CGuards

Reverse Engineering Challenge

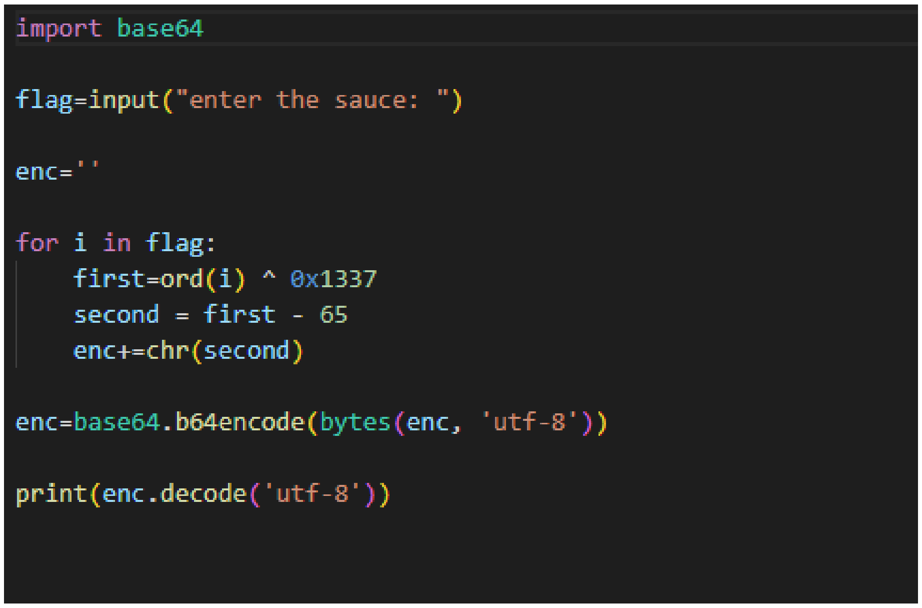
Description

This is a Reverse Engineering challenge written in Python marshal bytecode. The objective of the challenge is to understand the functionality of the marshal bytecode by disassembling it and then creating a code that can reverse the function to ultimately obtain the flag.

About the Challenge

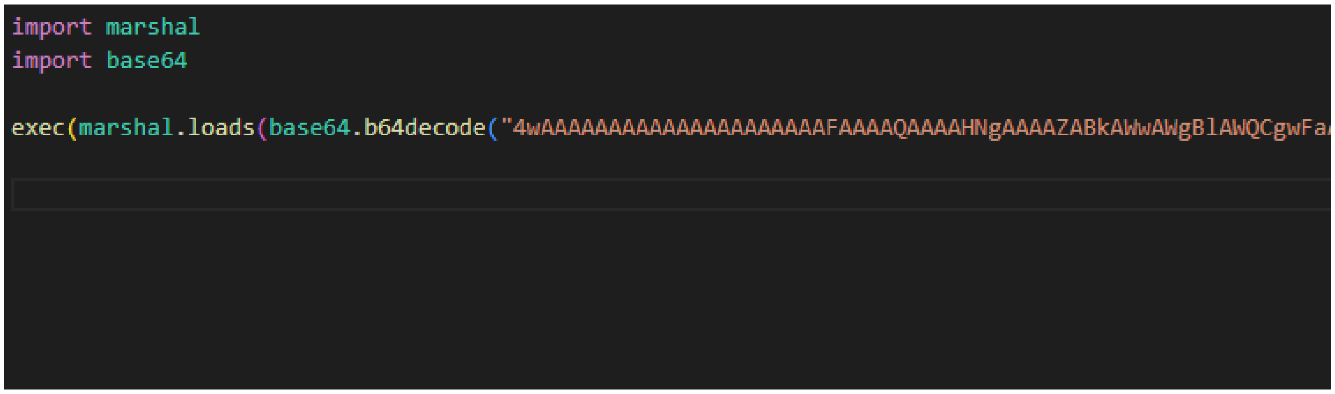
The challengers will be given a Python file with marshal byte code and an encrypted message of the flag.

