The biggest issue with this challenge is that the 5th called function in main is broken. It is intended to transform the current encoded input into morse, However it simply changes every character to a space. As a result we need to reverse engineer the first 4 functions. Or, rather, functions 2 3 and 4 as the first function simply converts "_" to "x". I will be covering these in reverse order.

Function 4:

The weirdest part to consider is the use of "-101" and "-37", as visible in the decompiled code. After a bit of research and using different compilers, I realized that we are working with BYTE values, meaning they range from -128 to 127. As a result, when this is broken C simply performs a mod 256 to restore the value to the range. This result in these interactions:

A (65) → -101 - 65 = -166 → Wraps to 90 ('Z')

Z (90) → -101 - 90 = -191 → Wraps to 65 ('A')

a (97) → -37 - 97 = -134 → Wraps to 122 ('z')

z (122) → -37 - 122 = -159 → Wraps to 97 ('a')

From this we conclude function for mirrors the letter (if we imagine we place a mirror after the 13th letter)

```
def f4(text: str) -> str:
    mirrored_text = []
    for char in text:
        if 'A' <= char <= 'Z':
            mirrored_text.append(chr(155 - ord(char)))
        elif 'a' <= char <= 'z':
            mirrored_text.append(chr(219 - ord(char)))
        else:
            mirrored_text.append(char)
    return ''.join(mirrored_text)
```

Function 3:

This is by far the most complex one. After playing around a bit with the code after translating in python, I noticed it mimics a matrix shuffling from AES:

For example, take ABCDEFGHIJKL. A matrix with 5 rows is formed and filled in this way:

AI

BHJ

CGK

DFL

E

Basically, in order of rows where a character is put, it does 1,2,3,4,5,4,3,2,1,2,3,4

Then it concatenates each row.

Reversing this is a bit complex, but the general steps I figured are:

1. Generating the original row assignment sequence based on the length of the encoded text.

2. Counting occurrences of each row index to determine how many characters belong to each row.

3. Splitting the encoded text into 5 regions according to those counts.

4. Rebuilding the matrix with the split text.

5. Traversing the matrix in the original row sequence to reconstruct the initial string.

```
def f3(encoded_text: str) -> str:
    length = len(encoded_text)
    row_sequence = []
    row_counts = [0] * 5
    row = 0
    direction = 1
    for _ in range(length):
        row_sequence.append(row)
        row_counts[row] += 1
        row += direction
        if row == 4 or row == 0:
            direction *= -1
    regions = []
    index = 0
    for count in row_counts:
        regions.append(list(encoded_text[index:index+count]))
        index += count
    original_text = []
    row_indices = [0] * 5
    for row in row_sequence:
        original_text.append(regions[row][row_indices[row]])
        row_indices[row] += 1
    return "".join(original_text)
```

Function 2:

After reading the code, we realize what happens here is the input and another string are converted to uppercase, the the i-th character from 2nd string is added to the i-th character from input string and if the result is not in uppercase range we wrap around to A. Reversing this shouldn't be an issue.

```
def f2(input_string: str) -> str:
    s2 = "ABCDEFGHIJKLMNOPQRSTUVWXYZASDFGASDFEE"
    result = []
    for i in range(len(input_string)):
        new_char = chr(ord(input_string[i]) - ord(s2[i]) + 65)
        if not ('A' <= new_char <= 'Z'):
```

```
            new_char = chr(ord(new_char) + 26)
        result.append(new_char)
    return ''.join(result)
```

After running these 3 function on the decoded morse, we get:

BATMANXWILLXALWAYSXWATCHXGOTHAMXCITY

After replacing the X's we get

BATMAN_WILL_ALWAYS_WATCH_GOTHAM_CITY

However, we need sha256(original message), though I tried a few of the more obvious guesses and in the original message ALL characters were lowercase.

CTF{sha256(tolower(message))}

Made with love by: AndreiCat