LILCTF2025 WriteUp

by zijeff

Warm Up

我们来逐行分析题目：

1. **密钥是固定的**：

   ```
   key = Random(2025).randbytes(16)
   ```

   这里使用了 random 库（梅森旋转算法），并且指定了种子 2025。这意味着无论何时运行这段代码，生成的 key 都是完全一样的。我们可以直接在解密脚本中生成同样的 Key。

2. **CBC模式的IV依赖于明文**：

   ```
   AES.new(key, AES.MODE_CBC, iv=FLAG[9:25])
   ```

   使用了 AES 的 CBC 模式。

   - **Key**: 已知。
   - **IV (初始化向量)**: 取自 FLAG 的第 9 到 25 个字节 (FLAG[9:25])。
   - **Plaintext**: 填充后的 FLAG。

而 **AES-CBC** 模式的解密过程如下：

$$P_i = D_k(C_i) \oplus C_{i-1}$$

其中，C_i 是密文块，P_i 是明文块，D_k 是解密函数。特殊地，C_{-1} 就是初始化向量。

假设 Flag 长度经过填充后被分为 $P_0, P_1, P_2 \ldots$ 几个块（每块16字节），密文分为 $C_0, C_1, C_2 \ldots$。

对于除了第一块以外的所有块 ($i > 0$)：

$$P_i = Dec_{key}(C_i) \oplus C_{i-1}$$

因为 Key 是已知的，且密文 C 是已知的，所以我们可以**直接解密出除了第一块以外的所有明文**。这意味着我们知道了 **FLAG[16:]** 的所有内容。

对于第一块 (P_0)：

$$P_0 = Dec_{key}(C_0) \oplus IV$$

这里 P_0 是 **FLAG[0:16]**，IV 是 **FLAG[9:25]**。令中间状态 $X_0 = Dec_{key}(C_0)$（这是我们能算出来的），那么则有：

$$FLAG[0:16] = X_0 \oplus FLAG[9:25]$$

如果我们按字节来看这个异或关系（设 f_i 为 Flag 的第 i 个字节）：

$$f_i = X_0[i] \oplus f_{i+9} \quad (\text{对于 } 0 \le i \le 15)$$

这就是解题的关键：

1. 我们通过解密 $C_1, C_2 \ldots$ 已经知道了 f_{16} 及其之后的所有字节。
2. 利用公式 $f_i = X_0[i] \oplus f_{i+9}$：
    - 当 $i = 7$ 时：$f_7 = X_0[7] \oplus f_{16}$。因为 f_{16} 已知，我们可以算出 f_7。
    - 以此类推，我们可以算出 f_7 到 f_{15}（利用已知的 f_{16} 到 f_{24}）。
3. 一旦我们知道了 f_9 到 f_{15}，我们再次利用公式：
    - 当 $i = 0$ 时：$f_0 = X_0[0] \oplus f_9$。因为 f_9 刚才算出来了，所以 f_0 可求。
    - 以此类推，算出 f_0 到 f_6。

所以，解题代码如下：

```
from random import Random
from Crypto.Cipher import AES
from Crypto.Util.Padding import unpad

def solve():
    ciphertext_hex =
"ae39cfab1ba8d38fc3761216c393caf16e3c3f13fe57e2dedd52f1b13072fa93df405c7e
731a193cabe5fd88ee3241f79aded62d139faba8c767a3b8efc5a855"

    ct = bytes.fromhex(ciphertext_hex)

    key = Random(2025).randbytes(16)
    print(f"[+] Recovered Key: {key.hex()}")

    # 我们使用 ECB 模式来手动处理每个块的解密和异或操作
    aes_ecb = AES.new(key, AES.MODE_ECB)

    plaintext_suffix = b""
    for i in range(16, len(ct), 16):
        curr_blk = ct[i:i+16]
        prev_blk = ct[i-16:i]

        dec_out = aes_ecb.decrypt(curr_blk)
        pt_blk = bytes(a ^ b for a, b in zip(dec_out, prev_blk))
        plaintext_suffix += pt_blk

    print(f"[+] Recovered suffix (block 1+): {plaintext_suffix}")

    # 我们构建一个列表来存放 flag，目前已知的是 suffix 部分
    # 前16个字节初始化为0
    flag_bytes = list(b'\x00' * 16) + list(plaintext_suffix)
```

```python
    # 计算第一块密文经过 AES 解密后的中间状态 X0
    # P0 = X0 ^ IV  =>  flag[0:16] = X0 ^ flag[9:25]
    x0 = aes_ecb.decrypt(ct[0:16])

    for i in range(7, 16):
        flag_bytes[i] = x0[i] ^ flag_bytes[i+9]

    for i in range(0, 7):
        flag_bytes[i] = x0[i] ^ flag_bytes[i+9]

    full_flag_padded = bytes(flag_bytes)
    try:
        flag = unpad(full_flag_padded, 16)
        print(f"[SUCCESS] FLAG: {flag.decode()}")
    except Exception as e:
        print(f"[!] Padding error or logic error: {e}")
        print(f"Raw recovered bytes: {full_flag_padded}")

if __name__ == "__main__":
    solve()
```