Main Code:



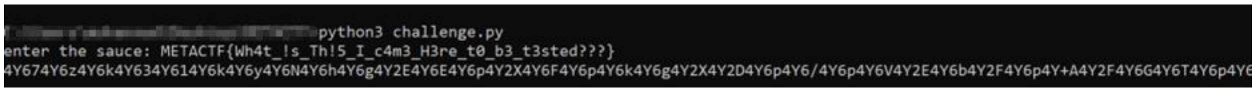
```python
import base64

flag=input("enter the sauce: ")

enc=''

for i in flag:
    first=ord(i) ^ 0x1337
    second = first - 65
    enc+=chr(second)

enc=base64.b64encode(bytes(enc, 'utf-8'))

print(enc.decode('utf-8'))
```

The Marshal Challenge Code:



```
import marshal
import base64

exec(marshal.loads(base64.b64decode("4wAAAAAAAAAAAAAAAAAAAAAFAAAAQAAAAHNgAAAAZABkAWwAWgBlAWQCgwFa
```

After executing the code, it will ask the challengers to enter a text and it will be encrypted as shown by the encryption method in the main code.

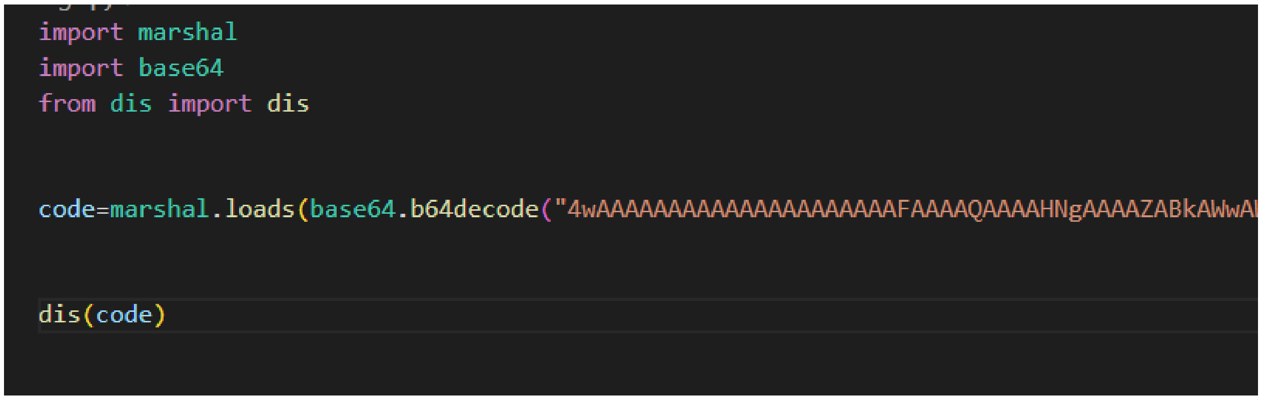


```
[redacted]python3 challenge.py
enter the sauce: METACTF{Wh4t_!s_Th!5_I_c4m3_H3re_t0_b3_t3sted???}
4Y674Y6z4Y6k4Y634Y614Y6k4Y6y4Y6N4Y6h4Y6g4Y2E4Y6E4Y6p4Y2X4Y6F4Y6p4Y6k4Y6g4Y2X4Y2D4Y6p4Y6/4Y6p4Y6V4Y2E4Y6b4Y2F4Y6p4Y+A4Y2F4Y6G4Y6T4Y6p4Y6
```

