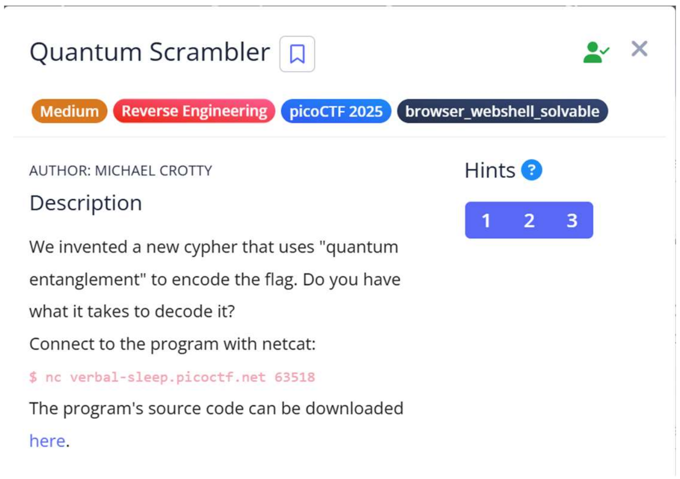
Quantum Scrambler

Written by: *Ach**i***de~~on~~

The challenge is taken from picoCTF, this one is pretty new considering it's from the 2025 competition



Quantum Scrambler

Medium · Reverse Engineering · picoCTF 2025 · browser_webshell_solvable

AUTHOR: MICHAEL CROTTY

Description

We invented a new cypher that uses "quantum entanglement" to encode the flag. Do you have what it takes to decode it?

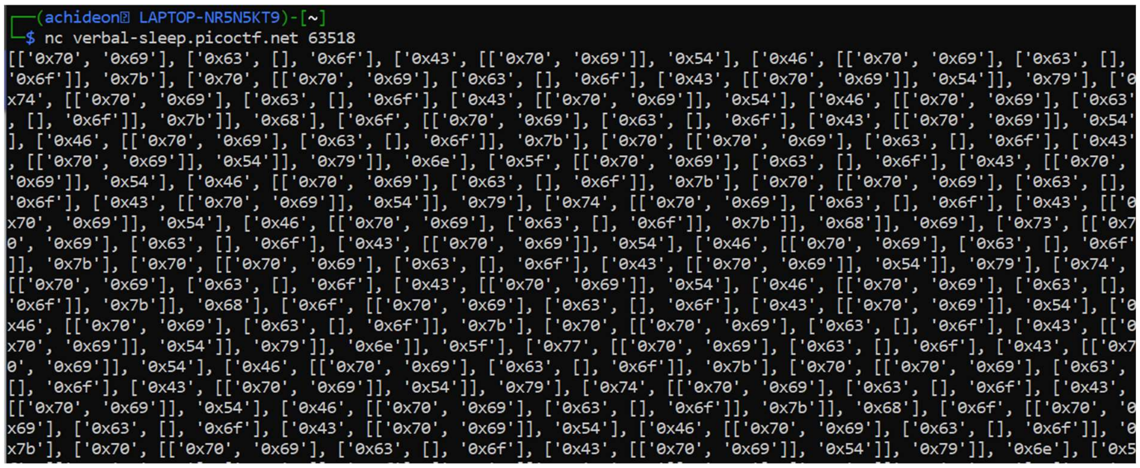
Connect to the program with netcat:

`$ nc verbal-sleep.picoctf.net 63518`

The program's source code can be downloaded here.

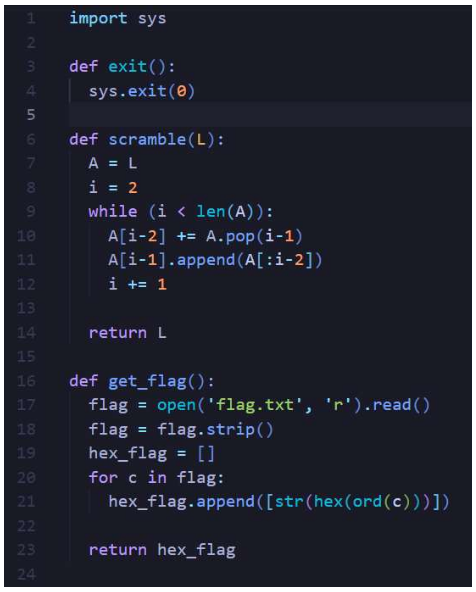
The file itself is a source code written in **python**

