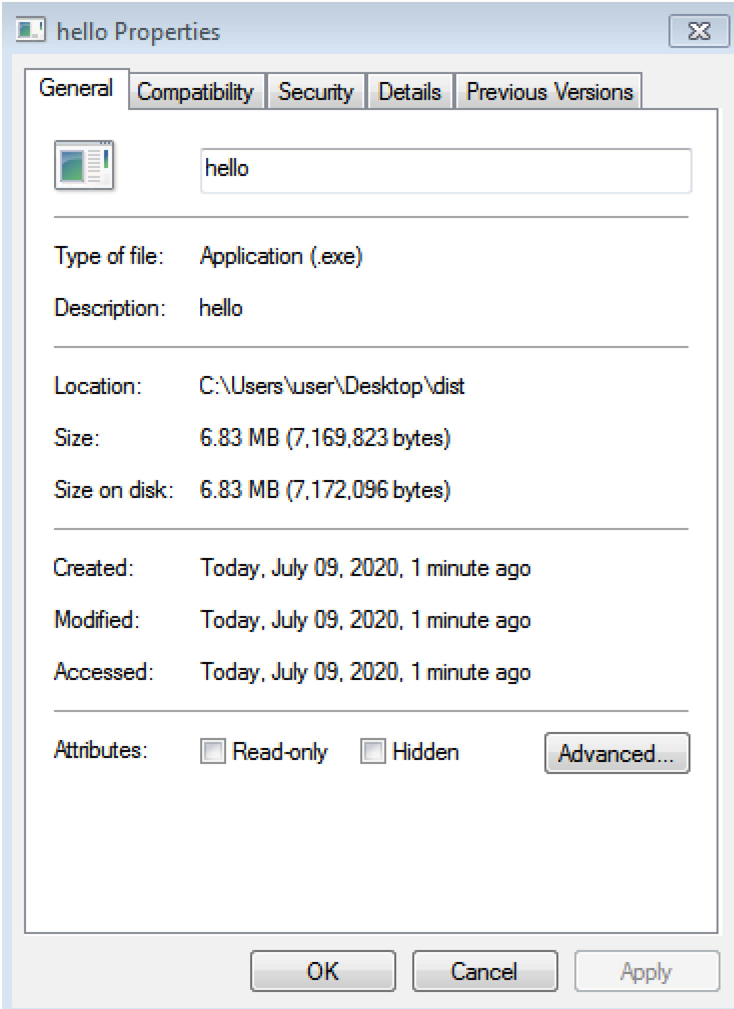
```
> type hello.py
print('Hello, world!')

> type setup.py
import py2exe
from distutils.core import setup
setup(
    console=['hello.py'],
    options={'py2exe': {'bundle_files': 1, 'compressed': True}},
    zipfile=None
)

> python setup.py py2exe
...

> dist\hello.exe
Hello, world!
```

The `hello.exe` created by py2exe is similar in size to PyInstaller coming in at 6.83 MB.



Nuitka

Nuitka is perhaps the most underutilized, and yet more advanced method of compiling 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.

Let’s create a “Hello, world!” Python program on Linux and compile it using Nuitka:

```
$ cat hello.py
print('Hello, world!')

$ nuitka3 hello.py
...

$ ./hello.bin
Hello, world!

$ file hello.bin
hello.bin: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically
linked, interpreter
/lib64/ld-linux-x86-64.so.2, BuildID[sha1]=eb6a504e8922f8983b23ce6e82c45a907c6ebadf,
for GNU/Linux
3.2.0, stripped

$ du -sh hello.bin
432K    hello.bin
```

Nuitka produced a portable binary very simply, and at 432 KB is a fraction of the size of what PyInstaller or py2exe can produce! How is Nuitka able to do this? Let’s take a look at the build folder:

```
$ cloc hello.build/
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C                               11           2263            709           8109
C/C++ Header                     1              1              0              7
-------------------------------------------------------------------------------
SUM:                            12           2264            709           8116
-------------------------------------------------------------------------------
```

Nuitka produced over 8,000 lines of C code from our 1 line Python program. The way Nuitka works is it actually translates the Python modules into C code and then uses libpython and static C files of its own to execute in the same way as CPython does.