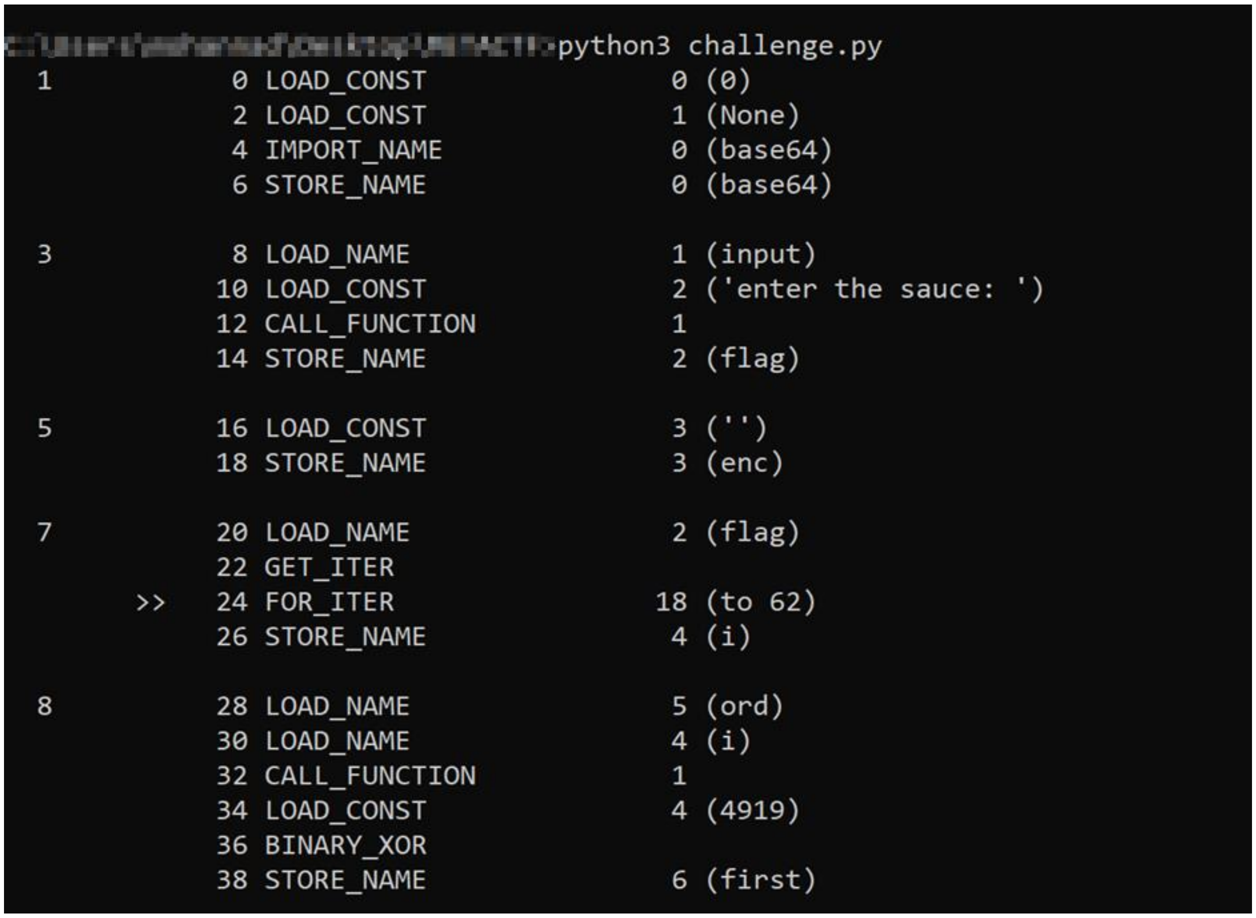
The goal here is to find out how the marshal code works and write a code to reverse the encryption method.

Solution

First, the challengers need to disassemble the marshal bytecode using the Python library **dis.**



```
import marshal
import base64
from dis import dis


code=marshal.loads(base64.b64decode("4wAAAAAAAAAAAAAAAAAAAAFAAAAQAAAAHNgAAAAZABkAWwA


dis(code)
```



