VishwaCTF

CHALLENGE NAME : [BAD COMPRESSION]

DEV : [SAKSHAM SAIPATWAR]

CATEGORY : [REVERSE ENGINEERING]

LEVEL : [HARD]

2024

Description :- *Mr. David was a student pursuing Doctrate of Science in MIT . He was developing a compression algorithm and wrote a code for the same and tested it by encoding a document ... which was sucessfully Encoded . But he mistakenly deleted that document , can you help him retrive this documnet from its encoded file ... you can refer to his diary .*

Step 1 : *Extract the contents of py installer generated executable file . (I've pasted the source code of 'pyinstxtractor' inside the ex.py file)*

```
Microsoft Windows [Version 10.0.22631.3296]
(c) Microsoft Corporation. All rights reserved.

D:\CyberSecurity\Stuff_for_Challenges\test>py ex.py D:\CyberSecurity\Stuff_for_Challenges\test\compress.exe
[+] Processing D:\CyberSecurity\Stuff_for_Challenges\test\compress.exe
[+] Pyinstaller version: 2.1+
[+] Python version: 3.12
[+] Length of package: 7705149 bytes
[+] Found 60 files in CArchive
[+] Beginning extraction...please standby
[+] Possible entry point: pyiboot01_bootstrap.pyc
[+] Possible entry point: pyi_rth_inspect.pyc
