Challenge #3 Solution

by Bob Jung

I guess I should probably start by explaining what the deal is with the goat.

Figure 1: Pop-Up Window for Solution #3

This is a picture of Elfie, our Nigerian Dwarf, who is basically the happiest creature in existence. My wife, Carrie Jung (also a malware reverse engineer on the FLARE team), named her after Elphaba, the main character from the Broadway play Wicked.

Anyways, this is a great view of our backyard mini-farm in New Mexico. The barn with all the windows, next to Elfie's ear, is actually our home office where I'm writing this right now. A fun fact is that the barn was actually the unofficial original location of Mandiant's Albuquerque office. There were 6 of us working in there at one point till things got crowded and we had to get a "real" office. It hasn't been the same without roosters interrupting telecons.

Overview

Alright, enough about farm animals. If you haven't run into "frozen" Python before, this executable would probably seem pretty challenging at first. There are a lot of tools out there

like PyInstaller, py2exe, and bbfreeze that will combine Python scripts into stand alone executables packaged with all of the dependencies they need to run.

https://github.com/pyinstaller/pyinstaller/wiki

http://www.py2exe.org/

https://pypi.python.org/pypi/bbfreeze

These packagers tend to create relatively large executables as they bundle up the Python interpreter, application code, python dependencies, and required native libraries into a single monolithic executable. For example, using PyInstaller to package a "Hello World" script will produce an executable approximately 8 MB in size.

Before you get too angry with me, you should know we encounter frozen Python in several of the malware families that we're tracking. In fact, the obfuscation of the script itself was partly inspired by the shenanigans I've had to untangle. I haven't seen any embedded goat pictures yet, though.

Pulling out the Python

If the icon wasn't a tip off, it should be apparent after some basic string analysis of the sample that we're dealing with Python.

Another clue that we're dealing with a packaged python script is the python library dependencies that are written to the file system:

```
%Temp%\_MEI19202\Microsoft.VC90.CRT.manifest
%Temp%\_MEI19202\msvcr90.dll
%Temp%\_MEI19202\msvcp90.dll
%Temp%\_MEI19202\msvcm90.dll
%Temp%\_MEI19202\python27.dll
%Temp%\_MEI19202\select.pyd
%Temp%\_MEI19202\unicodedata.pyd
%Temp%\_MEI19202\PySide.QtCore.pyd
%Temp%\_MEI19202\_hashlib.pyd
%Temp%\_MEI19202\bz2.pyd
%Temp%\_MEI19202\_ssl.pyd
%Temp%\_MEI19202\_socket.pyd
%Temp%\_MEI19202\pyside-python2.7.dll
%Temp%\_MEI19202\shiboken-python2.7.dll
%Temp%\_MEI19202\QtCore4.dll
%Temp%\_MEI19202\QtGui4.dll
%Temp%\_MEI19202\elfie.exe.manifest
```

Beyond that, performing a web search for some of the strings (For example: "Could not allocate buffer for TOC.") quickly leads you to the source for PyInstaller. There are several tools

out there that will attempt to extract the original files used to build Python packaged executables. In particular, Pyinstaller Extractor by extremecoders seems to work nicely:

http://sourceforge.net/projects/pyinstallerextractor/files/?source=navbar

Once you run the Pyinstaller Extractor tool on the executable, you'll notice a whole bunch of cruft that gets extracted. After a cursory survey of the files that are dropped, the "elfie" file seems pretty conspicuous with a ton of crazy looking Python string variables.

Deobfuscating the script

It should be noted that it's not always possible to get the actual Python source extracted from the executable. Other Python packagers like py2exe only include the compiled byte code from the scripts .pyc files. So if you ever encounter that problem you'll have the additional problem of attempting to decompile the .pyc files. In this case however, we lucked out and were able to grab the original source.

