AES部分

基础部分

Bringing It All Together

- 刚开始把前面的都整合在一起发现跑不动，最后发现一个小注意点就是要稍微修改一下sub_bytes函数的内容，在之前写的时候我是直接return字符内容，而整合起来的decrypt函数中是用数字作代换，所以将sub_bytes函数作了变动

```
N_ROUNDS = 10

key        = b'\xc3,\\\xa6\xb5\x80^\x0c\xdb\x8d\xa5z*\xb6\xfe\\'
ciphertext = b'\xd1O\x14j\xa4+O\xb6\xa1\xc4\x08B)\x8f\x12\xdd'

s_box = (
    0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5, 0x30, 0x01, 0x67, 0x2B, 0xFE,
0xD7, 0xAB, 0x76,
    0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0, 0xAD, 0xD4, 0xA2, 0xAF, 0x9C,
0xA4, 0x72, 0xC0,
    0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC, 0x34, 0xA5, 0xE5, 0xF1, 0x71,
0xD8, 0x31, 0x15,
    0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A, 0x07, 0x12, 0x80, 0xE2, 0xEB,
0x27, 0xB2, 0x75,
    0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0, 0x52, 0x3B, 0xD6, 0xB3, 0x29,
0xE3, 0x2F, 0x84,
    0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B, 0x6A, 0xCB, 0xBE, 0x39, 0x4A,
0x4C, 0x58, 0xCF,
    0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85, 0x45, 0xF9, 0x02, 0x7F, 0x50,
0x3C, 0x9F, 0xA8,
    0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5, 0xBC, 0xB6, 0xDA, 0x21, 0x10,
0xFF, 0xF3, 0xD2,
    0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17, 0xC4, 0xA7, 0x7E, 0x3D, 0x64,
0x5D, 0x19, 0x73,
    0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88, 0x46, 0xEE, 0xB8, 0x14, 0xDE,
0x5E, 0x0B, 0xDB,
    0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C, 0xC2, 0xD3, 0xAC, 0x62, 0x91,
0x95, 0xE4, 0x79,
    0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9, 0x6C, 0x56, 0xF4, 0xEA, 0x65,
0x7A, 0xAE, 0x08,
    0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6, 0xE8, 0xDD, 0x74, 0x1F, 0x4B,
0xBD, 0x8B, 0x8A,
    0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E, 0x61, 0x35, 0x57, 0xB9, 0x86,
0xC1, 0x1D, 0x9E,
    0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94, 0x9B, 0x1E, 0x87, 0xE9, 0xCE,
0x55, 0x28, 0xDF,
    0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68, 0x41, 0x99, 0x2D, 0x0F, 0xB0,
0x54, 0xBB, 0x16,
)
```

```
inv_s_box = (
    0x52, 0x09, 0x6A, 0xD5, 0x30, 0x36, 0xA5, 0x38, 0xBF, 0x40, 0xA3, 0x9E, 0x81,
0xF3, 0xD7, 0xFB,
    0x7C, 0xE3, 0x39, 0x82, 0x9B, 0x2F, 0xFF, 0x87, 0x34, 0x8E, 0x43, 0x44, 0xC4,
0xDE, 0xE9, 0xCB,
    0x54, 0x7B, 0x94, 0x32, 0xA6, 0xC2, 0x23, 0x3D, 0xEE, 0x4C, 0x95, 0x0B, 0x42,
0xFA, 0xC3, 0x4E,
    0x08, 0x2E, 0xA1, 0x66, 0x28, 0xD9, 0x24, 0xB2, 0x76, 0x5B, 0xA2, 0x49, 0x6D,
0x8B, 0xD1, 0x25,
    0x72, 0xF8, 0xF6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xD4, 0xA4, 0x5C, 0xCC, 0x5D,
0x65, 0xB6, 0x92,
    0x6C, 0x70, 0x48, 0x50, 0xFD, 0xED, 0xB9, 0xDA, 0x5E, 0x15, 0x46, 0x57, 0xA7,
0x8D, 0x9D, 0x84,
    0x90, 0xD8, 0xAB, 0x00, 0x8C, 0xBC, 0xD3, 0x0A, 0xF7, 0xE4, 0x58, 0x05, 0xB8,
0xB3, 0x45, 0x06,
    0xD0, 0x2C, 0x1E, 0x8F, 0xCA, 0x3F, 0x0F, 0x02, 0xC1, 0xAF, 0xBD, 0x03, 0x01,
0x13, 0x8A, 0x6B,
    0x3A, 0x91, 0x11, 0x41, 0x4F, 0x67, 0xDC, 0xEA, 0x97, 0xF2, 0xCF, 0xCE, 0xF0,
0xB4, 0xE6, 0x73,
    0x96, 0xAC, 0x74, 0x22, 0xE7, 0xAD, 0x35, 0x85, 0xE2, 0xF9, 0x37, 0xE8, 0x1C,
0x75, 0xDF, 0x6E,
    0x47, 0xF1, 0x1A, 0x71, 0x1D, 0x29, 0xC5, 0x89, 0x6F, 0xB7, 0x62, 0x0E, 0xAA,
0x18, 0xBE, 0x1B,
    0xFC, 0x56, 0x3E, 0x4B, 0xC6, 0xD2, 0x79, 0x20, 0x9A, 0xDB, 0xC0, 0xFE, 0x78,
0xCD, 0x5A, 0xF4,
    0x1F, 0xDD, 0xA8, 0x33, 0x88, 0x07, 0xC7, 0x31, 0xB1, 0x12, 0x10, 0x59, 0x27,
0x80, 0xEC, 0x5F,
    0x60, 0x51, 0x7F, 0xA9, 0x19, 0xB5, 0x4A, 0x0D, 0x2D, 0xE5, 0x7A, 0x9F, 0x93,
0xC9, 0x9C, 0xEF,
    0xA0, 0xE0, 0x3B, 0x4D, 0xAE, 0x2A, 0xF5, 0xB0, 0xC8, 0xEB, 0xBB, 0x3C, 0x83,
0x53, 0x99, 0x61,
    0x17, 0x2B, 0x04, 0x7E, 0xBA, 0x77, 0xD6, 0x26, 0xE1, 0x69, 0x14, 0x63, 0x55,
0x21, 0x0C, 0x7D,
)

def add_round_key(s, k):
    for i in range(4):
        for j in range(4):
            s[i][j] ^= k[i][j]
    return s

