FLARE-ON CHALLENGE 9 SOLUTION

BY PAUL TARTER (@HEFRPIDGE)

Challenge 11

Challenge Prompt

Protection, Obfuscation, Restrictions... Oh my!!

The good part about this one is that if you fail to solve it I don't need to ship you a prize.

7-zip password: flare

Solution

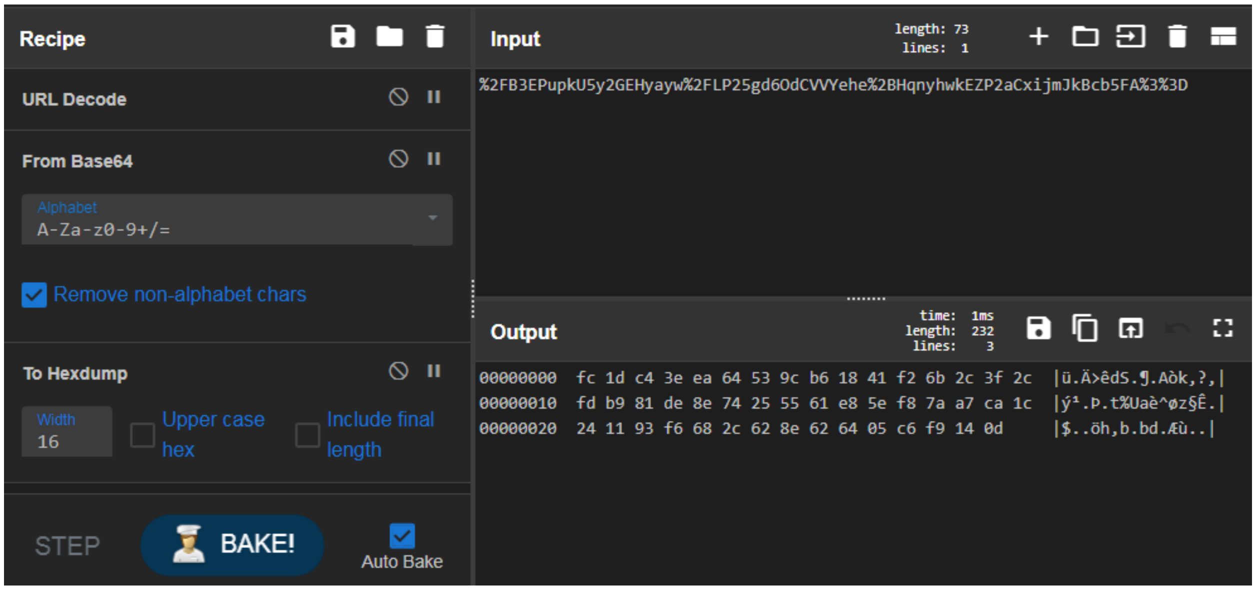
This challenge is a PyArmor protected challenge packed with PyInstaller. PyArmor does a great job at obfuscating and protecting python samples, sadly for this last challenge there were a couple unforeseen shortcuts to solve this challenge quickly. This sample was based off a PyArmor sample that was analyzed that included multiple modules to form a decent backdoor with anti-analysis too. To defeat the anti-analysis I approached it the same way as most did in this challenge as replacing the module with their own; This was not the intended solution. In the malware we needed to understand the capabilities of all the modules, without returning the code to source I modified the *cpython* source and built my own python libraries to create a tracing utility for Python. Some might note that there is tracing available in *cpython*, this is also protected against in PyArmor.

Simple Solution

Running this sample with *FakeNet* running sends out the following network traffic

```
11/04/22 11:00:42 AM [        DNS Server] Received A request for domain
'www.evil.flare-on.com'.
11/04/22 11:00:42 AM [          Diverter] 11.exe (3272) requested TCP 192.0.2.123:80
11/04/22 11:00:42 AM [    HTTPListener80]   POST / HTTP/1.1
11/04/22 11:00:42 AM [    HTTPListener80]   Host: www.evil.flare-on.com
11/04/22 11:00:42 AM [    HTTPListener80]   User-Agent: python-requests/2.28.1
11/04/22 11:00:42 AM [    HTTPListener80]   Accept-Encoding: gzip, deflate
11/04/22 11:00:42 AM [    HTTPListener80]   Accept: */*
11/04/22 11:00:42 AM [    HTTPListener80]   Connection: keep-alive
11/04/22 11:00:42 AM [    HTTPListener80]   Content-Length: 79
11/04/22 11:00:42 AM [    HTTPListener80]   Content-Type: application/x-www-form-
urlencoded
11/04/22 11:00:42 AM [    HTTPListener80]
11/04/22 11:00:42 AM [    HTTPListener80]
flag=%2FB3EPupkU5y2GEHyayw%2FLP25gd6OdCVVYehe%2BHqnyhwkEZP2aCxijmJkBcb5FA%3
%3D
```

