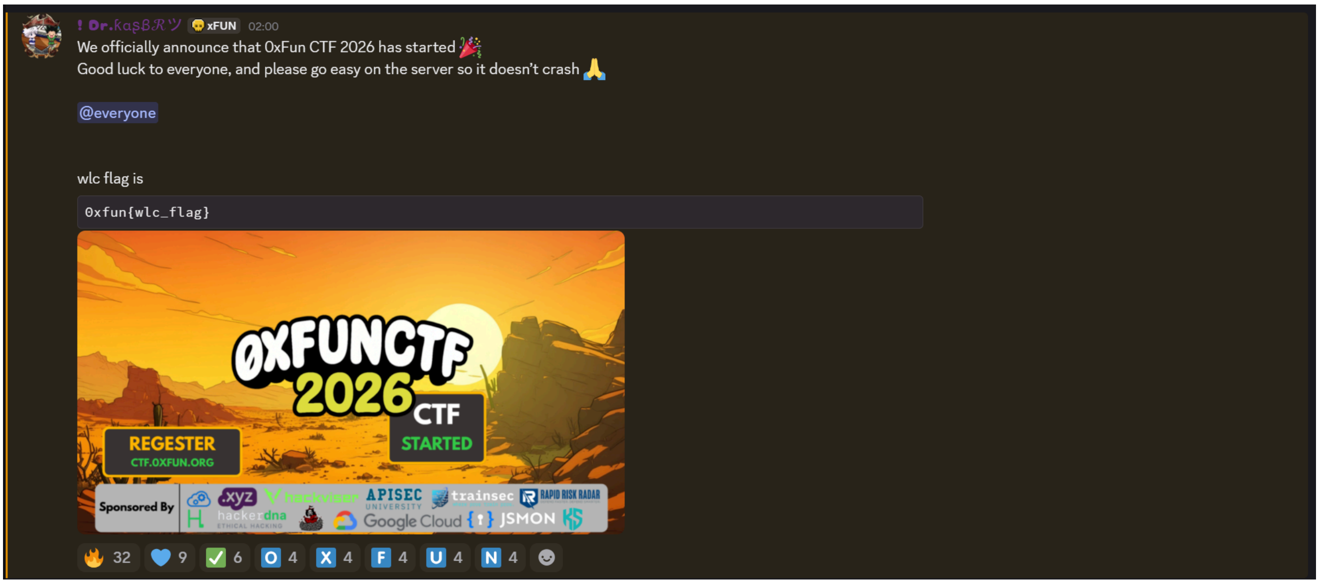
0xFunCTF WriteUp

by zijeff

Welcome

打开官方的discord服务器，稍加查看后即可得到flag。



Leonine Misbegotten

理解题目意思后就很好做了，每轮操作对当前状态随机选取一种编码方式加密，从而得到下一状态的一部分，再对当前状态求 **sha1** 作为下一状态的另一部分，拼接得到下一状态。

$$s_{n+1} = f(s_n) + sha1(s_n)$$

因为 **sha1** 得到的字节数为固定的，所以可以提取最后的20字节，遍历四种编码方式求解原码，再将其 **sha1** 值进行比对校验，16次循环处理后即可得到结果。

具体代码如下：

```
from base64 import b16decode, b32decode, b64decode, b85decode
from hashlib import sha1

# 定义解码函数列表
DECODERS = [b16decode, b32decode, b64decode, b85decode]

def solve():
    # 1. 读取 output 文件内容
    with open("output", "rb") as f:
        current = f.read()

    rounds = 16

    for i in range(rounds):
        # 2. 提取最后 20 字节的校验和
```

```python
        checksum = current[-20:]
        # 3. 提取前面的编码数据
        payload = current[:-20]

        found = False
        # 4. 尝试四种解码方式
        for decode_func in DECODERS:
            try:
                decoded = decode_func(payload)
                # 5. 校验 SHA1 是否匹配
                if sha1(decoded).digest() == checksum:
                    current = decoded
                    print(f"第 {i+1} 轮解密成功: 使用了
{decode_func.__name__}")
                    found = True
                    break
            except Exception:
                # 如果解码失败（比如字符不符合规范），尝试下一种
                continue

        if not found:
            print(f"错误: 在第 {i+1} 轮未能匹配到正确的解码方式。")
            break

    # 输出结果
    print("\n最终结果 (Flag):")
    print(current.decode(errors='ignore'))

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

运行后得到结果:

