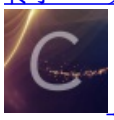
【WhaleCTF逆向题】第一期py转子——大结局！ writeup

转载

iqiqiya 于 2018-09-22 15:34:12 发布 876 收藏
分类专栏： 我的逆向之路 我的CTF之路 我的CTF进阶之路 文章标签： 【WhaleCTF逆向题】第一期py转子——大结局！ writ py转子——大结局！ writeup


我的逆向之路 同时被 3 个专栏收录
108 篇文章 10 订阅
订阅专栏


我的CTF之路
92 篇文章 5 订阅
订阅专栏

我的CTF进阶之路
108 篇文章 18 订阅
订阅专栏

题目信息：

这个是0CTF的一道题

转载自http://hebin.me/2018/01/24/%E8%A5%BF%E6%99%AEctf-py137/

The archive contains a compiled Python file `crypt.pyc` and an encrypted flag file `encrypted_flag`.

As the description states, the compiled Python code is mangled. Using uncompyle6 gives an error:

```
$ uncompyle6 crypt.pyc
# uncompyle6 version 2.9.9
# Python bytecode 2.7 (62211)
# Decompiled from: Python 2.7.13 (default, Jan 19 2017, 14:48:08)
# [GCC 6.3.0 20170118]
# Embedded file name: /Users/hen/Lab/0CTF/py/crypt.py
# Compiled at: 2017-01-06 01:08:38
Traceback (most recent call last):
  File "/usr/local/bin/uncompyle6", line 11, in <module>;
    sys.exit(main_bin())
  File "/usr/local/lib/python2.7/dist-packages/uncompyle6/bin/uncompile.py", line 163, in main_bin
    **options)
  File "/usr/local/lib/python2.7/dist-packages/uncompyle6/main.py", line 145, in main
    uncompyle_file(infile, outstream, showasm, showast, showgrammar)
  File "/usr/local/lib/python2.7/dist-packages/uncompyle6/main.py", line 72, in uncompyle_file
    is_pypy=is_pypy, magic_int=magic_int)
  File "/usr/local/lib/python2.7/dist-packages/uncompyle6/main.py", line 46, in uncompyle
    is_pypy=is_pypy)
  File "/usr/local/lib/python2.7/dist-packages/uncompyle6/semantics/pysource.py", line 2254, in deparse_cod
    tokens, customize = scanner.ingest(co, code_objects=code_objects, show_asm=showasm)
  File "/usr/local/lib/python2.7/dist-packages/uncompyle6/scanners/scanner2.py", line 230, in ingest
    pattr = free[oparg]
IndexError: tuple index out of range
```

Let's turn to a Python disassembler to dump the file structure. Because the opcodes are messed up, let's comment out the part of the disassembler code that prints out the bytecode (`dis.disassemble(code)`). It was also necessary to comment out some failing timestamp analysis code, which was not important.

Running the disassembler over the challenge file gives the following result:

```
$ python dec.py crypt.pyc
magic 03f30d0a
moddate 66346f58 (0)
code
   argcount 0
   nlocals 0
   stacksize 2
   flags 0040
   code
      990000990100860000910000990200880000910100990300880000910200
      99010053
   consts
      -1
      None
      code
         argcount 1
         nlocals 6
         stacksize 3
         flags 0043
         code
            990100680100990200680200990300680300610100990400469905002761
            020061010027610300279906004627990500276102009906004627990700
            276804009b0000600100610400830100680500610500600200610000 8301
            0053
         consts
            None
            '!@#$%^&amp;*'
```

```
            'abcdefgh'
            '<>{}:"'
            4
            '|'
            2
            'EOF'
         names ('rotor', 'newrotor', 'encrypt')
         varnames ('data', 'key_a', 'key_b', 'key_c', 'secret', 'rot')
         freevars ()
         cellvars ()
         filename '/Users/hen/Lab/0CTF/py/crypt.py'
         name 'encrypt'
         firstlineno 2
         lnotab 00010601060106012e010f01
      code
         argcount 1
         nlocals 6
         stacksize 3
         flags 0043
         code
            990100680100990200680200990300680300610100990400469905002761
            020061010027610300279906004627990500276102009906004627990700
            276804009b0000600100610400830100680500610500600200610000 8301
            0053
         consts
            None
            '!@#$%^&amp;*'
            'abcdefgh'
            '<>{}:"'
            4
            '|'
            2
            'EOF'
         names ('rotor', 'newrotor', 'decrypt')
         varnames ('data', 'key_a', 'key_b', 'key_c', 'secret', 'rot')
         freevars ()
         cellvars ()
         filename '/Users/hen/Lab/0CTF/py/crypt.py'
         name 'decrypt'
         firstlineno 10
         lnotab 00010601060106012e010f01
   names ('rotor', 'encrypt', 'decrypt')
   varnames ()
   freevars ()
   cellvars ()
   filename '/Users/hen/Lab/0CTF/py/crypt.py'
   name '<module>'
   firstlineno 1
   lnotab 0c010908
```