[+] Possible entry point: compress.pyc
[+] Found 103 files in PYZ archive
[+] Successfully extracted pyinstaller archive: D:\CyberSecurity\Stuff_for_Challenges\test\compress.exe

You can now use a python decompiler on the pyc files within the extracted directory

D:\CyberSecurity\Stuff_for_Challenges\test>
```

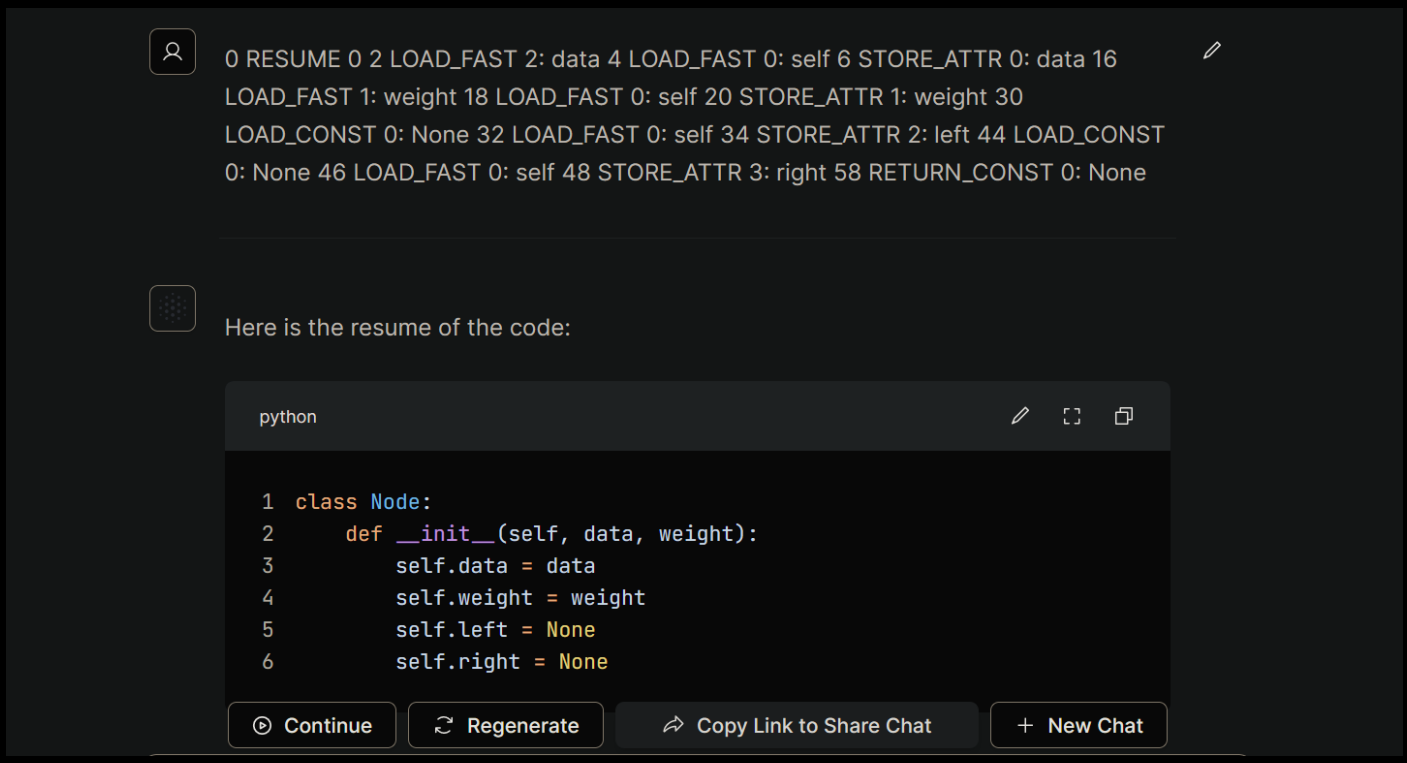
Step 2 : *Decompyle the '.pyc' file using 'pycdas' as 'pycdc' won't be useful with the latest python versions .*

```
Microsoft Windows [Version 10.0.22631.3296]
(c) Microsoft Corporation. All rights reserved.

D:\CyberSecurity\Stuff_for_Challenges\test>pycdas D:\CyberSecurity\Stuff_for_Challenges\test\compress.pyc
compress.pyc (Python 3.12)
[Code]
    File Name: compress.py
    Object Name: <module>
    Qualified Name: <module>
    Arg Count: 0
    Pos Only Arg Count: 0
    KW Only Arg Count: 0
    Stack Size: 4
    Flags: 0x00000000
    [Names]
        'heapq'
        'collections'
        'Node'
        'ltrim'
        'rtrim'
        'trim'
        'find_freq'
        'build_pq'
        'build_tree'
        'get_codes'
        'apply_changes'
        'write_to_file'
        'encode'
        'main'
        '__name__'
```