```
第 9 轮解密成功: 使用了 b16decode
第 10 轮解密成功: 使用了 b16decode
第 11 轮解密成功: 使用了 b85decode
第 12 轮解密成功: 使用了 b64decode
第 13 轮解密成功: 使用了 b85decode
第 14 轮解密成功: 使用了 b64decode
第 15 轮解密成功: 使用了 b16decode
第 16 轮解密成功: 使用了 b16decode

最终结果 (Flag):
0xfun{p33l1ng_l4y3rs_l1k3_an_0n10n}
```

The Slot Whisperer

显然这是一个线性同余生成器相关的问题，生成过程如下:

$$s_{n+1} = as_n + c \mod m$$

其中a, c, m我们全部已知，seed的值不知道，我们根据输出可以知道生成的前十个数字。进而我们知道

$$s_i = n_i + 100k \quad k \in \mathbb{Z}$$

又因为每次的state是模了m后的结果，所以必然小于m，因此我们可以爆破k的值从而恢复LCG的初始状态。

> **Tip**
>
> 因为$m = 2147483647$，而循环步长为100，所以只需要2100万次左右的循环即可，这是很快的。

具体代码如下:

```
def solve():
    # 题目给出的已知序列
    targets = [71, 6, 79, 39, 94, 28, 98, 72, 1, 76]

    # LCG 参数
    M = 2147483647
    A = 48271
    C = 12345

    print(f"[*] 正在爆破初始状态，正在验证完整序列: {targets}")
    found_state = None
    for k in range(M // 100 + 2):
        candidate = k * 100 + targets[0]
        if candidate >= M:
            break
        temp_state = candidate
        is_valid = True
        for i in range(1, len(targets)):
            temp_state = (A * temp_state + C) % M
            if temp_state % 100 != targets[i]:
                is_valid = False
                break

        if is_valid:
            found_state = candidate
            print(f"[+] 找到完美匹配的状态: {found_state}")
            break

    if found_state is not None:
        current_state = found_state

        for i in range(len(targets) - 1):
             current_state = (A * current_state + C) % M

        print("[+] 预测接下来的 5 个数字: ")
        predictions = []
        for _ in range(5):
            current_state = (A * current_state + C) % M
```

```
            output = current_state % 100
            predictions.append(str(output))

        print(" ".join(predictions))
    else:
        print("[-] 未找到解，可能是输入序列有误。")

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

运行截图如下：

```
smith@jeff-ciallo:~/tasks/0xFunCTF2026/The_Slot_Whisperer$ /bin/python3 /home/smith/tasks/0xFunCTF2026/The_Slot_Whisperer/exp.py
[*] 正在爆破初始状态，正在验证完整序列: [71, 6, 79, 39, 94, 28, 98, 72, 1, 76]
[+] 找到完美匹配的状态: 1523311471
[+] 预测接下来的 5 个数字:
53 69 71 78 40
```

提交后得到flag：

```
Predict the next 5 spins (space-separated): 53 69 71 78 40
JACKPOT! You've mastered the slot machine!
0xfun{sl0t_wh1sp3r3r_lcg_cr4ck3d}
```

MeOwl ECC

正常的椭圆曲线加密，题目使用私钥d作为加密密钥，求解思路在于求出d的值。尝试对曲线的阶进行分解，发现无法分解，但是可以得到曲线的阶等于模数p，所以我们可以使用 **Smart's Attack** 进行求解。