Next, there is the problem of understanding the ~ 56k lines of obfuscated source that appear to be adding random characters to strings that have variable names with annoying mix of zero's and capital o's.

```
O0OO0OO00O000OOOOO0OOOOO0O00O00OOO = 'IRGppV0FJM3BRRlNwWGhNNG'
OO0O0O00OO0O0OOOOOOOOO0O0OO00OOOO0 = 'UczRkNZZ0JVRHJjbnRJUWlJV3FRTkpo'
OOO000O0O00O0OOOOO0O0000O0OO0000 = 'xTStNRDJqZG9nRCtSU1V'
OOO0000O0OO0OOOOO000O0000O0OO0OO0 += 'Rbk51WXI4dmRaOXlwV3NvME0ySGp'
OOO0OOOOOOOO0000O000O00OO00OOO00 = 'ZnJvbSBQeVNpZGUgaW1wb3J'
O000OO0000000O0000OOOOO0OOO000OO0 = 'z'
O00O0O000OO0O0OO0O00OO0O0O0000OO0 = 'OFJVT1d0TTJZTUppSWhVNWFQbDI3MkJVMlNLVWMvYVh'
O0OO0O000O0OOO0O0OO0O00OO0000OO0 = 'aZzV1bFJPQnpIS1F5SWZuSDBMWUw0NGRGQ1lVeVF'
O000OOOOOO0OO0O0OO0O0O00OO00OO00 = 'RC9XQXdIQ3FmcGxoK1owYlB4M2xQeGxoT2lvMllRaTI5dnJ5WDdx'
OO0OOOO00O0OOOOO0OOOOO0O000OOO00 = 'hoOHBRUTFRWWttQklySmJDVWxaeTU4RWRQYXdFQktMUjl5VzJ6U'
O0O0OO000OOO0000OOO0OOOO0OO0000 = 'tnbVJJbHBaZVh3SmlKZ1FYTER'
OO0O0OOOO0O00O0OO0O00O0O00OOOOO0 = 'ejc2YjFq'
OOOO00O0OOOOO00O00OO00OOO0O00OOO0 = '9tUHJ1b0xMUFJObVpNWWxiWW91MGNvZWNwRjZhUFQ3RWVrZXo5MGRCdTFSb'
OOO0O000O0OOOOO0O00O00OO0O00OOO0 = '2cFVOa0lwMjlSSFV4TDFSSG4ySTBz'
OOOOO0OO0O00OOOOO0O0000O0O00OOOO0 = 'WUExiQ2IyUXdDY1VmQ3luQU42cXVQQlplZ0N'
O0O0OOO000O00O00O0O0OOOO000O0000 = 'hTUhJWnNISVpNSExlMG5LWnM5VmVFaS94cWppbUIvVVJpL014aThJSnhR'
OOO00O0OOOO0O0O0OOOO000O0OOO000 = 'MUFUL0ZEdnhYVVM4SzlvRlIyZXpDdVo3VGFTenprRWEzVGlFUWxD'
OO0OOO00OO0OO0O0O00O0O00O0OO0000 = 'NHBBMUJBTVErRzF'
```

Figure 2: Obfuscated source fragment for Solution #3

After a quick scan you'll notice that this appears to be consistent till the end of the script where all the string variables appear to be added together, base64 decoded, then exec'ed:

```
import base64
exec(base64.b64decode(OOOOOOOOO0OOOO0OOO0OOOO0OOOOOOO0 + O0OOOO0OO0OOOOOO00OOO0OOOOOO0O0 +
O0OO0OOOOO0OOOOOO0OOOOOO00OOO0O0 + OOOOO0OO0OO0OOO0OOOOO0OOO0O0OOO0 +
OOOOO0OOOO0O0OOO0OOOOOO0OOOOOO0O + OOOOOOOO0OOO0OOOOOOOO0OOOOO0OO0O +
O0OOOOOOO0OOOOOOOOOOOO0OOO0O0OOO + OOO0OOOOOO0O0OOOOOOOO0OOOOOOOOOO +
OOOOOOOOOOOOOO0OOOOOOOOOO0OOOOO0 + OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO0 +
```

```
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO + OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO +
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO + OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO +
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO + OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO +
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO + OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO +
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO + OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO +
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO + OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO +
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO + OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO +
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO + OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO +
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO + OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO +
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO + OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO +
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO + OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO +
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO + OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO +
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO + OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO +
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO + OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO +
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO + OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO +
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO + OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO +
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO + OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO +
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO + OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO +
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO + OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO +
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO + OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO +
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO + OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO +
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO + OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO +
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO + OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO +
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO + OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO +
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO + OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO +
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO + OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO +
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO + OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO))
```