Step 3 : *Construct the source code for the output using AI. Feed the text partwise don't feed all at once , doing so AI will try to approximate the code to the standard one , won't help you solving the challenge .*

```
                    'weight'
                    'left'
                    'right'
                [Locals+Names]
                    'self'
                    'weight'
                    'data'
                [Constants]
                    None
                [Disassembly]
                    0       RESUME                  0
                    2       LOAD_FAST               2: data
                    4       LOAD_FAST               0: self
                    6       STORE_ATTR              0: data
                    16      LOAD_FAST               1: weight
                    18      LOAD_FAST               0: self
                    20      STORE_ATTR              1: weight
                    30      LOAD_CONST              0: None
                    32      LOAD_FAST               0: self
                    34      STORE_ATTR              2: left
                    44      LOAD_CONST              0: None
                    46      LOAD_FAST               0: self
                    48      STORE_ATTR              3: right
                    58      RETURN_CONST            0: None
            [Code]
                File Name: compress.py
                Object Name: __lt__
                Qualified Name: Node.__lt__
                Arg Count: 2
```



In this way you can generate the entire code , or by now you should have got the idea about the code i.e. 'Huffman compression algo' just check for the differences compared to the standard Huffman code . The below are the differences …

```
[Constants]
    None
    7
[Disassembly]
        0       RESUME                  0
        2       BUILD_LIST              0
        4       STORE_FAST              1: pq
        6       LOAD_FAST               0: freq
        8       LOAD_ATTR               1: heapq
       28       CALL                    0
       36       GET_ITER
       38       FOR_ITER                54 (to 148)
       42       UNPACK_SEQUENCE         2
       46       STORE_FAST              2: char
       48       STORE_FAST              0: freq
       50       LOAD_GLOBAL             3: NULL + heapq
       60       LOAD_ATTR               4: Node
       80       LOAD_FAST               1: pq
       82       LOAD_FAST               0: freq
       84       LOAD_GLOBAL             7: NULL + ord
       94       LOAD_FAST               2: char
       96       CALL                    1
      104       LOAD_CONST              1: 7
      106       BINARY_OP               6 (%)
      110       BINARY_OP               0 (+)
      114       LOAD_GLOBAL             9: NULL + Node
      124       LOAD_FAST               0: freq
      126       LOAD_FAST               2: char
      128       CALL                    2
      136       BUILD_TUPLE             2
```

```
        Pos Only Arg Count: 0
        KW Only Arg Count: 0
        Stack Size: 2
        Flags: 0x00000003 (CO_OPTIMIZED | CO_NEWLOCALS)
        [Names]
            'weight'
        [Locals+Names]
            'self'
            'other'
        [Constants]
            None
        [Disassembly]
            0       RESUME                  0
            2       LOAD_FAST               0: self
            4       LOAD_ATTR               0: weight
            24      LOAD_FAST               1: other
            26      LOAD_ATTR               0: weight
            46      COMPARE_OP              68 (>)
            50      RETURN_VALUE
    None
```

Step 4 : As you have the entire code , its clear the algo encrypts the characters on the basis of its frequency of occurance in the string , which is provided in the diary , feed this freq into the source code to get the codes used per character .

```
def find_freq():
    freq = collections.defaultdict(int)
    11 = [53,65,70,72,75,82,84,86,87,89,97,104,105,115,119,123,125]
    12=[55,85]
    13=[52]
    14=[51,67,68,78]
    15=[95]
    16=[48]

    for i in 11:
        freq[chr(i)] = 1
    for i in 12:
        freq[chr(i)] = 2
    for i in 13:
        freq[chr(i)] = 3
    for i in 14:
        freq[chr(i)] = 4
    for i in 15:
        freq[chr(i)] = 5
    for i in 16:
        freq[chr(i)] = 6
    return freq
```

```
{'_': '000', 'D': '001', 'R': '01000', 'K': '01001', 'Y': '0101', '3': '0110', '4': '0111', 'C': '1000', '7
': '1001', 'U': '1010000', 'A': '1010001', 'H': '101001', 's': '10101', 'N': '1011', 'a': '110000', 'h': '1
10001', '}': '11001', '5': '110100', '{': '110101', 'W': '110110', 'V': '1101110', 'F': '110111100', 'T': '
110111101', 'w': '110111110', 'i': '110111111', '0': '111'}
```

Step 5 :- *Write a script to decrypt the given file and this prefix codes in it .*

```
huffman_codes ={'_': '000', 'D': '001', 'R': '01000', 'K': '01001', 'Y': '0101',
'3': '0110', '4': '0111', 'C': '1000', '7': '1001', 'U': '1010000', 'A': '1010001',
'H': '101001', 's': '10101', 'N': '1011', 'a': '110000', 'h': '110001', '}':
'11001', '5': '110100', '{': '110101', 'W': '110110', 'V': '1101110', 'F':
'110111100', 'T': '110111101', 'w': '110111110', 'i': '110111111', '0': '111'}
def decode_huffman(binary_string, huffman_codes):
    decoded_message = ""
    current_code = ""
    for bit in binary_string:
        current_code += bit
        if current_code in huffman_codes.values():
            decoded_message += next(key for key, value in huffman_codes.items() if
            value == current_code)
            current_code = ""
    return decoded_message

binary_string =
"1101110110111101101011100011101111101100001000110111111110111100110101100011110110
1101000101000110011101000000101111101000000000101101000111001011000100010011010010110
0000101000010110100110111111110110101101111110111000100011100101101100 1"
decoded_message = decode_huffman(binary_string, huffman_codes)
print("Decoded message:", decoded_message)
```

```
Decoded message: VishwaCTF{C0N4RA75_Y0U_D3C0D3D_7H3_UNKN0WN404_C0D3}
```