具体代码如下：

```
from sage.all import *

p =
1070960903638793793346073212977144745230649115077006408609822474051879875
814028659881855169
A = 0
B = 19
E = EllipticCurve(GF(p),[A,B])

Px =
8501944241313638385889097726391817163665759180015566294919862065642775888
35368712774900915
Py =
7495097064006679768827721826635063839521197238483009004818601469566312780
2641792062633486

Qx =
5425035864266975615401513495015263668243752271578636331175994098138359208
3045988845753867
```

```
Qy =
324772290891069325219931358863917293864610371020855881775477694333357303867104131696431188

P = E(Px, Py)
Q = E(Qx, Qy)
def SmartAttack(P,Q,p):
    E = P.curve()
    Eqp = EllipticCurve(Qp(p, 2), [ ZZ(t) + randint(0,p)*p for t in
E.a_invariants() ])

    P_Qps = Eqp.lift_x(ZZ(P.xy()[0]), all=True)
    for P_Qp in P_Qps:
        if GF(p)(P_Qp.xy()[1]) == P.xy()[1]:
            break

    Q_Qps = Eqp.lift_x(ZZ(Q.xy()[0]), all=True)
    for Q_Qp in Q_Qps:
        if GF(p)(Q_Qp.xy()[1]) == Q.xy()[1]:
            break

    p_times_P = p*P_Qp
    p_times_Q = p*Q_Qp

    x_P,y_P = p_times_P.xy()
    x_Q,y_Q = p_times_Q.xy()

    phi_P = -(x_P/y_P)
    phi_Q = -(x_Q/y_Q)
    k = phi_Q/phi_P
    return ZZ(k)

d = SmartAttack(P, Q, p)
print(d)

import hashlib
from Crypto.Cipher import AES, DES
from Crypto.Util.Padding import unpad
from Crypto.Util.number import long_to_bytes

d_found = d  # <--- 在这里填入上面算出的 d

# 题目密文参数
aes_iv_hex = "7d0e47bb8d111b626f0e17be5a761a14"
des_iv_hex = "86fd0c44751700d4"
ciphertext_hex = (
    "7d34910bca6f505e638ed22f412dbf1b50d03243b739de0090d07fb097ec0a2c"
    "a19158949f32e39cd84adea33d2229556f635237088316d2"
)
```

```python
def decrypt():
    if d_found == 0:
        print("[-] 请先运行 SageMath 脚本算出 d，并填入 d_found 变量")
        return

    # 1. 还原密钥
    k = long_to_bytes(d_found)
    aes_key = hashlib.sha256(k + b"MeOwl::AES").digest()[:16]
    des_key = hashlib.sha256(k + b"MeOwl::DES").digest()[:8]

    ciphertext = bytes.fromhex(ciphertext_hex)
    aes_iv = bytes.fromhex(aes_iv_hex)
    des_iv = bytes.fromhex(des_iv_hex)

    try:
        # 2. 解密外层 (DES)
        # DES 使用 pad(c1, 8)，解密后需 unpad 8
        des_cipher = DES.new(des_key, DES.MODE_CBC, iv=des_iv)
        c1_padded = des_cipher.decrypt(ciphertext)
        c1 = unpad(c1_padded, 8)

        # 3. 解密内层 (AES)
        # AES 使用 pad(flag, 16)，解密后需 unpad 16
        aes_cipher = AES.new(aes_key, AES.MODE_CBC, iv=aes_iv)
        flag_padded = aes_cipher.decrypt(c1)
        flag = unpad(flag_padded, 16)

        print(f"[+] Flag: {flag.decode()}")

    except Exception as e:
        print(f"[-] Decryption failed: {e}")

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

运行得到 flag：*oxfun{non_c4non1c4l_l1f7s_r_cool}*

BitStorm

看似十分复杂的加密过程，仔细查看后发现都是在二进制域 **GF(2)** 上是线性的，所以只要输出够多就可以建立线性方程组求解。

具体代码如下：

```python
from sage.all import *

# ==========================================
# 1. 符号化整数类
# ==========================================
```

```
DIM = 2048
F = GF(2)

class SymInt:
    """
    模拟 64 位整数，使用 GF(2) 向量加法代替 XOR
    """
    def __init__(self, bits=None):
        if bits is None:
            # 默认为 0 (全零向量)
            self.bits = [vector(F, DIM) for _ in range(64)]
        else:
            self.bits = bits
            # 截断或填充至 64 位
            if len(self.bits) > 64:
                self.bits = self.bits[:64]
            elif len(self.bits) < 64:
                self.bits += [vector(F, DIM) for _ in range(64 -
len(self.bits))]
    def __add__(self, other):
        # 对应位向量相加 (GF2 加法即异或)
        new_bits = [a + b for a, b in zip(self.bits, other.bits)]
        return SymInt(new_bits)