Running that through *CyberChef* yields some obviously encrypted data



```
length: 73
lines:  1
%2FB3EPupkU5y2GEHyayw%2FLP25gd6OdCVVYehe%2BHqnyhwkEZP2aCxijmJkBcb5FA%3%3D
```

```
time:   1ms
length: 232
lines:    3
00000000  fc 1d c4 3e ea 64 53 9c b6 18 41 f2 6b 2c 3f 2c  |ü.Ä>êdS.¶.Aòk,?,|
00000010  fd b9 81 de 8e 74 25 55 61 e8 5e f8 7a a7 ca 1c  |ý¹.Þ.t%Uaè^øz§Ê.|
00000020  24 11 93 f6 68 2c 62 8e 62 64 05 c6 f9 14 0d     |$..öh,b.bd.Æù..|
```

Triage

Identifying this sample is packed with *PyInstaller* can be evident from some giveaway strings

```
_MEIPASS2
Cannot open PyInstaller archive from executable (%s) or external archive (%s)
```

Identifying this sample is a PyArmor protected sample can even be determined from strings

```
pyarmor
PYARMOR
```

Looking at the strings output searching for *.py will show an interesting file with the name *crypt.py*

Knowing this is PyInstaller packed, a very useful script to use is *pyinstxtractor.py* from https://github.com/extremecoders-re/pyinstxtractor. Using this script is as simple as

```
python pyinstxtractor.py 11.exe
```

The output folder will be labeled *11.exe_extracted*. Within this directory is a fully portable python 3.7 framework. The directory contains the main script, *script 11.pyc*. This can be executed with python and will output some very useful information.

```
c:\sandbox\11.exe_extracted>python 11.pyc
Traceback (most recent call last):
  File "<dist\obf\11.py>", line 2, in <module>
  File "<frozen 11>", line 3, in <module>
  File "C:\Python37\lib\crypt.py", line 3, in <module>
    import _crypt
ModuleNotFoundError: No module named '_crypt'
```

pyinstxtractor.py places frozen module in an archive so they are not immediately able to be loaded. This folder is called *PYZ-00.pyz_extracted*. Copy crypt.py out of that folder into the working directory and the sample will work again. This can be verified by the output in Fakenet.

```
c:\sandbox\11.exe_extracted>copy PYZ-00.pyz_extracted\crypt.pyc .\
        1 file(s) copied.

c:\sandbox\11.exe_extracted>python 11.pyc

c:\sandbox\11.exe_extracted>
```

This sample of PyArmor was protected at restriction level 2. This means that the sample cannot be imported into a python interpreter and inspected.

```
c:\sandbox\11.exe_extracted>python
Python 3.7.9 (tags/v3.7.9:13c94747c7, Aug 17 2020, 18:58:18) [MSC v.1900 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import crypt
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<crypt.py>", line 2, in <module>
RuntimeError: Check restrict mode of module failed
>>>
```

The module crypt.py can be written instead of using the provided to help find out what is being called and try to leak information.

```
c:\sandbox\11.exe_extracted>copy NUL crypt.py
        1 file(s) copied.

c:\sandbox\11.exe_extracted>python 11.pyc
Traceback (most recent call last):
  File "<dist\obf\11.py>", line 2, in <module>
  File "<frozen 11>", line 12, in <module>
AttributeError: module 'crypt' has no attribute 'ARC4'
```