运行后得到结果：

```
smith@jeff-ciallo:~/Tasks/LILCTF2025$ /bin/python3 /home/smith/Tasks/LILCTF2025/warm_up/exp.py
[+] Recovered Key: bcd8c88e81f82b15821756a5a38768d6
[+] Recovered suffix (block 1+): b'@#$%_H4v3_fUn_W1tH_y0Ur_sYmM3try_3NcRyPt10n!}\x03\x03\x03'
[SUCCESS] FLAG: LILCTF{b1a6labl4@#$%_H4v3_fUn_W1tH_y0Ur_sYmM3try_3NcRyPt10n!}
smith@jeff-ciallo:~/Tasks/LILCTF2025$
```

ez_math

我们已知 A 向量，为了后续方便运算，我们写成列向量的形式：

$$A = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$$

我们根据 B 的生成方式，可以将其进行分解成两个矩阵相乘的形式：

$$B = \begin{bmatrix} v_1\lambda_1 \\ v_2\lambda_2 \end{bmatrix} = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} A$$

然后再观察生成的 C，可以发现貌似找到了思路：

$$C = A^{-1}B = A^{-1} \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} A$$

我们设

$$D = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix}$$

上述的分析说明，C 和 D 矩阵相似，由相似矩阵的特性可知其特征值一样。那么直接求 C 的特征值即可，再转换为 ASCII 字符便能获取到 flag。

具体代码如下：

```
from sage.all import *
from Crypto.Util.number import *

p =
962015477708887069426652167016898650800331486622231579012655250430484623669618373326682848940486027632615819190690739623423694721546629541863205611382616 1
C = matrix(GF(p),
[[706291047823278313876598317062668798120293718425540828760797178013948261652521527021667588732196579841882903827323269537021050308649122843485653862069964 5,709626890595646264332013766778033476364963565773249949110817162216420866268860929560768462063030103178913281420978494822280293008903028748401533675778780 1],

[734143005360617232960291140590575438672922466942532541912473384706069485348382539620084160912557492352553553218446715074638582644339203908607956290505980 8,255724429885608755550053849954229852680037768196690750251858072416536362017096846305015260208366599123014366951986682858767105931862754215336787959626087 2]])
[lambda1, lambda2] = C.eigenvalues()
print(lambda1, lambda2)
print(b'LILCTF{'+long_to_bytes(int(lambda1)) +
long_to_bytes(int(lambda2))+b'}')
```

运行后得到结果：

```
461081882199191304136043558055592717274072444511548267131743 310431440615324582056084165589022
b'LILCTF{It_w4s_the_be5t_of_times_1t_wa5_the_w0rst_of_t1me5}'
```

mid_math

本题和上边题目还是比较相似的，只是多了一步求解离散对数问题。
根据分析，可以发现矩阵 C 相似于一个对角矩阵：

$$\begin{bmatrix} a & 0 & 0 & 0 & 0 \\ 0 & b & 0 & 0 & 0 \\ 0 & 0 & c & 0 & 0 \\ 0 & 0 & 0 & d & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