    def __lshift__(self, n):
        # 左移 n: 低位补零向量，高位丢弃
        if n == 0: return self
        if n >= 64: return SymInt()
        zero_vec = vector(F, DIM)
        new_bits = [zero_vec] * n + self.bits[:-n]
        return SymInt(new_bits)

    def __rshift__(self, n):
        # 右移 n: 高位补零向量，低位丢弃
        if n == 0: return self
        if n >= 64: return SymInt()
        zero_vec = vector(F, DIM)
        new_bits = self.bits[n:] + [zero_vec] * n
        return SymInt(new_bits)

    def rotate_left(self, n):
        # 循环左移: (x << n) | (x >> (64-n))
        n = n % 64
        if n == 0: return self
        new_bits = [self.bits[(i - n) % 64] for i in range(64)]
        return SymInt(new_bits)

# ============================================
# 2. 模拟 RNG 逻辑 (使用 + 代替 ^)
# ============================================
```

```python
class SymbolicGiantLinearRNG:
    def __init__(self):
        self.state_size = 32
        self.state = []

        for i in range(self.state_size):
            shift = 64 * (self.state_size - 1 - i)
            bits = []
            for bit_idx in range(64):
                # bit_idx 0 is LSB of this chunk
                # 对应 seed 的绝对位位置
                seed_bit_index = shift + bit_idx

                # 创建基向量
                v = vector(F, DIM)
                if seed_bit_index < DIM:
                    v[seed_bit_index] = 1
                bits.append(v)
            self.state.append(SymInt(bits))

    def next(self):
        s = self.state
        taps = [0, 1, 3, 7, 13, 22, 28, 31]

        new_val = SymInt() # 0

        for i in taps:
            val = s[i]
            mixed = val + (val << 11) + (val >> 7)

            rot = (i * 3) % 64
            mixed = mixed.rotate_left(rot)

            new_val = new_val + mixed

        # new_val ^= (s[-1] >> 13) ^ ((s[-1] << 5) & mask)
        new_val = new_val + (s[-1] >> 13) + (s[-1] << 5)

        # 更新状态
        self.state = s[1:] + [new_val]

        # 计算输出
        out = SymInt()
        for i in range(self.state_size):
            if i % 2 == 0:
                out = out + self.state[i]
            else:
                val = self.state[i]
                # out ^= ((val >> 2) | (val << 62)) -> Rotate Left 62
```

```
                out = out + val.rotate_left(62)

        return out

# ==========================================
# 3. 主求解函数
# ==========================================

def solve():
    real_outputs = [
        11329270341625800450, 14683377949987450496, 11656037499566818711,
14613944493490807838,
        3705323136265793 29, 5006729399082841610, 8072429272270319226,
3035866339305997883,
        8753420467487863273, 15606411394407853524, 5092825474622599933,
6483262783952989294,
        15380511644426948242, 13769333495965053018, 5620127072433438895,
6809804883045878003,
        1965081297255415258, 2519823891124920624, 8990634037671460127,
3616252826436676639,
        1455424466699459058, 2836976688807481485, 11291016575083277338,
1603466311071935653,
        14629944881049387748, 3844587940332157570, 5842526375675565 89,
10739738025866331065,
        11650614949586184265, 18287913478034970 22, 9101164617572571488,
16034652114565169975,
        13629596693592688618, 17837636002790364294, 10619900844581377650,
15079130325914713229,
        5515526762186744782, 1211604266555550739, 11543408140362566331,
1842529427012603035 5,
        2629175584127737886, 6074824578506719227, 6900475985494339491,
3263181255912585281,
        12421969688110544830, 10785482337735433711, 10286647144557317983,
15284226677373655118,
        9365502412429803694, 4248763523766770934, 13642948918986007294,
3512868807899248227,
        14810275182048896102, 16743417430432403 80, 2846246760286049 9,
1060872896572731679,
        13208674648176077254, 14702937631401007104, 5386638277617718038,
8935128661284199759
    ]

    print("[*] Initializing Symbolic RNG...")
    rng = SymbolicGiantLinearRNG()

    matrix_rows = []
    target_vector = []

    # 取 40 个输出 (40 * 64 > 2048) 确保满秩
    print("[*] Collecting equations...")
```

```python
    for k, real_val in enumerate(real_outputs[:40]):
        sym_val = rng.next()

        for bit_i in range(64):
            # 获取符号化系数 (向量)
            coeffs = sym_val.bits[bit_i]

            # 获取真实值比特
            target_bit = (real_val >> bit_i) & 1

            matrix_rows.append(coeffs)
            target_vector.append(target_bit)

    print(f"[*] Constructing Matrix ({len(matrix_rows)} x {DIM})...")