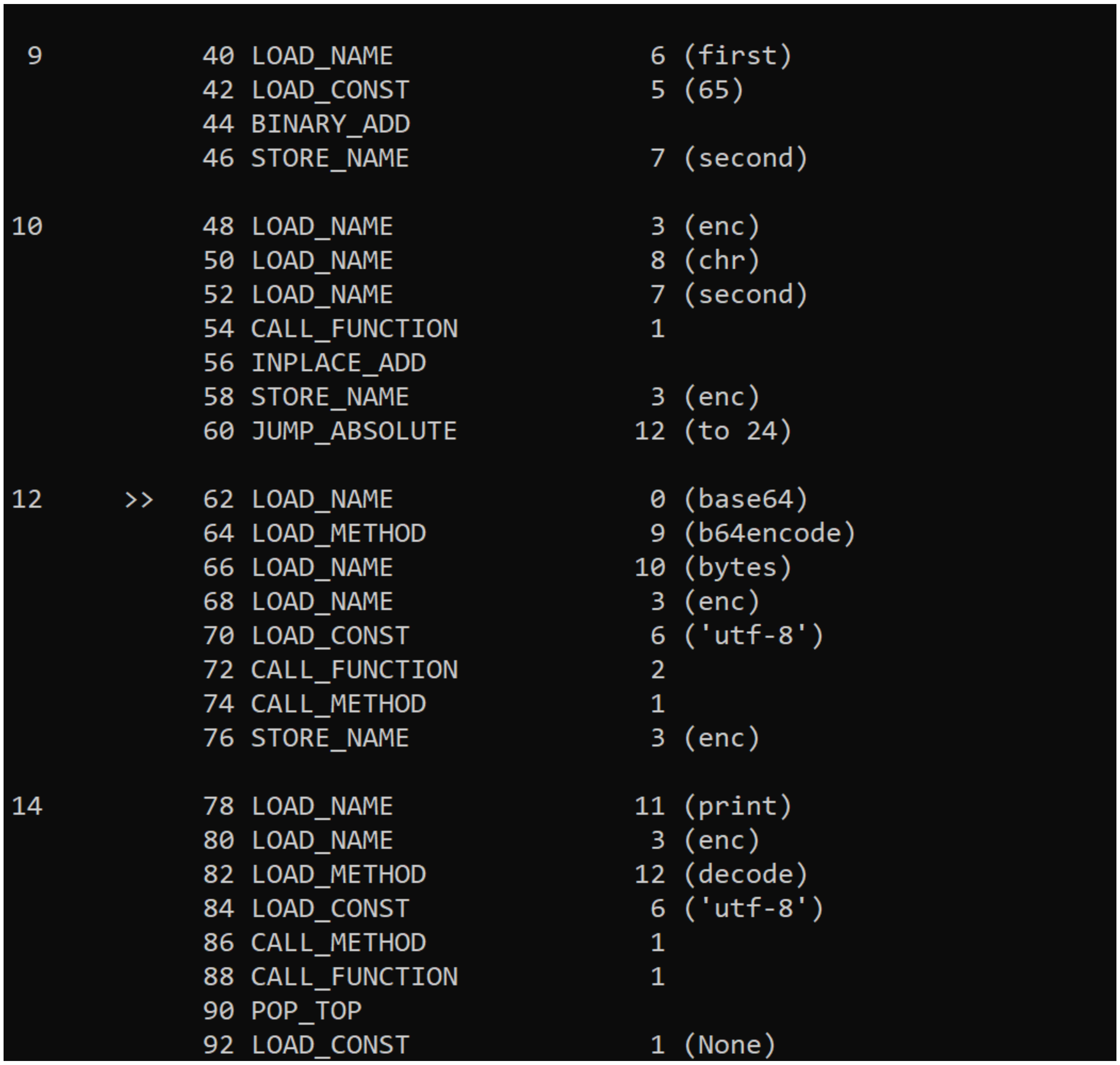
```
[redacted]>python3 challenge.py
  1           0 LOAD_CONST               0 (0)
              2 LOAD_CONST               1 (None)
              4 IMPORT_NAME              0 (base64)
              6 STORE_NAME               0 (base64)

  3           8 LOAD_NAME                1 (input)
             10 LOAD_CONST               2 ('enter the sauce: ')
             12 CALL_FUNCTION            1
             14 STORE_NAME               2 (flag)

  5          16 LOAD_CONST               3 ('')
             18 STORE_NAME               3 (enc)

  7          20 LOAD_NAME                2 (flag)
             22 GET_ITER
        >>   24 FOR_ITER                18 (to 62)
             26 STORE_NAME               4 (i)

  8          28 LOAD_NAME                5 (ord)
             30 LOAD_NAME                4 (i)
             32 CALL_FUNCTION            1
             34 LOAD_CONST               4 (4919)
             36 BINARY_XOR
             38 STORE_NAME               6 (first)
```

From the disassembled Python marshal bytecode, challengers can understand what the code is doing to make it easier to reverse the code.

From the previous image, challengers can see that it's importing the **base64** library and input that has "**enter the Sauce:**" text then it saves the input into the "**flag"** variable.

Then it defines an empty variable that's called "**enc". Then** it does a for loop on the "**flag**" variable for each character in it