Upon checking the netcat, you will be greeted with a very long text of hex stored in arrays. As seen below, the arrays were very complicated, some are nested, and this text goes on for multiple lines (I don't bother counting them)

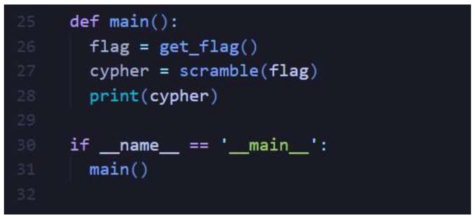


```
┌──(achideon⊡ LAPTOP-NR5N5KT9)-[~]
└─$ nc verbal-sleep.picoctf.net 63518
[['0x70', '0x69'], ['0x63', [], '0x6f'], ['0x43', [['0x70', '0x69']], '0x54'], ['0x46', [['0x70', '0x69'], ['0x63', [],
'0x6f']], '0x7b'], ['0x70', [['0x70', '0x69'], ['0x63', [], '0x6f'], ['0x43', [['0x70', '0x69']], '0x54']], '0x79'], ['0
x74', [['0x70', '0x69'], ['0x63', [], '0x6f'], ['0x43', [['0x70', '0x69']], '0x54'], ['0x46', [['0x70', '0x69'], ['0x63'
, [], '0x6f']], '0x7b']], '0x68'], ['0x6f', [['0x70', '0x69'], ['0x63', [], '0x6f'], ['0x43', [['0x70', '0x69']], '0x54'
], ['0x46', [['0x70', '0x69'], ['0x63', [], '0x6f']], '0x7b'], ['0x70', [['0x70', '0x69'], ['0x63', [], '0x6f'], ['0x43'
, [['0x70', '0x69']], '0x54']], '0x79']], '0x6e'], ['0x5f', [['0x70', '0x69'], ['0x63', [], '0x6f'], ['0x43', [['0x70',
'0x69']], '0x54'], ['0x46', [['0x70', '0x69'], ['0x63', [], '0x6f']], '0x7b'], ['0x70', [['0x70', '0x69'], ['0x63', [],
'0x6f'], ['0x43', [['0x70', '0x69']], '0x54']], '0x79'], ['0x74', [['0x70', '0x69'], ['0x63', [], '0x6f'], ['0x43', [['0
x70', '0x69']], '0x54'], ['0x46', [['0x70', '0x69'], ['0x63', [], '0x6f']], '0x7b']], '0x68']], '0x69'], ['0x73', [['0x7
0', '0x69'], ['0x63', [], '0x6f'], ['0x43', [['0x70', '0x69']], '0x54'], ['0x46', [['0x70', '0x69'], ['0x63', [], '0x6f'
]], '0x7b'], ['0x70', [['0x70', '0x69'], ['0x63', [], '0x6f'], ['0x43', [['0x70', '0x69']], '0x54']], '0x79'], ['0x74',
[['0x70', '0x69'], ['0x63', [], '0x6f'], ['0x43', [['0x70', '0x69']], '0x54'], ['0x46', [['0x70', '0x69'], ['0x63', [],
'0x6f']], '0x7b']], '0x68'], ['0x6f', [['0x70', '0x69'], ['0x63', [], '0x6f'], ['0x43', [['0x70', '0x69']], '0x54'], ['0
x46', [['0x70', '0x69'], ['0x63', [], '0x6f']], '0x7b'], ['0x70', [['0x70', '0x69'], ['0x63', [], '0x6f'], ['0x43', [['0
x70', '0x69']], '0x54']], '0x79']], '0x6e']], '0x5f'], ['0x77', [['0x70', '0x69'], ['0x63', [], '0x6f'], ['0x43', [['0x7
0', '0x69']], '0x54'], ['0x46', [['0x70', '0x69'], ['0x63', [], '0x6f']], '0x7b'], ['0x70', [['0x70', '0x69'], ['0x63',
[], '0x6f'], ['0x43', [['0x70', '0x69']], '0x54']], '0x79'], ['0x74', [['0x70', '0x69'], ['0x63', [], '0x6f'], ['0x43',
[['0x70', '0x69']], '0x54'], ['0x46', [['0x70', '0x69'], ['0x63', [], '0x6f']], '0x7b']], '0x68'], ['0x6f', [['0x70', '0
x69'], ['0x63', [], '0x6f'], ['0x43', [['0x70', '0x69']], '0x54'], ['0x46', [['0x70', '0x69'], ['0x63', [], '0x6f']], '0
x7b'], ['0x70', [['0x70', '0x69'], ['0x63', [], '0x6f'], ['0x43', [['0x70', '0x69']], '0x54']], '0x79']], '0x6e'], ['0x5
```