def bytes2matrix(text):
    """ Converts a 16-byte array into a 4x4 matrix.  """
    return [list(text[i:i+4]) for i in range(0, len(text), 4)]

def matrix2bytes(matrix):
    """ Converts a 4x4 matrix into a 16-byte array.  """
    return bytes([i for sublist in matrix for i in sublist])

def inv_shift_rows(s):
    s[1][1], s[2][1], s[3][1], s[0][1] = s[0][1], s[1][1], s[2][1], s[3][1]
    s[2][2], s[3][2], s[0][2], s[1][2] = s[0][2], s[1][2], s[2][2], s[3][2]
```

```
    s[3][3], s[0][3], s[1][3], s[2][3] = s[0][3], s[1][3], s[2][3], s[3][3]

xtime = lambda a: (((a << 1) ^ 0x1B) & 0xFF) if (a & 0x80) else (a << 1)

def mix_single_column(a):
    # see Sec 4.1.2 in The Design of Rijndael
    t = a[0] ^ a[1] ^ a[2] ^ a[3]
    u = a[0]
    a[0] ^= t ^ xtime(a[0] ^ a[1])
    a[1] ^= t ^ xtime(a[1] ^ a[2])
    a[2] ^= t ^ xtime(a[2] ^ a[3])
    a[3] ^= t ^ xtime(a[3] ^ u)

def mix_columns(s):
    for i in range(4):
        mix_single_column(s[i])

def inv_mix_columns(s):
    # see Sec 4.1.3 in The Design of Rijndael
    for i in range(4):
        u = xtime(xtime(s[i][0] ^ s[i][2]))
        v = xtime(xtime(s[i][1] ^ s[i][3]))
        s[i][0] ^= u
        s[i][1] ^= v
        s[i][2] ^= u
        s[i][3] ^= v
    mix_columns(s)

def sub_bytes(s, sbox=s_box):
    return [[int(sbox[a]) for a in row] for row in s]

def expand_key(master_key):
    """
    Expands and returns a list of key matrices for the given master_key.
    """

    # Round constants https://en.wikipedia.org/wiki/AES_key_schedule#Round_constants
    r_con = (
        0x00, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40,
        0x80, 0x1B, 0x36, 0x6C, 0xD8, 0xAB, 0x4D, 0x9A,
        0x2F, 0x5E, 0xBC, 0x63, 0xC6, 0x97, 0x35, 0x6A,
        0xD4, 0xB3, 0x7D, 0xFA, 0xEF, 0xC5, 0x91, 0x39,
    )

    # Initialize round keys with raw key material.
    key_columns = bytes2matrix(master_key)
    iteration_size = len(master_key) // 4

    # Each iteration has exactly as many columns as the key material.
    i = 1
    while len(key_columns) < (N_ROUNDS + 1) * 4:
        # Copy previous word.
```

```
        word = list(key_columns[-1])

        # Perform schedule_core once every "row".
        if len(key_columns) % iteration_size == 0:
            # Circular shift.
            word.append(word.pop(0))
            # Map to S-BOX.
            word = [s_box[b] for b in word]
            # XOR with first byte of R-CON, since the others bytes of R-CON are 0.
            word[0] ^= r_con[i]
            i += 1
        elif len(master_key) == 32 and len(key_columns) % iteration_size == 4:
            # Run word through S-box in the fourth iteration when using a
            # 256-bit key.
            word = [s_box[b] for b in word]

        # XOR with equivalent word from previous iteration.
        word = bytes(i^j for i, j in zip(word, key_columns[-iteration_size]))
        key_columns.append(word)

    # Group key words in 4x4 byte matrices.
    return [key_columns[4*i : 4*(i+1)] for i in range(len(key_columns) // 4)]


def decrypt(key, ciphertext):
    round_keys = expand_key(key) # Remember to start from the last round key and
work backwards through them when decrypting

    # Convert ciphertext to state matrix
    state = bytes2matrix(ciphertext)
    # Initial add round key step
    state = add_round_key(state,round_keys[-1])
    for i in range(N_ROUNDS - 1, 0, -1):
        inv_shift_rows(state)
        state = sub_bytes(state,inv_s_box)
        state = add_round_key(state,round_keys[i])
        inv_mix_columns(state)

    # Run final round (skips the InvMixColumns step)
    inv_shift_rows(state)
    state = sub_bytes(state,inv_s_box)
    state = add_round_key(state,round_keys[0])
    # Convert state matrix to plaintext
    plaintext = matrix2bytes(state)
    return plaintext


print(decrypt(key, ciphertext))
```