其特征值为 $a, b, c, d, 0$ 那么由 $D = C^{key}$，则 D 对应的特征值为：

$$a^{key}, b^{key}, c^{key}, d^{key}, 0 \mod p$$

在求出特征值后，我们便可以通过求解**离散对数问题**获取 key，之后便可以获取flag

具体代码如下：

```
from Crypto.Util.number import *
from Crypto.Cipher import AES
from Crypto.Util.Padding import unpad, pad

p = 14668080038311483271
C_list = [[11315841881544731102, 2283439871732792326,
6800685968958241983, 6426158106328779372, 9681186993951502212],
          [4729583429936371197, 9934441408437898498,
12454838789798706101, 1137624354220162514, 8961427323294527914],
          [12212265161975165517, 8264257544674837561,
10531819068765930248, 4088354401871232602, 14653951889442072670],
          [6045978019175462652, 11202714988272207073,
13562937263226951112, 6648446245634067896, 13902820281072641413],
          [1046075193917103481, 3617988773170202613, 3590111338369894405,
2646640112163975771, 59668646987501347077]]
D_list = [[1785348659555163021, 3612773974290420260, 8587341808081935796,
4393730037042586815, 10490463205723658044],
          [10457678631610076741, 16455271956876481407,
13013316081830726847, 12925223531522879912, 5478687620744215372],
          [9878636900393157276, 13274969755872629366,
3231582918568068174, 7045188483430589163, 5126509884591016427],
          [4914941908205759200, 7480989013464904670, 5860406622199128154,
8016615177615097542, 13266674393818320551],
          [3005316032591310201, 6624508725257625760, 7972954954270186094,
5331046349070112118, 61270264943042723957]]
msg =
b"\xcc]B:\xe8\xbc\x91\xe2\x93\xaa\x88\x17\xc4\xe5\x97\x87@\x0fd\xb5p\x81\
x1e\x98,Z\xe1n`\xaf\xe0%:\xb7\x8aD\x03\xd2Wu5\xcd\xc4#m'\xa7\xa4\x80\x0b\
xf7\xda8\x1b\x82k#\xc1gP\xbd/\xb5j"

Fp = GF(p)
C_mat = matrix(Fp, C_list)
D_mat = matrix(Fp, D_list)

#计算 C 和 D 的特征值
eval_C = C_mat.eigenvalues()
eval_D = D_mat.eigenvalues()

#去掉0特征值
eval_C_nonzero = [x for x in eval_C if x != 0]
eval_D_nonzero = [x for x in eval_D if x != 0]

#输出特征值
print("Nonzero eigenvalues of C:", eval_C_nonzero)
```

```
print("Nonzero eigenvalues of D:", eval_D_nonzero)

#尝试计算离散对数以找到key
key = None
for mu in eval_D_nonzero:
    try:
        #尝试计算第一个非零特征值对应的k值
        k = discrete_log(mu, eval_C_nonzero[0])
        #检查k是否满足条件
        valid = True
        for lam in eval_C_nonzero:
            if lam^k not in eval_D_nonzero:
                valid = False
                break
        if valid:
            key = k
            print(f"Found key: {key}")
            break
    except Exception as e:
        print(f"Error: {e}")
        continue

if key is None:
    raise ValueError("Key not found")

key_bytes = pad(long_to_bytes(key), 16)
aes = AES.new(key_bytes, AES.MODE_ECB)
flag = unpad(aes.decrypt(msg), 64)
print("Flag:", flag)
```

运行后得到结果：

```
Nonzero eigenvalues of C: [13548047239731931439, 10741008122066331899, 2915915082365181132, 25243
Nonzero eigenvalues of D: [14219969811373602463, 7805278355513795080, 7126986745593039829, 632194
Found key: 5273966641785501202
Flag: b'LILCTF{Are_y0u_5till_4wake_que5t1on_m4ker!}'
```

Space Travel

代码的核心部分在于生成 `key` 和泄露信息（Gift）：

1. **Key 的生成**:
    - `key` 是由 50 个来自 `vecs` 列表的二进制字符串拼接而成。
    - 索引生成方式为 `urandom(2) & 0xfff`，这意味着索引范围是 0 到 4095 ($2^{12}-1$)。
2. **泄露信息 (Linear Leakage)**:
    - 题目输出了 600 组 `[nonce, bit]`。