Let's check the source code then



```python
import sys

def exit():
  sys.exit(0)

def scramble(L):
  A = L
  i = 2
  while (i < len(A)):
    A[i-2] += A.pop(i-1)
    A[i-1].append(A[:i-2])
    i += 1

  return L

def get_flag():
  flag = open('flag.txt', 'r').read()
  flag = flag.strip()
  hex_flag = []
  for c in flag:
    hex_flag.append([str(hex(ord(c)))])

  return hex_flag
```



```python
def main():
    flag = get_flag()
    cypher = scramble(flag)
    print(cypher)

if __name__ == '__main__':
    main()
```

The main is very simple, it gets the flag, puts it in a scrambler, then prints the scrambled flag. Nothing is interesting on **get_flag()** except for one thing. Aside from reading the flag, it turns the flag into an array containing arrays of each character in hex in string format. As an example it writes out [['0x70'], ['0x69'], ..., ['0x7d']].

Our main focus should be directed to the **scramble(flag)** function, from a glance it seems that the function runs through every single element inside the flag array, let's break this function down.

Let's say we have an array `A = [[0], [1], [2], [3], [4], [5], [6]],`

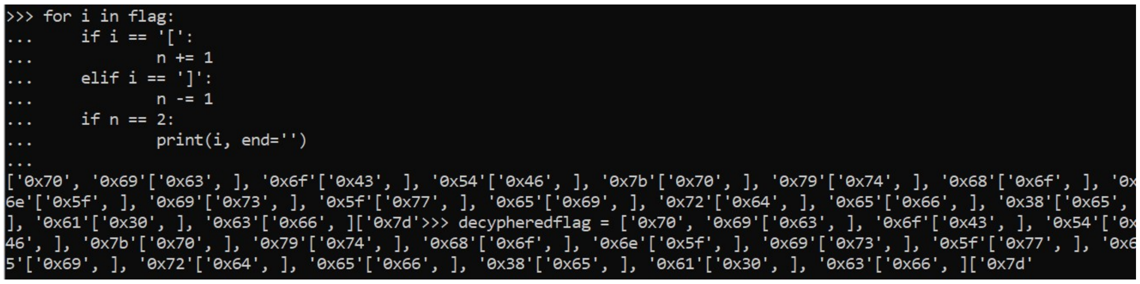
- on the first iteration `i = 2`
    - `A[0] += A.pop(1)` copies A[1], move it inside A[0], then deletes A[1]. The array now looks like this A = `[[0, 1], [2], [3], [4], [5], [6]]`
    - `A[1].append(A[:0])` copies every array until A[0] (exclusive) inside A[1]. The array now looks like this `[[0, 1], [2, []], [3], [4], [5], [6]]`
- on second iteration `i = 3`
    - `A[1] += A.pop(2)` copies A[2], move it inside A[1], then deletes A[2]. The array now looks like this A = `[[0, 1], [2, [], 3], [4], [5], [6]]`
    - `A[2].append(A[:1])` copies every array until A[1] (exclusive) inside A[2]. The array now looks like this `[[0, 1], [2, [], 3], [4, [[0,1]]], [5], [6]]`
- on the third (and last) iteration `i = 4`
    - I'm not going to write step by step but now the array looks like this `[['0x0', '0x1'], ['0x2', [], '0x3'], ['0x4', [['0x0', '0x1']], '0x5'], ['0x6', [['0x0', '0x1'], ['0x2', [], '0x3']]]]`

The length of the original array is 6 but the result seems to be very long right? Now imagine the flag which is much longer. However upon inspection, there seems to be a pattern inside the result, if we highlight the only first instance of an element, it'll look like this

`[['0x0', '0x1'], ['0x2', [], '0x3'], ['0x4', [['0x0', '0x1']], '0x5'], ['0x6', [['0x0', '0x1'], ['0x2', [], '0x3']]]]`

What does it mean? This means that the first instance of the original array element is only located inside the second layer of array. This also means that we can completely ignore the nested arrays inside the second layer array. So how do we extract the flag?