Figure 3: Concatenation and exec of obfuscated code in Solution #3

This is a common obfuscation pattern you'll see in a variety of scripting languages where the "outer" script will simply build up a string to execute. Getting the "inner" script is as easy as changing the "exec" to a "print" and executing the code in a Python interpreter to dump out the inner script that it would have executed. Since this script was relatively simple at only around 50 lines, at this point it should be relatively easy to spot the key hiding out in this block:

```
    def OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO(self):
        OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO = getattr(self, 'txeTnialPot'[::-1])()
        if (OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO == ''.join((OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO for
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO in reversed('moc.no-eralf@OOOOY.sev0000L.eiflE')))):
                self.OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO.setWindowTitle('!sseccus taerg'[::-1])
                self.OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO = True
                self.OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO.setVisible(False)
                self.OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO.setVisible(True)
```

Figure 4: Code Fragment showing reversed key in Solution #3

This inner script was actually programmatically obfuscated using a simple AST (Abstract Syntax Tree) obfuscator. I originally got interested in Python obfuscation after the first pure Python backdoor I had to analyze that weighed in at a few thousand lines of decently obfuscated Python. It was obvious that the malware authors had somehow programmatically modified the structure of their code as well as their variable names. After being on the receiving end, creating this flare-on challenge was a fun way to explore AST obfuscation. If you're interested in learning more, Jurien Bremer wrote a really excellent blog post and has some working code to get you started playing with ideas:

http://jbremer.org/Python-source-obfuscation-using-asts/

So that's pretty much it!