This is very impressive, and it seems highly likely the Nuitka “Python compiler” will see further adoption as time goes on. As we’ll see later, Nuitka might have a further, built-in advantage in protection against Reverse Engineering (RE). There already exist several tools to easily analyze binaries produced by PyInstaller and py2exe to recover Python source code. However, by Nuitka translating the Python code to C it is much more difficult to reverse engineer.

YO DAWG, I HEARD YOU LIKE TOOLS…

Python malware can take advantage of a massive ecosystem of open-source Python packages and repositories. Almost anything you could think of, someone has already built it using Python. This is a huge advantage to malware authors as simplistic capabilities can be cherry-picked from the open web and more complex capabilities likely don't need to be written from scratch.

Let's take a look at three simple, yet powerful tool examples:

1. Code Obfuscation
2. Taking Screenshots
3. Performing Web Requests

Tool Example 1 — Obfuscation

Malware authors using Python have many libraries they could use to obfuscate their Python code to make code readability much more difficult, such as: pyminifier and pyarmor.

Here's a small example of how `pyarmor` can obfuscate Python code:

```
$ cat hello.py
print('Hello, world!')

$ pyarmor obfuscate hello.py
...

$ cat dist/hello.py
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\x00\x0

\x18\xf4\x63\x79\xf6\xaa\xd7\xbd\xc8\x85\x25\x4e\x4f\xa6\x80\x72\x9f\x00\x00\x00\x00\x

0\x00\xec\x50\x8c\x64\x26\x42\xd6\x01\x10\x54\xca\x9c\xb6\x30\x82\x05\xb8\x63\x3f\xb0\

97\x0b\xc1\x49\xc9\x47\x86\x55\x61\x93\x75\xa2\xc2\x8c\xb7\x13\x87\xff\x31\x46\xa5\x29

xdf\x32\xed\x7a\xb9\xa0\xe1\x9a\x50\x4a\x65\x25\xdb\xbe\x1b\xb6\xcd\xd4\xe7\xc2\x97\x3

\xd3\xd0\x74\xb8\xd5\xab\x48\xd3\x05\x29\x5e\x31\xcf\x3f\xd3\x51\x78\x13\xbc\xb3\x3e\x

a\x05\xfb\xac\xed\xfa\xc1\xe3\xb8\xa2\xaa\xfb\xaa\xbb\xb5\x92\x19\x73\xf0\x78\xe4\x9f\

7a\x1c\x0c\x6a\xa7\x8b\x19\x38\x37\x7f\x16\xe8\x61\x41\x68\xef\x6a\x96\x3f\x68\x2b\xb7

x39\x51\xa3\xfc\xbd\x65\xdb\xb8\xff\x39\xfe\xc0\x3d\x16\x51\x7f\xc9\x7f\x8b\xbd\x88\x8

\xe1\x23\x61\xd0\xf1\xd3\xf8\xfa\xce\x86\x92\x6d\x4d\xd7\x69\x50\x8b\xf1\x09\x31\xcc\x

f\x37\x12\xd4\xbd\x3d\x0d\x6e\xbb\x28\x3e\xac\xbb\xc4\xdb\x98\xb5\x85\xa6\x19\x11\x74\

df', 1)

$ python dist/hello.py
Hello, world!
```

Tool Example 2 — Screenshots

Information stealing malware will often come with the capability to take screenshots of the users desktop in order to steal sensitive information. Using Python this is all too easy and there are several libraries to accomplish this, including: pyscreenshot and python-mss.

A screenshot can easily be taken with `python-mss` like this:

```
from mss import mss

with mss() as sct:
    sct.shot()
```

Tool Example 3 — Web Requests

Malware will often conduct web requests to do a variety of different things on a compromised endpoint, including: web-based command & control (C2), obtaining the external IP address, downloading a second stage payload, and more. Using Python, making web requests is very simple and can be done using the standard library or with open-source libraries such as: requests and httpx.

The external IP address of a compromised endpoint can easily be obtained using `requests` like so:

```
import requests

external_ip = requests.get('http://whatismyip.akamai.com/').text
```

THE STRENGTH OF `EVAL()`

Typically, Python's `eval()` built-in function is seen as very dangerous as it presents serious security risks when used in production code. However, `eval()` has a huge strength when used within Python malware.

The `eval()` function is very powerful and can be used to execute strings of Python code from within the Python program itself. This single function is often seen as an advanced capability in compiled malware. It is the ability to run high-level scripts or "plugins" on-the-fly when utilized correctly. This is similar to when C malware includes a Lua scripting engine to give the malware the ability to execute Lua scripts. This has been seen in high-profile malware such as Flame.