    M = Matrix(F, matrix_rows)
    b = vector(F, target_vector)

    print("[*] Solving linear system...")
    try:
        solution = M.solve_right(b)
    except ValueError as e:
        print("[-] Solver failed.")
        print(e)
        return

    print("[*] Reconstructing Flag...")

    seed_int = 0
    # solution 向量的索引 i 直接对应 seed_int 的第 i 位
    for i in range(DIM):
        if solution[i] == 1:
            seed_int |= (1 << i)

    try:
        flag_bytes = int(seed_int).to_bytes(256, 'big')
        flag = flag_bytes.replace(b'\x00', b'')
        print("\n" + "="*60)
        print("FLAG:", "0xfun{" + flag.decode(errors='ignore') + "}")
        print("="*60 + "\n")
    except Exception as e:
        print(f"[-] Decoding error: {e}")

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

运行得到结果:

```
[*] Initializing Symbolic RNG...
[*] Collecting equations...
[*] Constructing Matrix (2560 x 2048)...
[*] Solving linear system...
[*] Reconstructing Flag...

============================================================
FLAG: 0xfun{L1n34r_4lg3br4_W1th_Z3_1s_Aw3s0m3}
============================================================
```

The Roulette Conspiracy

很正常的 MT19937 随机数生成的问题，链接容器后发现没有限制 spin 的次数，所以我们可以获取连续的624个输出从而恢复随机数生成器的初始状态，进而预测接下来的10个输出。

唯一值得注意的就是脚本的数据上传格式（），具体代码如下：

```python
import random
from pwn import *

# --- 1. MT19937 逆向核心函数 ---
def inverse_right_shift(val, shift):
    res = val
    for i in range(32 // shift):
        res = val ^ (res >> shift)
    return res

def inverse_left_shift(val, shift, mask):
    res = val
    for i in range(32 // shift):
        res = val ^ ((res << shift) & mask)
    return res

def untemper(v):
    """撤销 MT19937 的 Tempering 过程，还原内部状态数组的值"""
    v = inverse_right_shift(v, 18)
    v = inverse_left_shift(v, 15, 0xefc60000)
    v = inverse_left_shift(v, 7, 0x9d2c5680)
    v = inverse_right_shift(v, 11)
    return v

# --- 2. 网络交互配置 ---
HOST = 'chall.0xfun.org'
PORT = 43843

io = remote(HOST, PORT)

def get_next_raw():
    """发送 spin 指令并获取还原 XOR 后的原始随机数"""
    # 匹配提示符 "> " (注意可能有空格)
    io.sendlineafter(b">", b"spin")
    # 接收返回的一行数字
```

```python
    line = io.recvline().decode().strip()
    # 兼容性处理：如果读到的是空行，再读一行
    while not line or not line.isdigit():
        line = io.recvline().decode().strip()

    obfuscated = int(line)
    # 题目逻辑：obfuscated = raw ^ 0xCAFEBABE
    raw = obfuscated ^ 0xCAFEBABE
    return raw

# --- 3. 收集 624 个状态以克隆生成器 ---
print("[*] 正在从服务器收集 624 个数据点...")
state_elements = []
for i in range(624):
    raw_val = get_next_raw()
    state_elements.append(untemper(raw_val))
    if (i + 1) % 100 == 0:
        print(f"已收集: {i+1}/624")

print("[+] 状态还原成功! ")

# --- 4. 同步本地生成器 ---
# 设置 index 为 624，确保下一次 getrandbits 会触发 twist() 生成下一组序列
predictor = random.Random()
predictor.setstate((3, tuple(state_elements + [624]), None))

# --- 5. 提交预测 (一行提交) ---
print("[*] 正在准备预测结果...")
io.sendlineafter(b">", b"predict")

# 生成接下来 10 个原始值 (raw bits)
predictions = []
for i in range(10):
    val = predictor.getrandbits(32)
    predictions.append(str(val))
    print(f"预测值 {i+1:2d}: {val}")

# 将 10 个数字用空格连接成一行发送
payload = " ".join(predictions)
print(f"[*] 发送 Payload: {payload}")
io.sendline(payload.encode())

# --- 6. 获取 Flag ---
print("[!] 等待服务器响应 Flag...")
try:
    # 使用 interactive() 方便在拿到 Flag 后手动操作，或直接 recvall
    # io.interactive()
    print(io.recvall(timeout=5).decode())
except Exception as e:
    print("\n[!] 读取结束或连接已关闭。")
```

```
io.close()
```