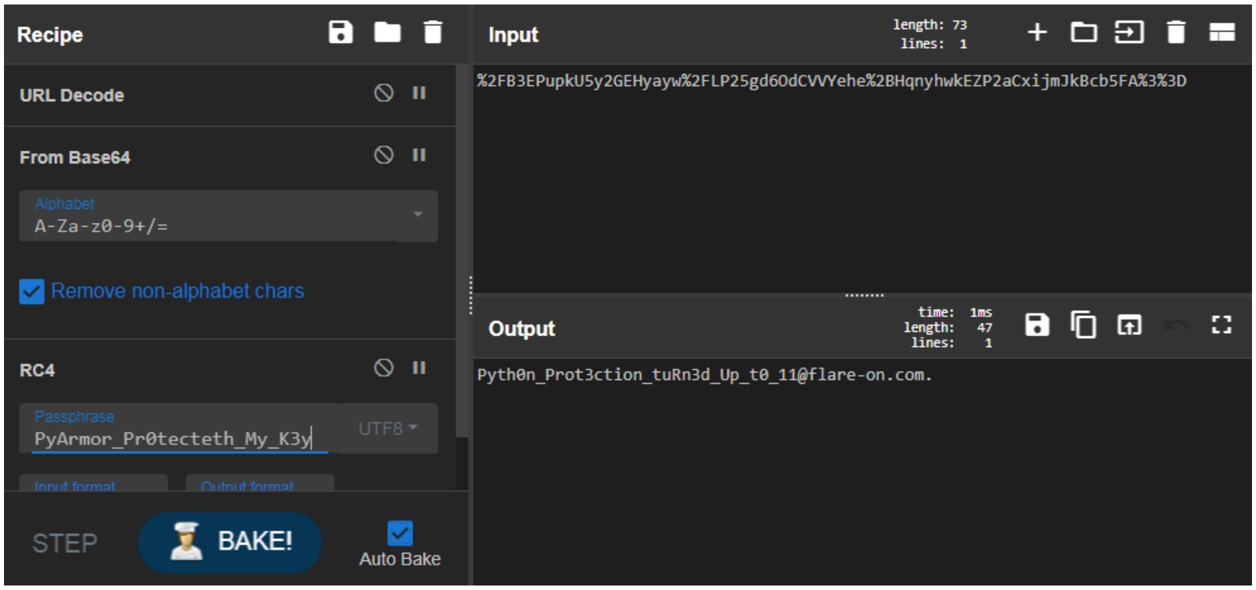
This sample wants something called *ARC4*. Try it as a function first with the following source code

```
from hexdump import hexdump
```

```
def ARC4(arg):
    hexdump(arg)
```

```
c:\sandbox\11.exe_extracted>python 11.pyc
00000000: 50 79 41 72 6D 6F 72 5F  50 72 30 74 65 63 74 65  PyArmor_Pr0tecte
00000010: 74 68 5F 4D 79 5F 4B 33  79                       th_My_K3y
Traceback (most recent call last):
  File "<dist\obf\11.py>", line 2, in <module>
  File "<frozen 11>", line 15, in <module>
AttributeError: 'NoneType' object has no attribute 'encrypt'
```

Just from this alone, one can assume this is an RC4 key for the encrypted traffic, trying that in Cyberchef yields the following


Recipe: URL Decode; From Base64 (Alphabet A-Za-z0-9+/=, Remove non-alphabet chars ✓); RC4 (Passphrase PyArmor_Pr0tecteth_My_K3y, UTF8)

Input (length: 73, lines: 1):

```
%2FB3EPupkU5y2GEHyayw%2FLP25gd6OdCVVYehe%2BHqnyhwkEZP2aCxijmJkBcb5FA%3%3D
```

Output (time: 1ms, length: 47, lines: 1):

```
Pyth0n_Prot3ction_tuRn3d_Up_t0_11@flare-on.com.
```

Let's imagine the flag that wasn't just retrieved easy. Go a step further and try to call *encrypt*. The *AttributeError* shows that it was trying to call *encrypt* on an object. This would mean instead of just creating another function called encrypt one needs to create a class for *ARC4* and then create a method for the class named *encrypt*.

```
class ARC4():
    def __init__(self, key):
        print("key: {}".format(key.decode()))

    def encrypt(self, flag):
        print("flag: {}".format(flag.decode()))
        return flag
```

```
c:\sandbox\11.exe_extracted>python 11.pyc
key: PyArmor_Pr0tecteth_My_K3y
flag: Pyth0n_Prot3ction_tuRn3d_Up_t0_11@flare-on.com
```

```
11/04/22 11:27:59 AM [        DNS Server] Received A request for domain
'www.evil.flare-on.com'.
11/04/22 11:27:59 AM [          Diverter] python.exe (2064) requested TCP
192.0.2.123:80
11/04/22 11:27:59 AM [    HTTPListener80]   POST / HTTP/1.1
11/04/22 11:27:59 AM [    HTTPListener80]   Host: www.evil.flare-on.com
11/04/22 11:27:59 AM [    HTTPListener80]   User-Agent: python-requests/2.25.1
11/04/22 11:27:59 AM [    HTTPListener80]   Accept-Encoding: gzip, deflate
11/04/22 11:27:59 AM [    HTTPListener80]   Accept: */*
11/04/22 11:27:59 AM [    HTTPListener80]   Connection: keep-alive
11/04/22 11:27:59 AM [    HTTPListener80]   Content-Length: 73
11/04/22 11:27:59 AM [    HTTPListener80]   Content-Type: application/x-www-form-
urlencoded
11/04/22 11:27:59 AM [    HTTPListener80]
11/04/22 11:27:59 AM [    HTTPListener80]
flag=UHl0aDBuX1Byb3QzY3Rpb25fdHVSbjNkX1VwX3QwXzExQGZsYXJlLW9uLmNvbQ%3D%3D
```

```
>>> from base64 import b64decode
>>> from urllib.parse import unquote
>>>
print(b64decode(unquote('UHl0aDBuX1Byb3QzY3Rpb25fdHVSbjNkX1VwX3QwXzExQGZsYXJlLW9uLmNvbQ%3D%3D
')))
b'Pyth0n_Prot3ction_tuRn3d_Up_t0_11@flare-on.com'
```