得到flag：crypto{MYAES128}

ECB Oracle

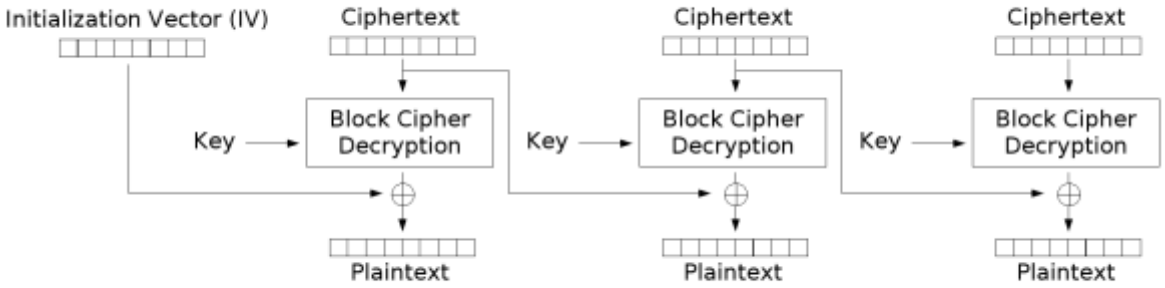
首先理解源码中`pad`方法的功能，就是把输入的字符填充为16的整数倍，我们可以利用这一特性推断出flag的长度（即当输入字符增加一个长度时，输出的块也增加一块）。具体到该题，当输入字符长度从6变为7时，ciphertext增加一块，由32bytes变成48bytes，因此可以推断出flag长度为32-7=25bytes