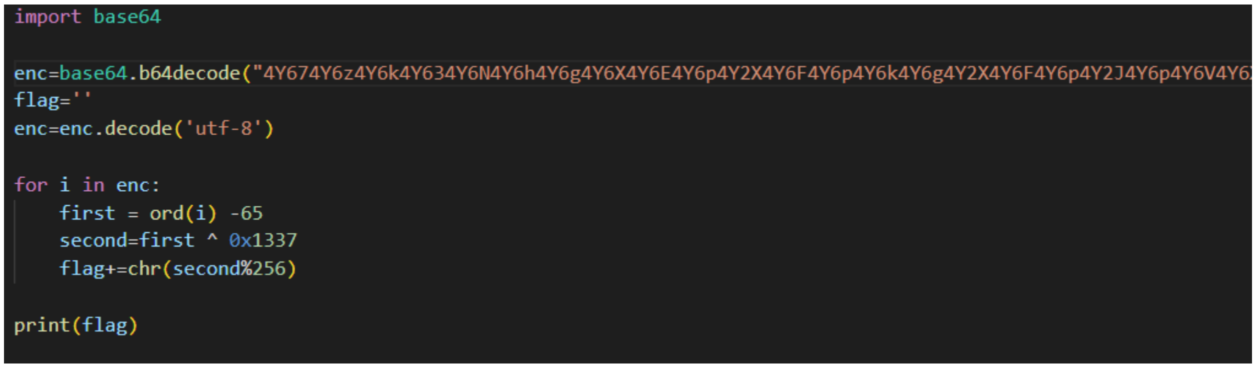
After that, it takes the characters and converts them to a decimal using the "**ord"** function and "**xor**" it with the value of 4919 which is 0x1337 in Hex, and stores it in the variable "**first".**



```
 9           40 LOAD_NAME                 6 (first)
             42 LOAD_CONST                5 (65)
             44 BINARY_ADD
             46 STORE_NAME                7 (second)

10           48 LOAD_NAME                 3 (enc)
             50 LOAD_NAME                 8 (chr)
             52 LOAD_NAME                 7 (second)
             54 CALL_FUNCTION             1
             56 INPLACE_ADD
             58 STORE_NAME                3 (enc)
             60 JUMP_ABSOLUTE            12 (to 24)

12      >>   62 LOAD_NAME                 0 (base64)
             64 LOAD_METHOD               9 (b64encode)
             66 LOAD_NAME                10 (bytes)
             68 LOAD_NAME                 3 (enc)
             70 LOAD_CONST                6 ('utf-8')
             72 CALL_FUNCTION             2
             74 CALL_METHOD               1
             76 STORE_NAME                3 (enc)

14           78 LOAD_NAME                11 (print)
             80 LOAD_NAME                 3 (enc)
             82 LOAD_METHOD              12 (decode)
             84 LOAD_CONST                6 ('utf-8')
             86 CALL_METHOD               1
             88 CALL_FUNCTION             1
             90 POP_TOP
             92 LOAD_CONST                1 (None)
```

After that, it takes what’s in the “**first”** variable and adds 65 to it then store it in a variable “**second”**.

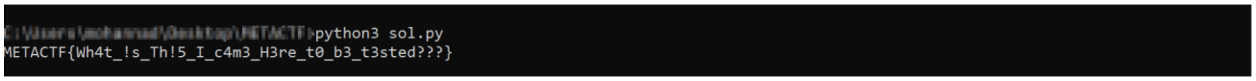
Then it converts the value in the “**second**” variable and converts from decimal to the character and stores it in the “enc” variable, then it decodes the value of the “**enc**” variable to “**UTF-8**” and prints out the encrypted flag.
The challengers need to write a code that reverses the marshal bytecode to get the flag.



```
import base64

enc=base64.b64decode("4Y674Y6z4Y6k4Y634Y6N4Y6h4Y6g4Y6X4Y6E4Y6p4Y2X4Y6F4Y6p4Y6k4Y6g4Y2X4Y6F4Y6p4Y2J4Y6p4Y6V4Y6
flag=''
enc=enc.decode('utf-8')

for i in enc:
    first = ord(i) -65
    second=first ^ 0x1337
    flag+=chr(second%256)

print(flag)
```

The code takes the encrypted text and decodes it from base64, then decodes it again to UTF-8.
After that it’s doing a for loop on the “**enc**” converts the characters to decimal and subtract 65 from the decimal value of the character and stores it to the variable “**first**”, Then it takes what’s in the “**first**” and “**xor**” it 0x1337 or with 4919 in decimal and store it in the variable called “**second**”.

After that, it takes the value of the “second” variable, mod it with 256, converts it from decimal to character, and stores it in the “**flag**” variable. After that, it prints the flag.



```
[redacted]>python3 sol.py
METACTF{Wh4t_!s_Th!5_I_c4m3_H3re_t0_b3_t3sted???}
```