- 计算方式：`bit = (bin(nonce & key).count("1")) % 2`。
- 数学意义：这是 `nonce` 和 `key` 两个向量在 GF(2) 有限域上的**内积** (Inner Product)。
- 设 K 为密钥的二进制向量，N_i 为第 i 个 nonce，b_i 为结果位，则有：

$$N_i \cdot K \equiv b_i \pmod 2$$

我们有 600 个方程。如果 Key 是完全随机的 800 位整数，则有 800 个变量。但这并不代表题目不可解。虽然每一个 bit 都在变，但因为 `vecs` 只有 4096 (2^{12}) 个元素，这些 16 位的向量在数学上必然构成一个 **维度不超过 12 的线性子空间（或仿射子空间）**。

1. **降维原理**：
   虽然每个 Block 是 16 bits，但它们是由 12 bits 的索引生成的。如果 `vecs` 是线性生成的，我们可以找到 **12 个基向量 (Basis Vectors)** 来表示所有的 `vecs` 元素。

$$Vec = C + k_0 \cdot B_0 + k_1 \cdot B_1 + \ldots + k_{11} \cdot B_{11}$$

   - C: 偏移向量 (Offset/Base vector)，通常取 `vecs[0]`。
   - B_i: 基向量。
   - k_i: 系数 (0 或 1)，这就是我们要解的**新变量**。
2. **变量统计**：
   - 新变量：50 个 Block × 12 个系数 = 600 个变量。
   - 方程数：600 个。
   - **600 = 600，此时满秩可解**

具体解题代码如下：

```
import ast
from hashlib import md5
from Crypto.Cipher import AES
import os

# ================= 1. 分析 Vecs 的线性结构 =================
print("[*] Analyzing the linear structure of vecs...")

# 将 vecs (字符串列表) 转换为 GF(2) 矩阵
# vecs 元素如 '0111...'
vecs_matrix = Matrix(GF(2), [[int(c) for c in s] for s in vecs])

# 取第一个向量作为基准偏移量 (Base Vector)
base_vec = vecs_matrix[0]

# 计算差分矩阵: {v - v0}
diff_vecs = [row - base_vec for row in vecs_matrix]
diff_matrix = Matrix(GF(2), diff_vecs)
```

```
# 计算线性基 (Basis)
basis = diff_matrix.row_space()
basis_vectors = basis.basis()

DIM = len(basis_vectors)
BLOCK_LEN = vecs_matrix.ncols() # 应该是 16
NUM_BLOCKS = 50

print(f"[+] Block Length: {BLOCK_LEN}")
print(f"[+] Subspace Dimension: {DIM}") # 预期是 12

# ================= 2. 构建线性方程组 =================
print("[*] Building linear system...")

# 读取 output.txt
with open("output.txt", "r") as f:
    content = f.read()
    if "🎁 :" in content:
        list_str = content.split("🎁 :")[1].split("\n")[0].strip()
    else:
        list_str = content.strip()
    data = ast.literal_eval(list_str)

M = []
B = []

for nonce_int, target_bit in data:
    row = []

    # rhs_val 用于存储方程右边的值 (target - 常数项)
    rhs_val = int(target_bit)

    # 遍历 50 个 block
    for block_idx in range(NUM_BLOCKS):
        # 提取当前 block 对应的 Nonce 片段 (16 bits)
        # 移位逻辑：Block 0 是最高位
        shift_amount = (NUM_BLOCKS - 1 - block_idx) * BLOCK_LEN