Let's imagine a hypothetical APT group is interacting remotely with some Python-based malware. If this group came into an unexpected situation where they needed to react quickly, being able to directly execute Python code on the end target would be highly beneficial. In addition, the Python malware could be placed on a target effectively "featureless" and capabilities could be executed on the target on an as-needed basis to remain stealthy.

INTO THE WILD

Alright, let's take a look at a few real world Python malware samples!

Image Source: Lord of the Rings — Fellowship of the Ring

SeaDuke

The SeaDuke malware is likely the most high-profile compromise that Python-based malware has been involved in. During 2015 and 2016, the Democratic National Committee (DNC) was compromised by two threat actor groups that have been attributed by many analysts to APT 28 & 29.

Some fantastic analysis of SeaDuke was conducted by Palo Alto's Unit 42. The decompiled Python source code Unit 42 uncovered can be found here. In addition, F-Secure published a great whitepaper on Duke malware that covers SeaDuke and associated malware.

The SeaDuke malware is a Python trojan that was made into a Windows executable using PyInstaller and packed with UPX. The Python source code was obfuscated to make the code more difficult for analysts to read. The malware had many capabilities including several methods to establish persistence on Windows, ability to run cross-platform, and perform web requests for command & control.

```
    def decode_data(self,pSsWAYdKJqgPHbRoVCwjkvMcmtuxInGEhaFfLBXUOrNlTzeDQy):
        pSsWAYdKJqgPHbRoVCwjkvMcmtuxInGEhaFfLBXUOrNlTQizey=HttpParserKlass()
        pSsWAYdKJqgPHbRoVCwjkvMcmtuxInGEhaFfLBXUOrNlTQizey.feed(pSsWAYdKJqgPHbRoVCwjkvMcmtuxInGEhaFfLBXUOrN
        pSsWAYdKJqgPHbRoVCwjkvMcmtuxInGEhaFfLBXUOrNlTzeDQy=pSsWAYdKJqgPHbRoVCwjkvMcmtuxInGEhaFfLBXUOrNlTQiz
        pSsWAYdKJqgPHbRoVCwjkvMcmtuxInGEhaFfLBXUOrNlTQizeD=pSsWAYdKJqgPHbRoVCwjkvMcmtuxInGEhaFfLBXUOrNlTQiz
        pSsWAYdKJqgPHbRoVCwjkvMcmtuxInGEhaFfLBXUOrNlTzeDQy=''.join(botKlass.decode_data_pattern.findall(pSs
        pSsWAYdKJqgPHbRoVCwjkvMcmtuxInGEhaFfLBXUOrNlTzeDQy=base64_b64decode(pSsWAYdKJqgPHbRoVCwjkvMcmtuxInG
        try:
            pSsWAYdKJqgPHbRoVCwjkvMcmtuxInGEhaFfLBXUOrNlTziQDe=CryptoKlass.tr_decrypt(pSsWAYdKJqgPHbRoV
            pSsWAYdKJqgPHbRoVCwjkvMcmtuxInGEhaFfLBXUOrNlTQizDy=json_loads(pSsWAYdKJqgPHbRoVCwjkvMcmtuxI
        except:
            for pSsWAYdKJqgPHbRoVCwjkvMcmtuxInGEhaFfLBXUOrNlTQizDe in pSsWAYdKJqgPHbRoVCwjkvMcmtuxInGEh
                try:
                    self.__send_request(url=pSsWAYdKJqgPHbRoVCwjkvMcmtuxInGEhaFfLBXUOrNlTQizDe)
                except Exception as exp:
                    pass
```

PWOBot

PWOBot is Python-based malware, similar to SeaDuke it is compiled using PyInstaller into a Windows executable. It was prevalent during 2013–2015 and affected several European organizations, mostly in Poland.

The malware was very full featured and included the ability to log key strokes, establish persistence on Windows, download & execute files, execute Python code, create web requests, and mine cryptocurrency. Some great analysis of PWOBot was conducted by Palo Alto's Unit 42.

PyLocky

PyLocky is a Python-based ransomware, compiled with PyInstaller into a Windows standalone executable. It targeted several different countries including the USA, France, Italy, and Korea. It included anti-sandbox capabilities, command & control, and encrypted files using 3DES (Triple DES) cipher.

