Flare-On 3: Challenge 6 Solution - khaki.exe

Challenge Author: Josh Homan

The sixth challenge, `khaki.exe`, is a fairly large Windows X86 executable that is approximately `3.6` MB. When running the program, we can see that it appears to be a simple guesser game. A sample execution is shown in Figure 1.

```
C:\>khaki.exe
(Guesses: 1) Pick a number between 1 and 100:50
 Too high, try again
(Guesses: 2) Pick a number between 1 and 100:25
Wahoo, you guessed it with 2 guesses
Status: 2 guesses
```

Figure 1: Running Kahki.exe

Moving past the sheer entertainment value of a guesser game, let's take a closer look at `khaki.exe`. Typically when I come across large files like this, I first attempt to determine why the file size is what it is. There a several reasons for an executable to have a large file size such as statically linking libraries, embedded resources or packing. After examining the strings for the `khaki.exe`, we can see some strings, shown in Figure 2, that suggest the executable is generated Py2Exe. This would explain the large file size based on the included Python interpreter.

```
PYTHON27.DLL
PY2EXE_VERBOSE
PYTHONSCRIPT
```

Figure 2: Strings suggesting Py2Exe

Opening the file with CFF Explorer, Figure 3, we can see the `khaki.exe` contains a resource named `PYTHONSCRIPT` starting with the four bytes "`12 34 56 78`". This is a good indication that executable is packed using Py2Exe.

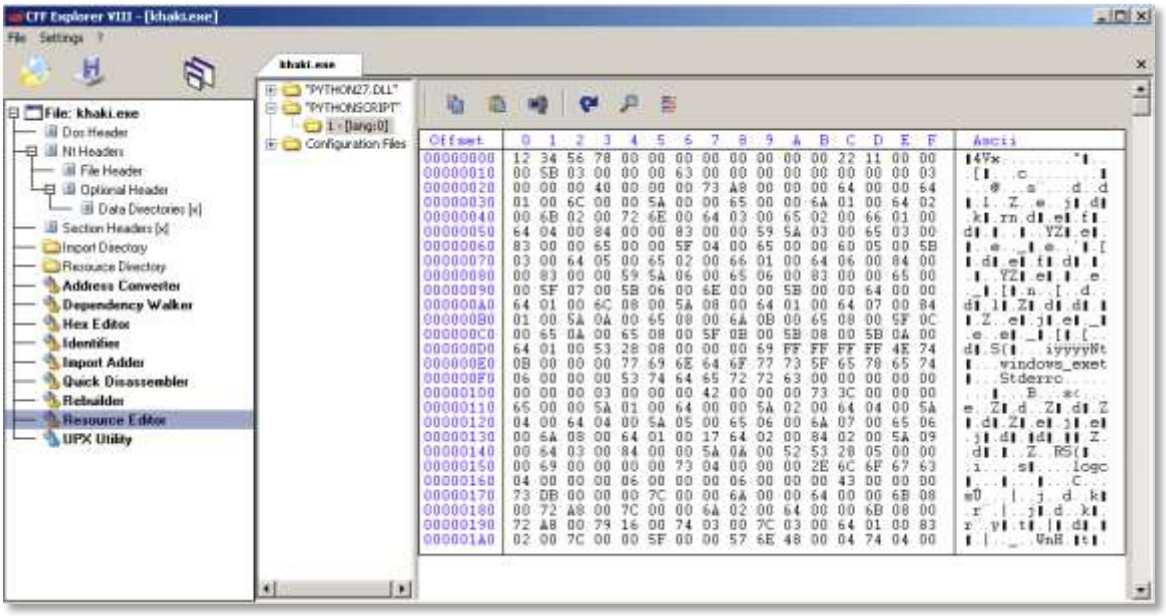

Figure 3: CFF Explorer displaying the PYTHONSCRIPT resource

Knowing the script is packed using Py2Exe, the next step is to try and extract the embedded Python script. The first thing to understand about a Py2exe executable is the script of interest is commonly located in the `PYTHONSCRIPT` resource. The resource starts with a 16-byte header followed by a NULL terminated archive name[1]. Immediately following the NULL terminated archive name is a Python marshal object containing Python code objects that are executed by Py2Exe. The marshal object for `kahki.exe` starts at offset `0x11` in Figure 3.

The `PYTHONSCRIPT` resource can easily be extracted using CFF Explorer.