        nonce_chunk = (nonce_int >> shift_amount) & ((1 << BLOCK_LEN) -
1)

        # 将 nonce_chunk 转为 Sage 向量
        # 注意位序：字符串 index 0 是最高位
        nonce_vec = vector(GF(2), [(nonce_chunk >> (BLOCK_LEN - 1 - i)) &
1 for i in range(BLOCK_LEN)])

        # 1. 计算常数部分的贡献 (Offset Contribution)
        # dot_product 结果是 GF(2) 元素，必须转为 int
        const_contribution = int(nonce_vec * base_vec)
```

```
            # 更新 RHS: 减去常数项 (在 GF(2) 中减法等于加法)
            rhs_val = (rhs_val + const_contribution) % 2

            # 2. 计算变量系数 (Coefficients of Basis)
            for basis_vec in basis_vectors:
                # 计算 nonce 对该基向量的投影
                coeff = int(nonce_vec * basis_vec)
                row.append(coeff)

        M.append(row)
        B.append(rhs_val)

# ================= 3. 求解 =================
print("[*] Solving with SageMath...")
MS = Matrix(GF(2), M)
VS = vector(GF(2), B)

try:
    # solve_right 求解 M * x = B
    coeffs = MS.solve_right(VS)
    print("[+] Solution found!")

    # ================= 4. 重组 Key =================
    print("[*] Reconstructing Key...")

    full_key_bits = ""

    # 遍历 50 个 Block，利用解出的系数重建每个 Block
    for block_idx in range(NUM_BLOCKS):
        # 取出当前 Block 对应的 DIM (12) 个系数
        block_coeffs = coeffs[block_idx*DIM : (block_idx+1)*DIM]

        # 重建向量: V = Base + Sum(c_i * B_i)
        recovered_vec = base_vec
        for i in range(DIM):
            if block_coeffs[i] == 1:
                recovered_vec += basis_vectors[i]

        # 将向量转回 '01' 字符串
        vec_str = "".join(str(x) for x in recovered_vec)
        full_key_bits += vec_str

    key_int = int(full_key_bits, 2)
    print(f"[+] Recovered Key: {key_int}")

    # ================= 5. 解密 Flag =================
    aes_key = md5(str(key_int).encode()).digest()
```

```
    ciphertext = b'r\x9f\xa5N\x19\xa3\x8b-
\xfen\xa7\xc9\xca\x87\x04\x873#ju\xc5\x06\xd0\x85L\x1ab\xa1XEre\xe5\x8c:h
'
    cipher = AES.new(key=aes_key, nonce=b"Tiffany", mode=AES.MODE_CTR)
    flag = cipher.decrypt(ciphertext)
    print(f"🚩 FLAG: {flag}")

except ValueError:
    print("[-] No solution found. System might be inconsistent.")
except Exception as e:
    print(f"[-] An error occurred during reconstruction/decryption: {e}")
```