推算出flag长度之后，我们又已经知道flag的格式为crypto{}，又已知每个块16字节，当我们发送15个字节，且这段明文必须放在标志之前，所以我们可以知道下一个字节必须是flag的第一个字节，又ECB加密模式下每个块是独立的，且拥有自己的加密文本，因此如果第16个字节与flag的第一个字节相同，就可以得到和第一块相同的加密文本。以此类推就可以bruteforce出flag的具体内容。具体代码如下：

```
import string
import requests

def encrypt(plaintext):
    r = requests.get(f"https://aes.cryptohack.org/ecb_oracle/encrypt/{plaintext}")
    return bytes.fromhex(r.json()['ciphertext'])

ciphers=[]
for i in range(1,17):
    garbage = i*b'?'.hex()
    ct = encrypt(garbage)
    ciphers.append(ct)

alpha = list(dict.fromkeys("crypto{eainshr_}" + string.ascii_lowercase +
string.digits + string.ascii_uppercase))
flag = b""
for i in range(31,7,-1):
    for char in alpha:
        char =char.encode()
        ct = encrypt((i*b"?"+flag+char).hex())[:32]
        exp = encrypt((i*b"?").hex())[:32]
        if ct == exp:
            flag += char
            print(f"{char.decode()}",flush=True,end='')
            break
```

> flag:crypto{p3n6u1n5_h473_3cb}

CBC Byte Flip



Cipher Block Chaining (CBC) mode decryption

CBC 字节反转攻击

原理

观察解密过程可以发现如下特性：

- IV向量影响第一个明文分组
- 第n个密文分组可以影响第n+1个明文分组

假设第n个密文分组为C_n，解密后的第n个明文分组为P_n，$P_{n+1} = C_n xor f(C_{n+1})$

对于某个信息已知的原文和密文，我们可以修改第n个密文块C_n为$C_n xor P_{n+1} xor A$，然后再对这条密文进行解密，那么解密后的第n个明文块就会变成A

非对称密码

RSA攻击的学习

选择明文攻击

假设有一个加密oracle,但是不知道n和e，那么我们可以通过加密oracle获取n，且在e比较小时，可以利用Pollard's kangaroo算法获取e

我们可以加密 2，4，8，16。那么我们可以知道

$c_2 = 2^e \bmod n$

$c_4 = 4^e \bmod n$

$c_8 = 8^e \bmod n$

那么

$c_2^2 \equiv c_4 \bmod n$

$c_2^3 \equiv c_8 \bmod n$

故而

$c_2^2 - c_4 = kn$

$c_2^3 - c_8 = tn$

可以求出kn和tn的最大公因数，得到的很大概率就是n

任意密文解密

假设Alice创建密文$C = P^e mod n$,并且把C发送给Bob,同时假设我们要对爱丽丝加密后的任意密文解密，而不只是解密C，那么可以通过拦截C，通过以下步骤求出P：

- 选择任意的$X \in Z_n$，且X与N互素
- 计算$Y = C \times X^e mod n$
- 由于可以进行选择密文攻击，那么求得的Y对应的解密结果就是$Z = Y^d$
- 由于$Z = Y^d = (C \times X^e)^d = C^d X = P^{ed} X = PX mod n$,由于X与N互素，很容易求得相应的逆元，进而可以得到P

