After analyzing the source code for the server, we conclude the following:

1) The xor-ing is done AFTER the encryption, not before (like CBC does), meaning it is completely reversible.

Let's say we have IV and 2 blocks of cipher, C1 and C2.

C1= IV XOR C1, C2= C1 XOR C2.

And the end result is IV + C1 + C2.

Meaning, we can recover the ciphertext blocks by applying the same operations again, in reverse order.

2) The encryption is done on prefix + string_we_control + flag.

We can find the length of prefix and therefore canceling by making the string we control just a's, and incrementing the number of a's we have by one until we get 2 identical blocks.

3) string_we_control + flag in ECB mode means we can perform a chosen plaintext attack. It's a classic attack, so I won't go in the details.

Note: I chose to make the exploit on 16 byte blocks, since that's what my AES ECB CPA is made on anyway.

Now, to filter out the 32 block connections (which happen sometimes) I figured upon testing if I send "a" as message, I can either get 112 length responses or 128. From this it's obvious 112 is 16 block size and 128 is 32 block size, as they can't be the other way around.

Step 1: finding the length of the prefix

```python
from pwn import *
import base64

BLOCK_SIZE = 16

def xor(x, y):
    return bytes(a ^ b for a, b in zip(x, y))

def reverse_xor(blocks):
    for i in range(len(blocks)-1, 1, -1):
        blocks[i] = xor(blocks[i], blocks[i-1])
    return blocks

def check_equal_consecutive_blocks(reversed_data, block_size):
    blocks = [reversed_data[i:i + block_size] for i in range(0,
len(reversed_data), block_size)]
    for i in range(1, len(blocks)):
        if blocks[i] == blocks[i - 1]:
            return True, blocks[i], i-1, i
    return False, None, None, None
```

```
server_ip = '35.246.139.54'
server_port = 30118
p = remote(server_ip, server_port)

# Step 1: Check if the block size is 16 or 32
p.recvuntil(b"Msg:")
message = "a"
p.sendline(message)
encrypted_data = p.recvline().strip()
encrypted_bytes = base64.b64decode(encrypted_data)
if len(encrypted_bytes) == 128:
    exit("Block size is 32, quitting")

# Step 2: Find the length of the prefix
while True:
    p.recvuntil(b"Msg:")
    p.sendline(message)
    encrypted_data = p.recvline().strip()
    encrypted_bytes = base64.b64decode(encrypted_data)
    blocks = [encrypted_bytes[i:i + BLOCK_SIZE] for i in range(0,
len(encrypted_bytes), BLOCK_SIZE)]
    reversed_blocks = reverse_xor(blocks)
    reversed_data = b''.join(reversed_blocks)
    equal, matching_block, block1_index, block2_index =
check_equal_consecutive_blocks(reversed_data, BLOCK_SIZE)
    if equal:
        print(f"Found two equal consecutive blocks with {len(message)} 'a's:")
        print(f"Matching Block: {matching_block}")
        print(f"Block {block1_index} and Block {block2_index} are equal.")
        break
    message += "a"
```

After running this code we see that we use 38 a's to get 2 equal blocks, blocks 3 and 4 (index 0). Quick math: 38-32=6 a's in block two, prefix is 16-6=10 bytes long.

So, to summarize: the first block is the IV, the 2nd block is the prefix (10 chars) + 6 a's (6 chars) to neutralize the effect of the prefix on the CPA.

This means we need to adapt our CPA attack so we send 6 a's extra no matter what and start from the 3rd block. This is to neutralize the effect of the prefix.

As a result, the **while True:** will become a function, slightly modified to our needs. This function will act as the encryption for the CPA.

All things considered, here is the final solve script:

```python
from pwn import *
import base64
BLOCK_SIZE = 16

def xor(x, y):
    return bytes(a ^ b for a, b in zip(x, y))

def reverse_xor(blocks):
    for i in range(len(blocks)-1, 1, -1):
        blocks[i] = xor(blocks[i], blocks[i-1])
    return blocks

def send_message(message):
    p.recvuntil(b"Msg:")
    neutralize="aaaaaa"
    p.sendline(neutralize+message)
    encrypted_data = p.recvline().strip()
    encrypted_bytes = base64.b64decode(encrypted_data)
    blocks = [encrypted_bytes[i:i + BLOCK_SIZE] for i in range(0,
len(encrypted_bytes), BLOCK_SIZE)]
    reversed_blocks = reverse_xor(blocks)
    reversed_blocks = reversed_blocks[2:]
    reversed_data = b''.join(reversed_blocks)
    return reversed_data

def brute_force_plaintext_blocks(ciphertext):
    charset = "0123456789abcdefCTF{}"
    decrypted_text = ""
    num_blocks = len(ciphertext) // BLOCK_SIZE - 2
    for block_index in range(1, num_blocks+1):
        block_plaintext = ""
        for i in range(1, BLOCK_SIZE + 1):
            prefix = "a" * (BLOCK_SIZE - i)
            encrypted_candidate = send_message(prefix)[:BLOCK_SIZE * block_index]
            for char in charset:
                candidate = prefix + decrypted_text + block_plaintext + char
                if send_message(candidate)[:BLOCK_SIZE * block_index] ==
encrypted_candidate:
                    block_plaintext += char
                    break
            else:
                break

        decrypted_text += block_plaintext
    return decrypted_text
```

```python
server_ip = '34.159.151.77'
server_port = 30720
context.log_level='error'
p = remote(server_ip, server_port)
p.recvuntil(b"Msg:")
message = "a"
p.sendline(message)
encrypted_data = p.recvline().strip()
encrypted_bytes = base64.b64decode(encrypted_data)
if len(encrypted_bytes) == 128:
    exit("Block size is 32, quitting")
print(brute_force_plaintext_blocks(encrypted_bytes))
```

After a short wait we get the flag. Yey!

Made with love by: AndreiCat