[1] Details on the Py2Exe header can be viewed source located at https://sourceforge.net/p/py2exe/svn/HEAD/tree/trunk/py2exe/py2exe/build_exe.py#l850

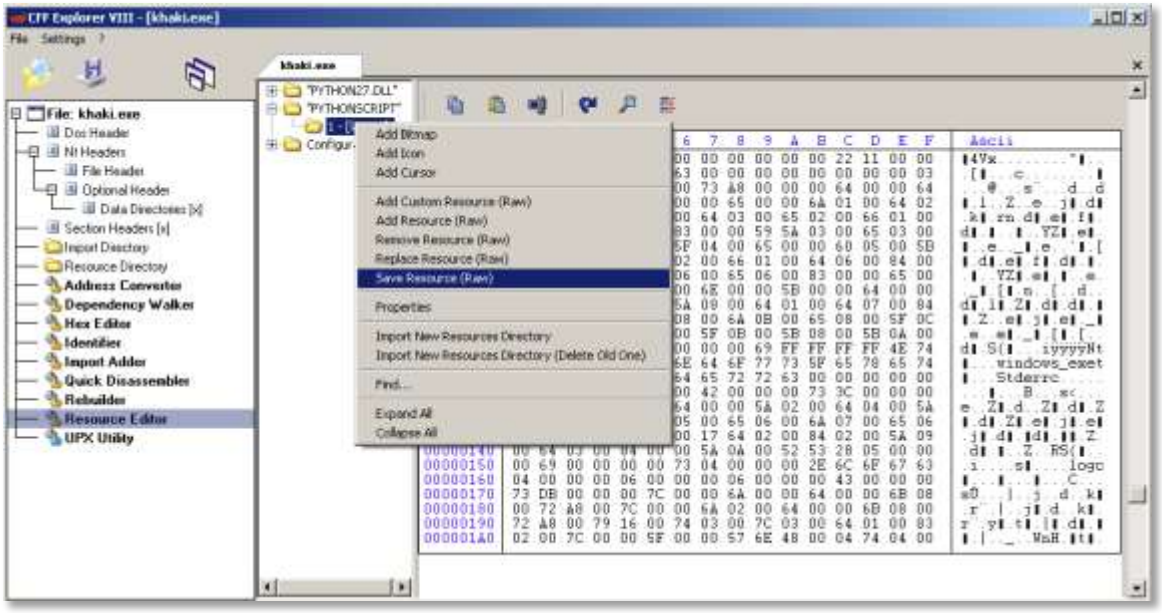

Figure 4: CFF Explorer to extract PYTHONSCRIPT resource

We can use the Python `marshal`[2] module to decode the resource and the `uncompyle6`[3] module to attempt to decompile the code object.

```
import sys, marshal, StringIO
from uncompyle6.main import uncompyle

if __name__ == "__main__":
    with open(sys.argv[1], "rb") as rsrc_file:
        py2exe_rsrc = rsrc_file.read()

    offset = py2exe_rsrc[0x010:].find("\x00")

    py2exe_code = marshal.loads(py2exe_rsrc[0x10 + offset + 1:])

    source = StringIO.StringIO()
    uncompyle(2.7, py2exe_code[-1], source)
    source = source.getvalue()
    print source
```

Figure 5: Python script to extract last code object and decompile

Running the script from Figure 5, we see that the decompilation was not successful.

[2] https://docs.python.org/2/library/marshal.html

[3] https://github.com/rocky/python-uncompyle6

```
$ python extract_py2exe.py khaki_PYTHONSCRIPT.bin
Traceback (most recent call last):
  File "extract_py2exe.py", line 22, in <module>
    uncompyle(float("2.7"), py2exe_code[-1], source)
  File "/lib/uncompyle6/main.py", line 39, in uncompyle
    code_objects=code_objects, is_pypy=is_pypy)
  File "/lib/uncompyle6/semantics/pysource.py", line 2450, in deparse_code
    tokens, customize = scanner.ingest(co, code_objects=code_objects)
  File "/lib/uncompyle6/scanners/scanner2.py", line 126, in ingest
    jump_targets = self.find_jump_targets()
  File "/lib/uncompyle6/scanners/scanner2.py", line 840, in find_jump_targets
    self.detect_structure(offset, op)
  File "/lib/uncompyle6/scanners/scanner2.py", line 617, in detect_structure
    jmp = self.next_except_jump(i)
  File "/lib/uncompyle6/scanners/scanner2.py", line 458, in next_except_jump
    self.jump_forward | frozenset([self.opc.RETURN_VALUE])
AssertionError
```