格密码部分

格的含义

- 定义在非空有限集合上的偏序集合L，满足集合L中的任意元素a,b,使得a,b在L中存在一个最大下界和最小上界。
- 格是 m 维欧式空间 R^m 的 n 个线性无关向量$b_i (1 \leq i \leq n)$ 的所有整系数的线性组合，即 $L(B) = \sum x_i b_i : x_i \in Z, 1 \leq i \leq n$，称这n个向量为格L的一组基，格的秩为n，格的位数为m
- 一些基本定义与计算困难性问题
  - 连续最小长度
  - 最短向量问题（SVP)
  - 近似最短向量问题
  - 连续最小长度问题
  - 最短线性无关向量问题
  - 唯一最短向量问题
  - 最近向量问题

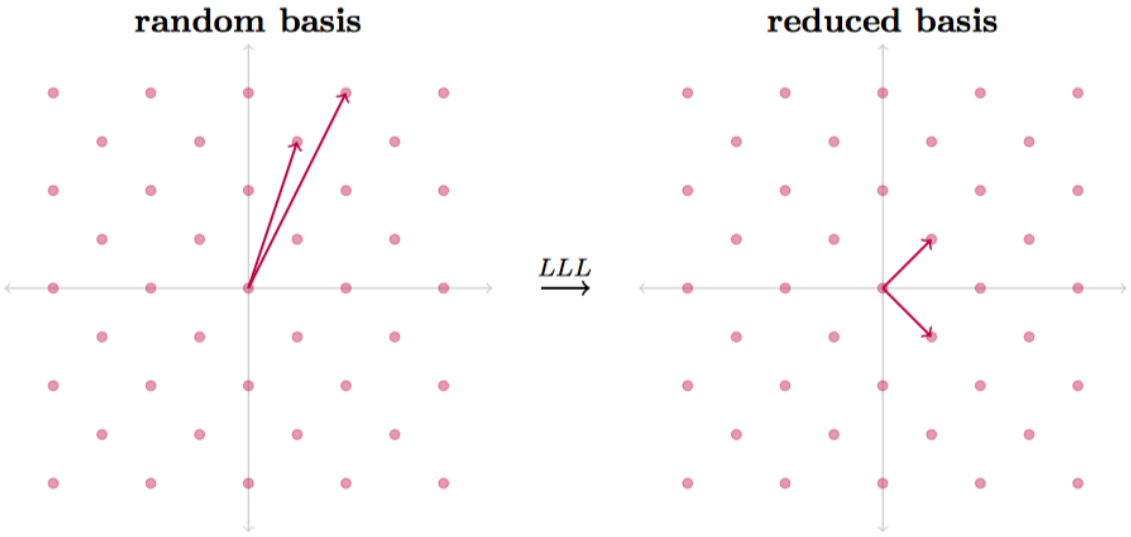
格基规约

Definition 1. *Let L be a lattice with a basis B. The δ-LLL algorithm applied on L's basis B produces a new basis of L : $B' = \{b_1, \ldots, b_n\}$ satisfying :*

$$\forall\, 1 \leq j < i \leq n \text{ we have } |\mu_{i,j}| \leq \frac{1}{2} \tag{1}$$

$$\forall\, 1 \leq i < n \text{ we have } \delta \cdot \|\tilde{b_i}\|^2 \leq \|\mu_{i+1,i} \cdot \tilde{b}_i + \tilde{b}_{i+1}\|^2 \tag{2}$$

$$\text{with } \mu_{i,j} = \frac{b_i \cdot \tilde{b}_j}{\tilde{b}_j \cdot \tilde{b}_j} \text{ and } \tilde{b}_1 = b_1 \text{ (Gram-Schmidt)}$$



通过此方法生成的基具有如下性质

Property 1. *Let L be a lattice of dimension n. In polynomial time, the LLL algorithm outputs reduced basis vectors v_i, for $1 \leq i \leq n$, satisfying :*

$$\|v_1\| \leq \|v_2\| \leq \ldots \leq \|v_i\| \leq 2^{\frac{n(n-1)}{4(n+1-i)}} \cdot det(L)^{\frac{1}{n+1-i}}$$