Some great analysis of PyLocky was conducted by Trend Micro. Talos Intelligence analysts reversed engineered PyLocky and were able to create a file decryptor for victims to restore their encrypted files.

PoetRAT

PoetRAT is a Python-based trojan that targeted the Azerbaijan government and energy sector in early 2020. The trojan enumerated systems and stole information related to ICS/SCADA systems that control wind turbines.

The malware was dropped using malicious Microsoft Word documents. The RAT presented many capabilities for stealing information including file extraction over FTP, taking images with webcams, uploading additional tools, keylogging, browser enumeration, and credential theft. Talos Intelligence reported on this threat actor and produced a fantastic writeup on the unknown actor that used this malware.

This short script was used by the threat actor to capture web cam images:

```
import cv2

camera = cv2.VideoCapture(0)
for i in range(10):
    return_value, image = camera.read()
    cv2.imwrite('opencv'+str(i)+'.png', image)
del(camera)
```

Image Source: Talos Intelligence

Open Source

In addition to the malware found in the wild, several Python RATs are available open-source such as pupy and Stitch. These open-source Python trojans show just how complex and feature rich Python malware can be. The pupy RAT is cross-platform, features an all-in-memory execution guideline, leaves a very low footprint, can combine several C2 encryption methods, migrate into processes using reflective injection, and can load remote python code from memory.

PYTHON MALWARE ANALYSIS TOOLS

There are many tools available to analyze Python malware, even in compiled form. Let’s take a cursory look at what tools malware analysts can use to tear into Python malware.

uncompyle6

The successor to decompyle, uncompyle, and uncompyle2- uncompyle6 is a native Python cross-version decompiler and fragment decompiler. It can be used to translate Python bytecode back into Python source code.

For example, taking our “Hello, world!” script from earlier and executing it as a module I’m presented with a pyc file (byte code). We can recover the source code of our script by using uncompyle.

```
$ xxd hello.cpython-38.pyc
00000000: 550d 0d0a 0000 0000 16f3 075f 1700 0000  U.........._....
00000010: e300 0000 0000 0000 0000 0000 0000 0000  ................
00000020: 0002 0000 0040 0000 0073 0c00 0000 6500  [email protected]
00000030: 6400 8301 0100 6401 5300 2902 7a0d 4865  d.....d.S.).z.He
00000040: 6c6c 6f2c 2077 6f72 6c64 214e 2901 da05  llo, world!N)...
00000050: 7072 696e 74a9 0072 0200 0000 7202 0000  print..r....r...
00000060: 00fa 2d2f 686f 6d65 2f75 7365 722f 746d  ..-/home/user/tm
00000070: 702f 7079 7468 6f6e 5f61 7274 6963 6c65  p/python_article
00000080: 2f6e 2f74 6573 742f 6865 6c6c 6f2e 7079  /n/test/hello.py
00000090: da08 3c6d 6f64 756c 653e 0100 0000 f300  ........
000000a0: 0000 00

$ uncompyle6 hello.cpython-38.pyc | grep -v '#'
print('Hello, world!')
```

pyinstxtractor.py (PyInstaller Extractor)

The PyInstaller Extractor can extract Python data from PyInstaller compiled executables. It’s very simple to run:

```
> python pyinstxtractor.py hello.exe
...
```

This will produce pyc files you can then use with the uncompyle6 decompiler to recover source code.

python-exe-unpacker

The pythonexeunpack.py script can be used to unpack and decompile executables that are built with py2exe. It can be used like so:

```
> python python_exe_unpack.py -i hello.exe
...
```

DETECTING PYTHON COMPILED EXECUTABLES

Both PyInstaller and py2exe when compiled on Windows place unique strings within their binary executable. Which means they can be detected with simple YARA rules.

PyInstaller writes the string “pyi-windows-manifest-filename” near the end of the executable, you can see it here in a hex editor (HxD):