There is lots of interesting info that we can glean from this output (and by reading the Python opcode documentation and source code):

- this file uses the rotor library and defines 2 methods - `encrypt` and `decrypt`
- `encrypt` and `decrypt` method bodies look almost identical; naturally we need to look at `decrypt` closely
- `rotor` functions `newrotor` and `decrypt` are used
- there are some kinds of key variables `key_a`, `key_b` and `key_c` being used, and also something called a `secret` (a

decryption key?)

- there are some interesting constants embedded in the code, including `!@#$%^&*`, `abcdefgh`, and `<>{}:"`; do these correspond to `key_a` through `_c`?

In order to start analyzing the code let's create our own decryption function and decompile it. It would likely have the following components:

- a parameter passed in
- a result string returned
- some constant initialization
- some operations to create a decryption key
- code to decrypt the data

Here's a first version of that code. We will make some assumptions about how the variables in the challenge file (`data`, `secret`, etc.) are actually used:

```
import rotor
 def decrypt(data):
 key_a = '!@#$%^&*'
 key_b = 'abcdefgh'
 key_c = '<>{}:"'
 secret = key_a + key_b + key_c
 rot = rotor.newrotor(secret)
 return rot.decrypt(data)

enc = open("encrypted_flag", "rb").read()
print decrypt(enc)
```

Let's re-enable opcode analysis in the disassembler and run it over this new file:

```
$ python -c "import py_compile;py_compile.compile('ex.py');"; python dec.py ex.pyc
magic 03f30d0a
moddate e2f2cf58 (0)
code
   argcount 0
   nlocals 0
   stacksize 3
   flags 0040
   code
      6400006401006c00005a0000640200840000 5a0100650200640300640400
      8302006a03008300005a0400650100650400830100474864010053
  1           0 LOAD_CONST               0 (-1)
              3 LOAD_CONST               1 (None)
              6 IMPORT_NAME              0 (rotor)
              9 STORE_NAME               0 (rotor)

  3          12 LOAD_CONST               2 (<code object decrypt at 0x7f664ada8530, file "ex.py", line 3>)
             15 MAKE_FUNCTION            0
             18 STORE_NAME               1 (decrypt)

 13          21 LOAD_NAME                2 (open)
             24 LOAD_CONST               3 ('encrypted_flag')
             27 LOAD_CONST               4 ('rb')
             30 CALL_FUNCTION            2
             33 LOAD_ATTR                3 (read)
             36 CALL_FUNCTION            0
```

```
            39 STORE_NAME               4 (enc)

15          42 LOAD_NAME                1 (decrypt)
            45 LOAD_NAME                4 (enc)
            48 CALL_FUNCTION            1
            51 PRINT_ITEM
            52 PRINT_NEWLINE
            53 LOAD_CONST               1 (None)
            56 RETURN_VALUE
  consts
     -1
     None
     code
        argcount 1
        nlocals 6
        stacksize 2
        flags 0043
        code
           6401007d01006402007d02006403007d03007c01007c0200177c0300177d
           04007400006a01007c04008301007d05007c05006a02007c000083010053
 4           0 LOAD_CONST               1 ('!@#$%^&amp;*')
             3 STORE_FAST               1 (key_a)

 5           6 LOAD_CONST               2 ('abcdefgh')
             9 STORE_FAST               2 (key_b)

 6          12 LOAD_CONST               3 ('<>{}:"')
            15 STORE_FAST               3 (key_c)

 8          18 LOAD_FAST                1 (key_a)
            21 LOAD_FAST                2 (key_b)
            24 BINARY_ADD
            25 LOAD_FAST                3 (key_c)
            28 BINARY_ADD
            29 STORE_FAST               4 (secret)

10          32 LOAD_GLOBAL              0 (rotor)
            35 LOAD_ATTR                1 (newrotor)
            38 LOAD_FAST                4 (secret)
            41 CALL_FUNCTION            1
            44 STORE_FAST               5 (rot)

11          47 LOAD_FAST                5 (rot)
            50 LOAD_ATTR                2 (decrypt)
            53 LOAD_FAST                0 (data)
            56 CALL_FUNCTION            1
            59 RETURN_VALUE
        consts
           None
           '!@#$%^&amp;*'
           'abcdefgh'
           '<>{}:"'
        names ('rotor', 'newrotor', 'decrypt')
        varnames ('data', 'key_a', 'key_b', 'key_c', 'secret', 'rot')
        freevars ()
        cellvars ()
        filename 'ex.py'
        name 'decrypt'
        firstlineno 3
        lnotab 0001060106010602 0e020f01
```

```
    'encrypted_flag'
    'rb'
names ('rotor', 'decrypt', 'open', 'read', 'enc')
varnames ()
freevars ()
cellvars ()
filename 'ex.py'
name '<module>'
firstlineno 1
lnotab 0c02090a1502
```

