```python
#!/usr/bin/env python3

import json
import socket
import random
from string import ascii_letters

MESSAGES = [
    "Pad to the left",
    "Unpad it back now y'all",
    "Game hop this time",
    "Real world, let's stomp!",
    "Random world, let's stomp!",
    "AES real smooth~"
]

# Pad all the messages to 32 bytes so that they are all at the same length
MESSAGES = [
    msg.ljust(32) for msg in MESSAGES
]

# this function is same as xor() but it can take bytes or hex type and returns hex
def modified_xor(x, y):
    if type(x) != bytes:
        x = bytes.fromhex(x)
    if type(y) != bytes:
        y = bytes.fromhex(y)

    return bytes(a ^ b for a, b in zip(x, y)).hex()


# ================================================================================
#   Config Variables (Change as needed)
# ================================================================================

# Remember to change the port if you are reusing this client for other challenges
PORT = 50403

# Change this to REMOTE = False if you are running against a local instance of the server
REMOTE = True

# Remember to change this to graded.aclabs.ethz.ch if you use this for graded labs
HOST = "aclabs.ethz.ch"

# ================================================================================
#   Client Boilerplate (Do not touch, do not look)
# ================================================================================

fd = socket.create_connection((HOST if REMOTE else "localhost", PORT)).makefile("rw")

def run_command(command):
    """Serialize `command` to JSON and send to the server, then deserialize the response"""
    fd.write(json.dumps(command) + "\n")
    fd.flush()
    return json.loads(fd.readline())

# ================================================================================
#    Write Your Solution Below
# ================================================================================

# this is my own implementation of PKCS7 padding as described in the previous labs
def pkcs_pad(msg, k:int):
    bmsg = msg
    if type(msg) != bytes:
        bmsg = msg.encode('utf-8')

    lth = len(bmsg)
    p = lth % k
    padding = bytes([k-p]*(k-p))
    return bytes(bmsg+padding)


predicted_ivs = {}

# we create a dicitonary with all the possible seeds so that we can easily map an iv to a seed
for seed in MESSAGES:
    rng = random.Random(seed)
    predicted_ivs[seed] = rng.randbytes(16).hex()

inverted_dict = {value: key for key, value in predicted_ivs.items()} # this dictionary contains the first iv generated by random
# with as seed a message from the set of messages we are given. This is done so that we can then replicate and predict the IVs generated by the server


first_half = ""
second_half = ""

r0 = run_command({"command": "encrypt", "msg": ""})
server_chosen_seed = inverted_dict[r0['iv']]  # here we use the dictionary above to know which seed has been used by the server


rng = random.Random(server_chosen_seed) # we then pass the seed to a random generator locally on our side
rng.randbytes(16) # since the server has ran randbytes(16) once for our first encrypt command, we "burn" one randbytes() so that we catch up with the server

# in this for loop we first extract the first half of the secret byte by byte. Note here that we are going to miss one byte from the first half since if we send 16 it would not
# place the secret in the  first block since it would be complete. So we have to start already with 1 byte from the secret
for counter in range(15,0,-1):
    sent = run_command({"command": "encrypt", "msg": (counter*b'0').hex()}) # here we first send 15 bytes which would place 15 zeros + one byte of the secret since the block is 16 byt
    ciphertext_1 = sent['ctxt'][:32] # we save ciphertext we want to compare to. So right now we care about the first block since it contains bytes of the secret
    old_iv = sent['iv']
    rng.randbytes(16) # here we "burn" another randbyte since it was used on the server by runing the above encryp command. We need to keep up

    # we try for every letter to find a matching ciphertext
    for i in range(len(ascii_letters)):
        next_iv_predict = rng.randbytes(16) # this is what we predict the server will use next as iv since it will also run randbytes() with same seed as ours
        crafted_message = modified_xor(counter*b'0' + first_half.encode('utf-8') + ascii_letters[i].encode('utf-8'),modified_xor(old_iv,next_iv_predict)) # we send (message xor predic
        # will cancel the predictd iv and use the old iv to encrypt the message
        r1 = run_command({"command": "encrypt", "msg": crafted_message})

        if r1['ctxt'][:32] == ciphertext_1:
            first_half += ascii_letters[i] # we add the symbol if we find a match
            break



# this is for the second half of the secret. Here we can get the entire half cause we are "pushing" the secret into an empty block one by one.
for counter in range(1,17):

    sent = run_command({"command": "encrypt", "msg": (counter*b'0').hex()})   # here we reveal last symbol first, then second to last, and so on. By sending 1 byte we "push" the last
    # byte of the secret onto a new block. So we have secret byte + PKCS7 padding to fill 16 bytes block.
    ciphertext_1 = sent['ctxt'][64:96] # here we save the last block cipher since that is in our interest now
```

```
    old_iv = sent['ctxt'][32:64] # the old iv is not the ciphertext block before the last one. (CBC)

    rng.randbytes(16) # we again "burn" one randombyte run

    # bruteforcing cause why not
    for i in range(len(ascii_letters)):
        next_iv_predict = rng.randbytes(16)
        crafted_message = modified_xor(pkcs_pad(ascii_letters[i]+second_half,16).hex(),modified_xor(old_iv,next_iv_predict)) # this is same as for the first half except that here we n
        # in order to match with that the server will be encrypting
        rl = run_command({"command": "encrypt", "msg": crafted_message}) # we send our crafter message

        if rl['ctxt'][:32] == ciphertext_1:
            second_half = ascii_letters[i] + second_half # we get the seconf half but since it is in reverse we add each new symbol "on top" of the exising ones
            break


# here we are running one last bruteforce to find a middle missing byte
sent = run_command({"command": "encrypt", "msg": ""})   # encrypt entire message so we send no additional bytes
# byte of the secret onto a new block. So we have secret byte + PKCS7 padding to fill 16 bytes block.
ciphertext_1 = sent['ctxt'][:32] # we only care about the first block
old_iv = sent['iv']  # the used iv
rng.randbytes(16) # we again "burn" one randombyte run

# bruteforcing
for i in range(len(ascii_letters)):
    next_iv_predict = rng.randbytes(16)
    # here we add an ascii character to the end of the first half and we see if the ciphertext matches
    crafted_message = modified_xor((first_half + ascii_letters[i]).encode('utf-8').hex(),modified_xor(old_iv,next_iv_predict))

    rl = run_command({"command": "encrypt", "msg": crafted_message}) # we send our crafter message

    if rl['ctxt'][:32] == ciphertext_1: # check if match
        first_half += ascii_letters[i]
        break

secret = first_half+second_half # entire secret *mic drop*

sol = run_command({"command": "guess", "guess": secret})

print(sol['flag'])
```