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text
005CADE0  2E 64 6C 6C 00 00 00 00 00 00 00 00 00 00 00 00  .dll............
005CADF0  00 00 00 00 30 00 1E 4D 35 00 00 01 E1 00 00 04  ....0..M5...á...
005CAE00  06 01 62 68 65 6C 6C 6F 2E 65 78 65 2E 6D 61 6E  ..bhello.exe.man
005CAE10  69 66 65 73 74 00 00 00 00 00 00 00 00 00 00 00  ifest...........
005CAE20  00 00 00 00 20 00 1E 4F 16 00 01 6C EC 00 03 04  .... ..O...lì...
005CAE30  98 01 62 70 79 65 78 70 61 74 2E 70 79 64 00 00  ˜.bpyexpat.pyd..
005CAE40  00 00 00 00 20 00 1F BC 02 00 17 9C 9D 00 37 18  .... ..¼...œ..7.
005CAE50  98 01 62 70 79 74 68 6F 6E 33 36 2E 64 6C 6C 00  ˜.bpython36.dll.
005CAE60  00 00 00 00 20 00 37 58 9F 00 00 37 B6 00 00 6A  .... .7XŸ..7¶..j
005CAE70  98 01 62 73 65 6C 65 63 74 2E 70 79 64 00 00 00  ˜.bselect.pyd...
005CAE80  00 00 00 00 20 00 37 90 55 00 06 E2 D4 00 0F 05  .... .7.U..âÔ...
005CAE90  80 01 62 75 63 72 74 62 61 73 65 2E 64 6C 6C 00  €.bucrtbase.dll.
005CAEA0  00 00 00 00 30 00 3E 73 29 00 05 87 4D 00 0D D4  ....0.>s)..‡M..Ô
005CAEB0  98 01 62 75 6E 69 63 6F 64 65 64 61 74 61 2E 70  ˜.bunicodedata.p
005CAEC0  79 64 00 00 00 00 00 00 00 00 00 00 00 00 00 00  yd..............
005CAED0  00 00 00 00 50 00 43 FA 76 00 00 00 00 00 00 00  ....P.Cúv.......
005CAEE0  00 00 6F 70 79 69 2D 77 69 6E 64 6F 77 73 2D 6D  ..opyi-windows-m
005CAEF0  61 6E 69 66 65 73 74 2D 66 69 6C 65 6E 61 6D 65  anifest-filename
005CAF00  20 68 65 6C 6C 6F 2E 65 78 65 2E 6D 61 6E 69 66   hello.exe.manif
005CAF10  65 73 74 00 00 00 00 00 00 00 00 00 00 00 00 00  est.............
005CAF20  00 00 00 00 30 00 43 FA 76 00 03 25 A6 00 0B D5  ....0.Cúv..%¦..Õ
005CAF30  6A 01 78 62 61 73 65 5F 6C 69 62 72 61 72 79 2E  j.xbase_library.
005CAF40  7A 69 70 00 00 00 00 00 00 00 00 00 00 00 00 00  zip.............
005CAF50  00 00 00 00 20 00 47 20 1C 00 11 52 85 00 11 52  .... .G ...R…..R
005CAF60  85 00 7A 50 59 5A 2D 30 30 2E 70 79 7A 00 00 00  ….zPYZ-00.pyz...
005CAF70  00 4D 45 49 0C 0B 0A 0B 0E 00 58 7F C9 00 58 72  .MEI......X.É.Xr
005CAF80  A1 00 00 0C D0 00 00 00 24 70 79 74 68 6F 6E 33  ¡...Ð...$python3
005CAF90  36 2E 64 6C 6C 00 00 00 00 00 00 00 00 00 00 00  6.dll...........
005CAFA0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
005CAFB0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
005CAFC0  00 00 00 00 00 00 00 00 00                       .........
```

Here’s a YARA rule for detecting PyInstaller compiled executables (Source):

```
import "pe"

rule PE_File_pyinstaller
{
    meta:
        author = "Didier Stevens (https://DidierStevens.com)"
        description = "Detect PE file produced by pyinstaller"
    strings:
        $a = "pyi-windows-manifest-filename"
    condition:
        pe.number_of_resources > 0 and $a
}
```

Here’s a second YARA rule for detecting py2exe compiled executables (Source):

```
import "pe"

rule py2exe
{
  meta:
        author = "Didier Stevens (https://www.nviso.be)"
        description = "Detect PE file produced by py2exe"
  condition:
        for any i in (0 .. pe.number_of_resources - 1):
          (pe.resources[i].type_string ==
"P\x00Y\x00T\x00H\x00O\x00N\x00S\x00C\x00R\x00I\x00P\x00T\x00")
}
```

CONCLUSION

That’s all for now from the world of Python malware. It’s very interesting watching malware trends change as computer systems become faster and easier to operate. As a security industry we need to keep an eye on Python-based malware, or it might just sink its fangs into us when we’re least expecting.