Figure 6: Exception generated on first decompile attempt

The next step is to take a look at the bytecode itself. The Python `dis`[4] module is a very helpful tool for disassembling Python code objects. A simple change to the script in Figure 5 will display the disassembly for the bytecode that is failing to decompile.

```
import sys, marshal, StringIO
import dis

if __name__ == "__main__":
    with open(sys.argv[1], "rb") as rsrc_file:
        py2exe_rsrc = rsrc_file.read()

    offset = py2exe_rsrc[0x010:].find("\x00")

    py2exe_code = marshal.loads(py2exe_rsrc[0x10 + offset + 1:])

    dis.dis(py2exe_code[-1])
```

Figure 7: Python script to disassemble code object

Running the above script on the `khaki.exe PYTHONSCRIPT` resource, we can immediately see some things that stand out.

[4] https://docs.python.org/2/library/dis.html

```
$ python dis_py2exe.py khaki_PYTHONSCRIPT.bin
  2           0 LOAD_CONST               0 (-1)
              3 LOAD_CONST               1 (None)
              6 IMPORT_NAME              0 (sys)
              9 STORE_NAME               0 (sys)
             12 LOAD_CONST               0 (-1)
             15 LOAD_CONST               0 (-1)
             18 POP_TOP
             19 LOAD_CONST               1 (None)
             22 ROT_TWO
             23 ROT_TWO
             24 IMPORT_NAME              1 (random)
             27 NOP
             28 STORE_NAME               1 (random)

  4          31 LOAD_CONST               2 ('Flare-On ultra python obfuscater
2000')
             34 STORE_NAME               2 (__version__)
             37 ROT_TWO
             38 ROT_TWO
<- Truncated ->
```

Figure 8: Python bytecode disassembly

One of the first things that stands out is the string `'Flare-On ultra python obfuscater 2000'`. This string was included to give a hint the code is packed and probably not typical Python bytecode and contains a spelling error, because of me not spell checking correctly.

Examining the bytecode more closely, we can see there are some odd code sequences that should not occur in legitimately generated bytecode. One of the first occurrences is at offset `15` with the `LOAD_CONST` followed by the `POP_TOP` instructions. This loads a constant onto the stack and immediately removes it, leaving the stack in its original state. The second is the `NOP` instruction at offset `27`. `NOP` instructions are not typically included in Python compilers. The next odd code sequence are the pair of `ROT_TWO` instructions starting at offset `37`. The `ROT_TWO` instruction rotates the top two elements on the stack. Executing two `ROT_TWO` instructions in sequence leaves the stack in its original state. These code sequences do not change the execution of the underlying program. They are essentially junk code, equivalent to NOP instructions, which are intended to confuse the compiler.

These code sequences cause the decompiler to fail because it relies on predictable output from the Python compiler to identify and produce source code. These additional instructions cause the decompiler to not match known code constructs and generate an exception.

To decompile the code object from `khaki.exe`, we need to remove the code sequences that are preventing the decompiler from producing source code. Modifying Python bytecode is not as simple and removing instructions or replacing instructions with `NOP` instructions. For example, when removing an instruction, the remaining code needs to be refactored to new offsets. Fortunately, the `bytecode_graph` module can do all of this for us. The module is available from `https://github.com/fireeye/flare-bytecode_graph`. In the examples folder, there is even a script `bytecode_deobf_blog.py`[5] that removes the bytecode instruction sequences that prevent `khaki.exe` from decompiling. Running the script against `khaki.exe` we get the source shown in Figure 9.