The final flag for this challenge is:

```
Pyth0n_Prot3ction_tuRn3d_Up_t0_11@flare-on.com
```

Complex Solution

This sample was built in Advanced mode with its own licensed capsule so when PyArmor remapped the bytecode open projects could not easily remap and create a readable pyc file.

PyArmor by default does not protect python modules or builtins. This means that data can be leaked if one traces python. If one builds cpython project themselves, they can enable tracing; PyArmor Blocks this. One can implement their own tracing and that was a approach for solving this challenge. As mentioned before, a lot more information was needed in other PyArmor samples analyzed and this approach was found to be very useful.

The function *PyEval_EvalFrameDefault* evaluates all code blocks. The code is still encrypted at this point with PyArmor, but other code block data is not. Given this information one can inspect code blocks just like we would with a python interpreter. The following is logged from a tracing utility

```
[CODE OBJECT]
  FileName: <frozen 11>
  Name: <module>
  Names:
    crypt
    base64
    requests
    config
    ARC4
    cipher
    b64encode
    encrypt
    flag
    post
    exceptions
    RequestException
    e
    Exception
    __armor_wrap__

[CODE OBJECT]
  FileName: <frozen crypt>
  Name: <module>
  Names:
    ARC4
    __armor_wrap__

[CODE OBJECT]
  FileName: <frozen crypt>
  Name: ARC4
  Names:
    __name__
    __module__
    __qualname__
    __init__
    PRGA
    encrypt
    __armor_wrap__
```

This information is very helpful for triage analysis of modules. PyEval_EvalFrameDefault can be further traced to handle any opcode. Methods and Functions are handled in many locations mostly originating from *call.c*. Some strategic locations allow a trace of the calls with input and return data to be analyzed. As noted above the protected code is run inside of a PyArmor runtime library that implements its own python framework with mixed opcode handling. Although if a call to any builtin or non-protected module takes place the data from that is leaked. RC4 was implemented instead of using a module like PyCryptoDome, this was for two reasons: keep the sample smaller and to introduce this data leak easily.

```
key_len = len(key)
```

The line above would leak the key outside of PyArmor protected runtime and is exemplified in the following excerpt from a trace dump

```
[CALL_FUNCTION]
  __armor_wrap__

[CALL_FUNCTION]
  len
    bytes:
      [032f83a0]    50 79 41 72 6D 6F 72 5F   50 72 30 74 65 63 74 65   PyArmor_ Pr0tecte
      [032f83b0]    74 68 5F 4D 79 5F 4B 33   79                        th_My_K3 y

    (len) ==> int: 25 (len)

    (__armor_wrap__) ==> int: 46

  [CALL_FUNCTION]
  b2a_base64
    bytes:
      [0049b420]    FC 1D C4 3E EA 64 53 9C   B6 18 41 F2 6B 2C 3F 2C   .....dS. ..A.k...
      [0049b430]    FD B9 81 DE 8E 74 25 55   61 E8 5E F8 7A A7 CA 1C   .....t.U a...z...
      [0049b440]    24 11 93 F6 68 2C 62 8E   62 64 05 C6 F9 14         ....h.b. bd....

    (b2a_base64) ==> bytes:
    [00461cd0]    42 33 45 50 75 70 6B   55 35 79 32 47 45 48 79      .B3EPupk U5y2GEHy
    [00461ce0]    61 79 77 w4C 50 32 35   67 64 36 4F 64 43 56 56     ayw.LP25 gd6OdCVV
    [00461cf0]    59 65 68 65 e48 71 6E   79 68 77 6B 45 5A 50 32     Yehe.Hqn yhwkEZP2
    [00461d00]    61 43 78 69 6A 6D 4A 6B   42 63 62 35 46 41 A.      aCxijmJk Bcb5FA..
```

©2021 Mandiant, Inc. All rights reserved. Mandiant is a registered trademark of Mandiant, Inc. All other brands, products, or service names are or may be trademarks or service marks of their respective owners.