First, copy the long text output inside a txt file, we will be processing this text to get the flag. Then, write a code that will only prints out characters if it's located inside the second layer, if the characters are located in any other layer then it'll be ignored.

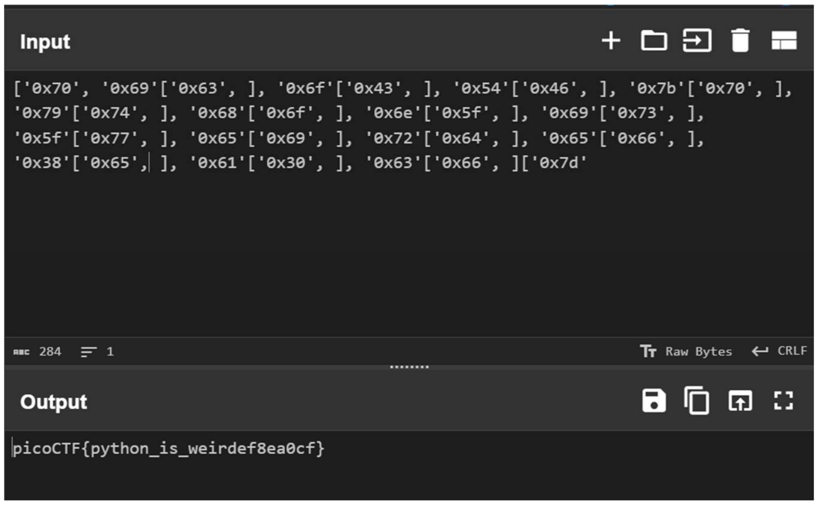


```
>>> for i in flag:
...     if i == '[':
...             n += 1
...     elif i == ']':
...             n -= 1
...     if n == 2:
...             print(i, end='')
...
['0x70', '0x69'['0x63', ], '0x6f'['0x43', ], '0x54'['0x46', ], '0x7b'['0x70', ], '0x79'['0x74', ], '0x68'['0x6f', ], '0x
6e'['0x5f', ], '0x69'['0x73', ], '0x5f'['0x77', ], '0x65'['0x69', ], '0x72'['0x64', ], '0x65'['0x66', ], '0x38'['0x65',
], '0x61'['0x30', ], '0x63'['0x66', ]['0x7d'>>> decypheredflag = ['0x70', '0x69'['0x63', ], '0x6f'['0x43', ], '0x54'['0x
46', ], '0x7b'['0x70', ], '0x79'['0x74', ], '0x68'['0x6f', ], '0x6e'['0x5f', ], '0x69'['0x73', ], '0x5f'['0x77', ], '0x6
5'['0x69', ], '0x72'['0x64', ], '0x65'['0x66', ], '0x38'['0x65', ], '0x61'['0x30', ], '0x63'['0x66', ]['0x7d'
```

I know it's kinda messy but now we have the hex for each characters inside the flag

['0x70', '0x69'['0x63',], '0x6f'['0x43',], '0x54'['0x46',], '0x7b'['0x70',], '0x79'['0x74',], '0x68'['0x6f',], '0x6e'['0x5f',], '0x69'['0x73',], '0x5f'['0x77',], '0x65'['0x69',], '0x72'['0x64',], '0x65'['0x66',], '0x38'['0x65',], '0x61'['0x30',], '0x63'['0x66',]['0x7d'

Put it through hex decoder and voila, we have the flag



```
['0x70', '0x69'['0x63', ], '0x6f'['0x43', ], '0x54'['0x46', ], '0x7b'['0x70', ],
'0x79'['0x74', ], '0x68'['0x6f', ], '0x6e'['0x5f', ], '0x69'['0x73', ],
'0x5f'['0x77', ], '0x65'['0x69', ], '0x72'['0x64', ], '0x65'['0x66', ],
'0x38'['0x65', ], '0x61'['0x30', ], '0x63'['0x66', ]['0x7d'
```

```
picoCTF{python_is_weirdef8ea0cf}
```

Flag: picoCTF{python_is_weirdef8ea0cf}