运行后得到结果：

```
[*] 发送 Payload: 2415991180 191537815 3733612075 2078345470 4270005197 3281523449 442089945 4279216897 2896
412834 3420731745
[!] 等待服务器响应 Flag...
[+] Receiving all data: Done (128B)
[*] Closed connection to chall.0xfun.org port 43843
 Predict next 10 raw values (space-separated): PERFECT! You've untwisted the Mersenne Oracle!
0xfun{m3rs3nn3_tw1st3r_unr4v3l3d}
```

Hawk_II

打开一看快吓死我了，代码量是真的大。但仔细观察后我们发现，题目好像直接在输出中打印了私钥。这下爽了，直接一个简单脚本带走。

```python
from hashlib import sha256
from Crypto.Cipher import AES
from Crypto.Util.Padding import unpad

# ==========================================
# 1. 填入题目输出的 IV 和 ENC
# ==========================================
iv_hex = "ac518ee77848d87912548668d3240aa4"
enc_hex =
"ab425b6c2c0a6760a5e9c52ba25dfc47da97afeeceb9823e553dcccc971b0f25c876ea63
ed867d77e3295082064a3f69"

# ==========================================
# 2. 填入题目输出的 sk 字符串
# ==========================================
```

```
sk_str = """(z^255 + z^254 + 4*z^253 - z^252 - z^251 + 2*z^250 + 3*z^249
+ 3*z^246 + z^244 + z^243 - 3*z^242 - 2*z^240 - z^238 - z^237 + 2*z^236 +
2*z^235 - z^234 + 2*z^233 + 3*z^232 - 2*z^231 - z^230 + 2*z^229 - z^228 -
3*z^227 + z^226 - z^224 + 3*z^223 + z^222 + z^221 - 2*z^220 + z^219 -
3*z^218 + 2*z^217 - z^216 + 3*z^215 + 6*z^214 + z^213 - 2*z^212 - 2*z^211
- 3*z^210 - 2*z^209 - z^208 - 3*z^207 - z^206 + z^205 + z^204 + 3*z^203 -
z^202 + 3*z^201 + z^200 - 2*z^199 - 5*z^198 - 2*z^197 - 2*z^196 - 2*z^195
+ 3*z^194 + 6*z^193 - 2*z^192 + z^191 - 2*z^190 - z^189 + z^187 + 2*z^186
- 3*z^185 - 5*z^184 + 5*z^182 - z^181 + 2*z^180 - 4*z^179 + 3*z^178 -
z^177 + z^175 - 4*z^173 - 2*z^171 + z^169 + 2*z^168 + z^167 - z^166 +
4*z^165 - 3*z^164 - 3*z^163 - 3*z^162 + z^161 - 3*z^160 - 3*z^159 +
2*z^158 - 2*z^156 - 2*z^155 + 3*z^154 - z^153 - 2*z^152 - z^151 - 2*z^150
- z^149 + z^148 + z^147 - 4*z^146 - 2*z^145 + 3*z^144 + 4*z^143 - z^142 -
2*z^141 + 2*z^139 - 2*z^138 - 3*z^136 + z^135 + z^134 + 3*z^133 - z^132 +
z^131 - 4*z^130 - 2*z^129 - z^128 + z^127 - z^126 - 3*z^125 + z^124 +
2*z^123 + 3*z^122 - 4*z^120 + 6*z^119 - 3*z^118 + z^117 + z^116 + 4*z^115
+ 2*z^114 - z^113 + 2*z^112 - z^111 - z^110 - 4*z^109 - 3*z^107 + z^106 +
3*z^105 + 2*z^103 - z^102 + z^101 - z^100 + z^98 + z^95 - 3*z^94 + z^93 -