运行后得到结果：

```
[*] Analyzing the linear structure of vecs...
[+] Block Length: 16
[+] Subspace Dimension: 12
[*] Building linear system...
[*] Solving with SageMath...
[+] Solution found!
[*] Reconstructing Key...
[+] Recovered Key: 52695276121194075877863536151577959854179081336516126047668203438081260176
🚩 FLAG: b'LILCTF{Un1qUe_s0luti0n_1N_sUbSp4C3!}'
```

Linear

构造格求解，具体的构造写在脚本注释中了。好吧，其实是我不会（），Gemini王朝了。

求解代码如下：

```
from sage.all import *
from pwn import *
import ast

HOST,PORT = "challenge.imxbt.cn" , 31501
io = remote(HOST, PORT)

# 接收 A (格式: [[...]])
io.recvuntil(b"[[")
raw_A_str = "[[" + io.recvuntil(b"]]").decode()
A_list = ast.literal_eval(raw_A_str)
# 接收 b (格式: [...])
io.recvuntil(b"[")
raw_b_str = "[" + io.recvuntil(b"]").decode()
b_list = ast.literal_eval(raw_b_str)

def solve_sis():

    # 定义矩阵和向量
    # 这里使用 ZZ (整数环)
    A = Matrix(ZZ, A_list)
```

```
b = vector(ZZ, b_list)

nrows = A.nrows()   # 16
ncols = A.ncols()   # 32

print(f"[*] 正在初始化格矩阵 (Rows: {nrows}, Cols: {ncols})...")

# ------------------------------------------------------
# 构造格 (Lattice Construction) - Embedding Method
# ------------------------------------------------------
# 我们构造一个矩阵 M，形式如下：
# [ I   |  K * A^T ]
# [ 0   | -K * b^T ]
#
# 这里的 K 是一个巨大的权重。
# 如果我们有一个向量 v = (x_1, ..., x_32, 1)
# v * M = (x, K(Ax - b))
# 如果 Ax = b，那么结果向量的右半部分将全部为 0。
# LLL 算法会寻找短向量，因为 x 很小而 K 很大，算法会优先强制右半部分为 0。
# ------------------------------------------------------

# 权重 K：必须足够大，使得满足方程约束比减小 x 的数值更重要。
# x 的最大值约 10^5，K 取 10^10 以上比较安全。
K = 2**60

# 构造矩阵 M
# 维度：(32个变量 + 1个常数项) 行， (32列存x + 16列存约束) 列
M = Matrix(ZZ, ncols + 1, ncols + nrows)

# 1. 左上角：单位矩阵 I (用于保持 x 的值)
M.set_block(0, 0, Matrix.identity(ncols))

# 2. 右上角：K * A^T (方程系数)
M.set_block(0, ncols, K * A.transpose())

# 3. 最后一行：常数项处理
# 左下角是 0 (默认就是0，不用设)
# 右下角是 -K * b (方程的常数项移项)
for i in range(nrows):
    M[ncols, ncols + i] = -K * b[i]

print("[*] 正在运行 LLL 算法 (这可能需要几秒钟)...")

# 运行 LLL 规约
L_reduced = M.LLL()

print("[*] LLL 完成，正在搜索解向量...")

solution = None
```

```
    # 遍历规约后基底的每一行
    for row in L_reduced:
        # 提取向量的右半部分 (应该全为 0)
        check_part = row[ncols:]

        # 提取向量的左半部分 (可能是解 x)
        x_part = row[:ncols]

        # 检查1：约束部分是否全为 0
        if not check_part.is_zero():
            continue

        # 检查2：x 部分不能全为 0
        if x_part.is_zero():
            continue

        # 修正符号：LLL 可能会返回 -x
        # 题目中 x 是正整数，所以如果我们发现大部分是负数，就取反
        # 简单的启发式：看第一个非零元素是否为负
        if x_part[0] < 0:
            x_part = -x_part

        # 检查3：代回验证 Ax = b
        # 将 vector 转换为 list 方便计算
        x_candidate = list(x_part)

        # 验证
        recalc_b = [sum(A_list[i][j] * x_candidate[j] for j in
range(ncols)) for i in range(nrows)]

        if recalc_b == b_list:
            print("[+] 成功找到解！")
            solution = x_candidate
            break

    if solution:
        x = " ".join(map(str, solution))
        print(x)
        io.sendlineafter(b"Enter your solution: ", x.encode())
        print(io.recvall().decode())
        io.close()
    else:
        print("[-] 未找到解。可能原因：")
        print("2. 权重 K 不够大 (尝试增大 K)")

# 执行
solve_sis()
```

运行后得到结果：

```
[+] 成功找到解!
35 19074 81893 58041 76514 61485 8973 5519 10007 11369 2339 31799 2029 68360 15536 10160 25423 12
[x] Receiving all data
[x] Receiving all data: 0B
[x] Receiving all data: 71B
[+] Receiving all data: Done (71B)
[*] Closed connection to challenge.imxbt.cn port 31501
Bravo! Here is your flag:
LILCTF{a23bf0c6-b7d7-40d9-8cd0-54769555829f}
```