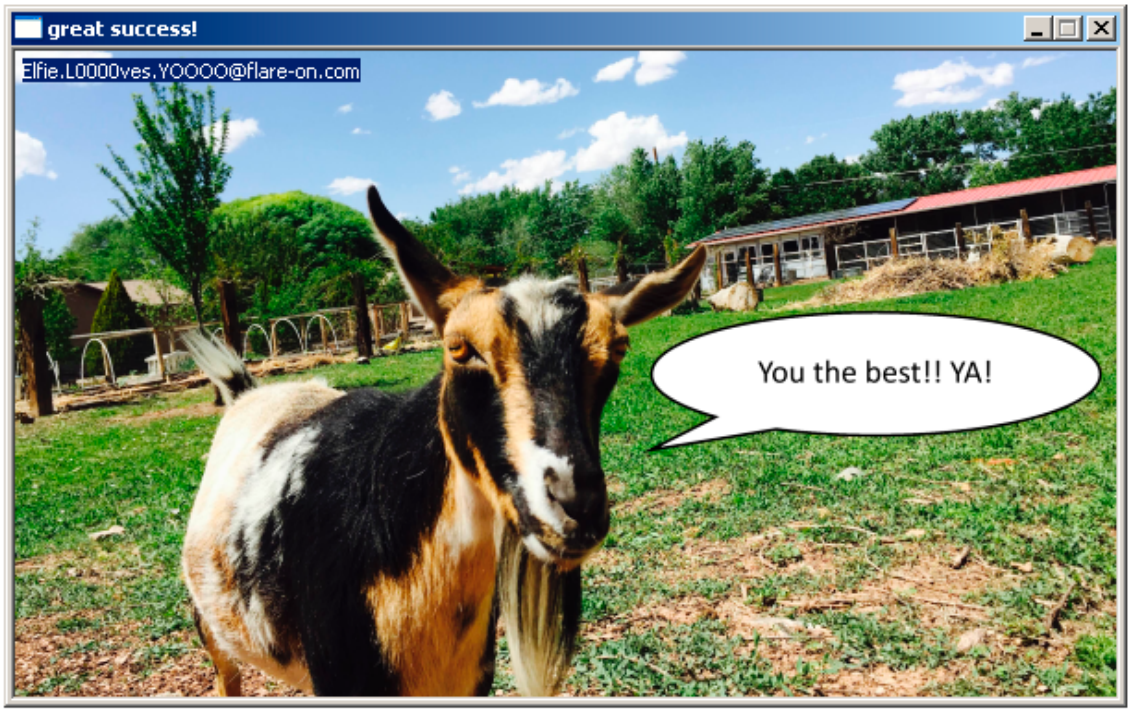
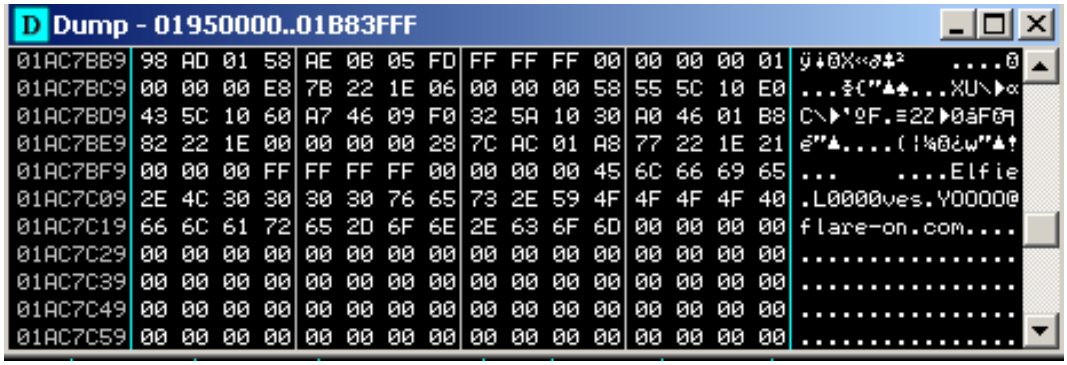


Figure 5: Success Screen from Solution #3

Encore! The cheater way to solve this

OK fine, as we always say "there is no cheating in reversing malware", but I should mention that some people hit upon an insanely easy way to solve this one in four seconds. Once you ran the program, you could have just searched all memory for the ASCII string "flare-on" to find the key. Basically, the Python interpreter leaves a copy of the string in memory after the call to `reversed('moc.no-eralf@OOOOY.sev0000L.eiflE')`, and you have a shot at finding it before the string gets garbage collected. You can try this from the memory map view in Olly, just right click on the first section and select "Search", then type "flare-on.com" in the ASCII field.


Dump - 01950000..01B83FFF
01AC7BB9 98 AD 01 58 AE 0B 05 FD FF FF FF 00 00 00 00 01
01AC7BC9 00 00 00 E8 7B 22 1E 06 00 00 00 58 55 5C 10 E0
01AC7BD9 43 5C 10 60 A7 46 09 F0 32 5A 10 30 A0 46 01 B8
01AC7BE9 82 22 1E 00 00 00 00 28 7C AC 01 A8 77 22 1E 21
01AC7BF9 00 00 00 FF FF FF FF 00 00 00 00 45 6C 66 69 65
01AC7C09 2E 4C 30 30 30 30 76 65 73 2E 59 4F 4F 4F 4F 40
01AC7C19 66 6C 61 72 65 2D 6F 6E 2E 63 6F 6D 00 00 00 00
01AC7C29 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01AC7C39 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01AC7C49 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01AC7C59 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00