[5] https://github.com/fireeye/flare-bytecode_graph/blob/master/examples/bytecode_deobf_blog.py

```
#Embedded file name: poc.py
import sys, random
__version__ = 'Flare-On ultra python obfuscater 2000'
target = random.randint(1, 101)
count = 1
error_input = ''
while True:
    print '(Guesses: %d) Pick a number between 1 and 100:' % count,
    input = sys.stdin.readline()
    try:
        input = int(input, 0)
    except:
        error_input = input
        print 'Invalid input: %s' % error_input
        continue

    if target == input:
        break
    if input < target:
        print 'Too low, try again'
    else:
        print 'Too high, try again'
    count += 1

if target == input:
    win_msg = 'Wahoo, you guessed it with %d guesses\n' % count
    sys.stdout.write(win_msg)
if count == 1:
    print 'Status: super guesser %d' % count
    sys.exit(1)
if count > 25:
    print 'Status: took too long %d' % count
    sys.exit(1)
else:
    print 'Status: %d guesses' % count

if error_input != '':
    tmp = ''.join((chr(ord(x) ^ 66) for x in error_input)).encode('hex')
    if tmp != '312a232f272e27313162322e372548':
        sys.exit(0)

    stuffs = [67,139,119,165,232,86,207,61,79,67,45,58,230,190,181,74,65,
     148,71,243,246,67,142,60,61,92,58,115,240,226,171]

    import hashlib
    stuffer = hashlib.md5(win_msg + tmp).digest()
    for x in range(len(stuffs)):
        print chr(stuffs[x] ^ ord(stuffer[x % len(stuffer)])),

    print
```

Figure 9: Decoded Khaki.exe script

Now that the Python script for `khaki.exe` is decompiled, we can verify it does contain the functionality for the guesser game. The interesting part, besides the excitement of a guesser game, is the end of the script where it inspects the `error_input` variable. The script XORs the contents of the `error_input` variable with `66` and compares it with the hex string `312a232f272e27313162322e372548`. We can decode this string with the simple Python script shown in Figure 10.

```
>>> print(''.join((chr(ord(x) ^ 66) for x in
'312a232f272e27313162322e372548'.decode("hex"))))

shameless plug
```

Figure 10: XOR decoding

The `tmp` variable is then concatenated with the `win_msg` variable and result is MD5 hashed. The MD5 hash is then XORed onto the contents of the `stuffs` variable. Now that we understand the encoding we can write a small script to decode the `stuffs` variable. The script, shown in Figure 11, brute forces `win_msg` values from `0` to `25` and inspects the output for the string `flare-on.com`.

```
import hashlib

stuffs = [67,139,119,165,232,86,207,61,79,67,45,58,230,190,181,74,65,
 148,71,243,246,67,142,60,61,92,58,115,240,226,171]

for count in range(25):
    win_msg = 'Wahoo, you guessed it with %d guesses\n' % count

    md5_msg = hashlib.md5(win_msg + '312a232f272e27313162322e372548').digest()

    decoded = ""
    for x in range(len(stuffs)):
        decoded += chr(stuffs[x] ^ ord(md5_msg[x % len(md5_msg)]))

    if "@flare-on.com" in decoded:
        print("%d:%s" % (count,decoded))
        break
```

Figure 11: Key decoding script

After running the script in Figure 11, we get the output shown in Figure 12.

```
11:1mp0rt3d_pygu3ss3r@flare-on.com
```

Figure 12: Decoded key

Based on the output, we can see the key is `1mp0rt3d_pygu3ss3r@flare-on.com`. To obtain the key

from the command line, we have to satisfy a couple variables. First, the script requires a non-integer input with the contents "`shameless plug`". Second, the number of guesses to win the game must be 11. A complete run of the program is shown in Figure 13.

```
C:\>khaki.exe
(Guesses: 1) Pick a number between 1 and 100:50
 Too low, try again
(Guesses: 2) Pick a number between 1 and 100:75
 Too low, try again
(Guesses: 3) Pick a number between 1 and 100:85
 Too low, try again
(Guesses: 4) Pick a number between 1 and 100:90
 Too low, try again
(Guesses: 5) Pick a number between 1 and 100:95
 Too high, try again
(Guesses: 6) Pick a number between 1 and 100:93
 Too low, try again
(Guesses: 7) Pick a number between 1 and 100:shameless plug
 Invalid input: shameless plug

(Guesses: 7) Pick a number between 1 and 100:93
 Too low, try again
(Guesses: 8) Pick a number between 1 and 100:93
 Too low, try again
(Guesses: 9) Pick a number between 1 and 100:93
 Too low, try again
(Guesses: 10) Pick a number between 1 and 100:93
 Too low, try again
(Guesses: 11) Pick a number between 1 and 100:94
Wahoo, you guessed it with 11 guesses
Status: 11 guesses

1 m p 0 r t 3 d _ p y g u 3 s s 3 r @ f l a r e - o n . c o m
```

Figure 13: Executing khaki.exe to get the key

One final note, the name khaki.exe does not have anything to do with the challenge. Nick Harbour chose the name out of admiration of my somewhat reliable fashion choice of khaki pants.