Disassembler gives us more useful information:

- most instructions are either a single opcode, or an opcode and a 2-byte parameter
- parameter offsets are 0-based (duh)
- Python instructions heavily use the stack - data is pushed on it and many opcodes process the top one or two items on the stack
- at least one opcode (`53` - `RETURN_VALUE`) was not obfuscated - it's the same in the code for the challenge
- some of our guesses about names and meanings of different variables worked - `names` and `varnames` fields match corresponding challenge file fields perfectly

Now we will break down the challenge binary opcode stream into individual operations and try to match them to instructions in our decryption code. Looks like `rotor` call functionality can be matched directly (assuming we are correct about `secret` being the decryption key):

```
990100
680100
990200
680200
990300
680300
610100
990400
46
990500
27
610200
610100
27
610300
27
990600
46
27
990500
27
610200
990600
46
27
990700
27
680400 STORE_NAME 4 (secret)
9b0000 LOAD_GLOBAL 0 (rotor)
600100 LOAD_ATTR 1 (newrotor)
610400 LOAD_FAST 4 (secret)
830100 CALL_FUNCTION 1
680500 STORE_NAME 5 (rot)
610500 LOAD_FAST 5 (rot)
600200 LOAD_ATTR 2 (decrypt)
610000 LOAD_FAST 0 (data)
830100 CALL_FUNCTION 1
53 RETURN_VALUE
```

This gives us the following translation for opcodes:

- `68` - `STORE_NAME`
- `9b` - `LOAD_GLOBAL`
- `60` - `LOAD_ATTR`
- `61` - `LOAD_FAST`
- `83` - `CALL_FUNCTION` (was not obfuscated)

Also it looks like `99` is actually `LOAD_CONST`. Let's fill in this information:

```
990100 LOAD_CONST 1 ('!@#$%^&amp;*')
680100 STORE_NAME 1 (key_a)
990200 LOAD_CONST 2 ('abcdefgh')
680200 STORE_NAME 2 (key_b)
990300 LOAD_CONST 3 ('<>{}:"')
680300 STORE_NAME 3 (key_c)
610100 LOAD_FAST 1 (key_a)
990400 LOAD_CONST 4 (4)
46
990500 LOAD_CONST 5 ('|')
27
610200 LOAD_FAST 2 (key_b)
610100 LOAD_FAST 1 (key_a)
27
610300 LOAD_FAST 3 (key_c)
27
990600 LOAD_CONST 6 (2)
46
27
990500 LOAD_CONST 5 ('|')
27
610200 LOAD_FAST 2 (key_b)
990600 LOAD_CONST 6 (2)
46
27
990700 LOAD_CONST 7 ('EOF')
27
680400 STORE_NAME 4 (secret)
9b0000 LOAD_GLOBAL 0 (rotor)
600100 LOAD_ATTR 1 (newrotor)
610400 LOAD_FAST 4 (secret)
830100 CALL_FUNCTION 1
680500 STORE_NAME 5 (rot)
610500 LOAD_FAST 0 (rot)
600200 LOAD_ATTR 2 (decrypt)
610000 LOAD_FAST 0 (data)
830100 CALL_FUNCTION 1
53 RETURN_VALUE
```

This looks very promising. We only need to find what kinds of manipulations are done on the encryption key and we will be done.

Opcodes `46` and `27` are the remaining unknowns. `46` works on a string and a numeric argument, and `27` works on 2 strings. Essentially we have the following expression:

```
secret = (key_a OP46 4) OP27 '|' OP27 ((key_b OP27 key_a OP27 key_c) OP46 2) OP27 '|' OP27 (key_b OP46 2) O
```

After some trial and error with our code, we find that `46` is a multiplication operation, and `27` is an addition. This gives us the following final version of decryption code:

```
import rotor
def decrypt(data):
 key_a = '!@#$%^&*'
 key_b = 'abcdefgh'
 key_c = '<>{}:"'
 secret = key_a*4 + '|' + (key_b+key_a+key_c)*2 + '|' + key_b*2 + 'EOF'
 rot = rotor.newrotor(secret)
 return rot.decrypt(data)
i = open("encrypted_flag", "rb").read()
print decrypt(i)
```