baaaaag

可以发现是非常原始的背包加密系统，直接造格打 LLL 会发现出不了结果，但是使用更强的 BKZ，然后 block_size 设置的稍微大一点就能出。

具体代码如下：

```python
from Crypto.Cipher import AES
from Crypto.Util.Padding import unpad
import hashlib

# ========================================
# 题目数据
# ========================================
```

```
a = [96503203064581947322688027 9, 69968039176889166559855637 3,
10221777542147449012476775 27, 68076771457439559544852929 7,
10511445904428308301606561 47, 11686606887363022197983801 51,
79638734985655429244399504 9, 74057984980918893972302493 7,
94077212136244058297697807 1, 78743875275475188522960774 7,
10577103717631435227692620 19, 79217018432468183371098777 1,
91284439267929738675438658 1, 90678750637311520850622183 1,
10733560679722267348033317 11, 12302488919206894782364288 03,
71342684847951300577449733 1, 97952724725653823911643505 1,
97949676556679854682826543 7, 83693951544224330025249947 9,
11852819990506464511675832 69, 67349019882721371756851917 9,
77637820143550560531634851 7, 80992077335220023642245166 7,
10324506925354715342827507 57, 11163460004005452159137540 39,
11477888462835527690491238 03, 99443946404950306551700939 3,
82564532376726226500625753 7, 10767427217244132646363182 41,
73178201865914290417901678 3, 65616288935475835337169913 1,
10455204142634987040195525 71, 12137149723951705837819769 83,
94995072999919857608078100 1, 11500329935791347500994655 19,
97599266297091938867280077 3, 11291486997961429438318430 99,
89887179814153756862410693 9, 99771831450525047078751328 1,
63154345208923289050792561 9, 83133589917337092927963394 3,
11867487655211759503117479 1, 88425219490391268086507130 1,
10160204179167612819867174 67, 89620558291720184760965614 7,
95944042363273888410708630 7, 99336810053669052099561280 7,
70260227799384988754650485 1, 11028074386056494027490344 81,
62953942733308163869153808 9, 88766325868033859419614738 7,
10019658832591526846614934 09, 10438116834839624801621336 33,
93871375938318690481977133 9, 10236996412683105993715686 53,
78402582285896075770394530 9, 98618263451270758797104773 1,
10647394257414115257214371 19, 12094280510669080712902869 53,
66751067384333396364175117 7, 64282891954276033985127355 1,
10866285373093682882043425 99, 10848489449605066636682988 59,
66782729520037363103877595 9, 75263413734831278376172350 7,
70799429779574476136888894 9, 74799898263068858982828436 3,
71018479117533390929159318 9, 65118393015472571680794670 9,
72483660722340007434386807 9, 11189935380915902997216478 99]
b = 34962396275078207988771864327
ciphertext =
b'Lo~G\xf46>\xd609\x8e\x8e\xf5\xf83\xb5\xf0\x8f\x9f6&\xea\x02\xfa\xb1_L\x
85\x93\x93\xf7,`|\xc6\xbe\x05&\x85\x8bC\xcd\xe6?TV4q'

def solve():
    n = len(a)
    print(f"[*] Problem dimension: n = {n}")

    # 使用极其巨大的权重 K，彻底消除部分和的局部极小值干扰
    K = 2**200

    # 构造精确的 (n+1) x (n+1) 矩阵
    M = Matrix(ZZ, n + 1, n + 1)
```

```
for i in range(n):
    M[i, i] = 2           # 对角线为 2
    M[i, n] = K * a[i]    # 最后一列乘以巨大权重 K

for i in range(n):
    M[n, i] = 1           # 最后一行前 n 列为 1
M[n, n] = K * b           # 右下角为 K * b

print("[*] Reconstructed highly weighted CJLOSS lattice.")

# 检查矩阵并提取 p 的函数
def check_matrix(B_mat):
    for row in B_mat:
        # 严格检查最后一列是否为0（由于权重K极大，这里为0意味着100%满足背包和）
        if row[n] != 0:
            continue

        vec = row[:n]
        # 检查前n项是否全为 1 或 -1
        if all(abs(v) == 1 for v in vec):
            print(f"[+] Found short vector (1/-1): {vec[:10]}...")

            # 格基规约可能找到目标向量，也可能找到它的相反数 (-vec)
            # 因此我们生成两种情况进行测试
            x_candidates = [
                [(v + 1) // 2 for v in vec],
                [(-v + 1) // 2 for v in vec]
            ]

            for x in x_candidates:
                calc_sum = sum(x[i] * a[i] for i in range(n))
                if calc_sum == b:
                    print(f"[+] Subset sum verified successfully!")

                    # 还原 p
                    p = 0
                    for i in range(n):
                        p += int(x[i]) * (1 << i)

                    print(f"[*] Recovered p: {p}")

                    # 解密
                    key = hashlib.sha256(str(p).encode()).digest()
                    cipher = AES.new(key, AES.MODE_ECB)
                    try:
                        pt = cipher.decrypt(ciphertext)
                        flag = unpad(pt, 16).decode()
                        print(f"[SUCCESS] FLAG: {flag}")
                        return True
```

```
                        except Exception as e:
                            print(f"[-] Padding error during decryption:
{e}")
        return False

    block_size  = 30
    B_bkz = M.BKZ(block_size=block_size)
    if check_matrix(B_bkz):
        return

    print("\n[-] All lattice reduction techniques failed. Please check
the inputs.")

if __name__ == "__main__":
    solve()
```

运行后得到结果：

```
[*] Problem dimension: n = 72
[*] Reconstructed highly weighted CJLOSS lattice.
[+] Found short vector (1/-1): (-1, -1, 1, -1, -1, 1, -1, 1, -1, 1)...
[+] Subset sum verified successfully!
[*] Recovered p: 420862665310382568 5156
[SUCCESS] FLAG: LILCTF{M4ybe_7he_brut3_f0rce_1s_be5t}
```