2*z^92 + z^91 - 5*z^90 + 3*z^89 - z^88 - z^87 - 4*z^86 + z^85 - z^84 +
2*z^82 - 2*z^81 + z^78 + z^75 + 5*z^74 + z^73 + z^71 - 3*z^70 - z^68 +
2*z^67 - z^66 + z^65 - 2*z^64 + 2*z^63 + z^62 - 3*z^61 - 2*z^60 - 3*z^58
+ 2*z^57 + z^56 - 2*z^55 + z^54 + z^53 + 4*z^52 - z^51 + z^50 - z^49 +
z^48 + z^47 - z^46 + 2*z^45 - z^43 - z^42 + z^41 - z^40 - 7*z^39 + 3*z^38
+ 3*z^37 + z^36 - z^35 + 2*z^34 + 2*z^31 + 2*z^29 - 3*z^28 - z^26 + z^25
- z^24 + 2*z^23 - 3*z^22 - z^21 + z^20 - 2*z^18 + z^17 + z^15 - z^14 +
2*z^12 - 2*z^11 - z^10 + 2*z^9 + z^8 - z^7 + z^6 - 3*z^5 - z^4 + 3*z^3 -
5*z^2 - z, 2*z^255 - z^254 + z^253 + z^252 - 4*z^250 + 2*z^249 - z^248 +
z^246 + z^244 + 2*z^242 - z^241 + 2*z^239 + 2*z^238 - 2*z^237 - 3*z^236 +
z^235 - 2*z^234 + z^233 + 3*z^231 + z^230 + z^229 + 3*z^228 - 2*z^227 -
z^226 - z^224 - z^223 + 4*z^222 - 3*z^220 - z^219 + 2*z^218 - z^217 -
2*z^216 + 3*z^215 + 2*z^214 - 2*z^213 + z^210 + z^209 + 5*z^207 + z^206 -
z^205 - z^202 - 2*z^200 - 2*z^199 + z^198 + z^197 - z^196 + 3*z^195 +
4*z^194 - 2*z^193 + 3*z^191 - z^189 + 5*z^188 - 5*z^187 + z^185 + 3*z^184
- 4*z^183 + 3*z^182 - z^181 - z^180 + 4*z^179 + 3*z^178 - 3*z^176 + z^175
- z^174 + z^173 - 3*z^171 + z^169 - z^168 - 3*z^167 - z^165 + z^164 +
2*z^162 - z^161 - 2*z^160 + z^158 + 2*z^156 + z^155 + 2*z^154 - z^153 +
4*z^152 - 4*z^151 - 3*z^150 + 4*z^149 - 2*z^147 + z^146 - z^145 - z^144 +
z^143 + z^142 - 2*z^140 + 2*z^139 + 2*z^138 + 3*z^137 - 2*z^136 + z^135 -
3*z^134 - 3*z^133 - z^132 + z^131 + 2*z^130 - z^129 + 4*z^128 - z^127 -
z^126 + z^125 - 3*z^124 - z^123 + z^122 + z^121 - 3*z^120 + z^119 + z^118
+ 3*z^117 - 2*z^116 - z^115 + z^114 + 2*z^113 + 4*z^112 + 2*z^111 -
2*z^110 + z^109 - 2*z^108 + z^107 - 2*z^105 + 4*z^103 - z^101 - 2*z^100 +
6*z^98 + 3*z^97 - z^94 - 2*z^92 - z^91 - 3*z^90 + z^89 + z^88 + 2*z^87 +
z^86 - z^85 + 2*z^84 + 2*z^83 - z^82 - 3*z^81 + 3*z^80 - 2*z^79 - 2*z^78
+ 2*z^77 + 3*z^76 + z^75 + z^70 - z^69 - z^68 + 2*z^67 - 2*z^66 - z^65 -
3*z^64 - z^63 - 2*z^62 + z^61 + 2*z^60 + 3*z^59 + z^58 + z^57 - 5*z^56 +
2*z^55 + 3*z^54 + z^53 + 4*z^52 + z^51 - z^50 - z^49 + z^48 - 3*z^47 -
z^46 + 2*z^44 - z^42 - 3*z^41 + 3*z^38 + z^37 - 2*z^35 - 2*z^34 - z^33 +
z^32 + 4*z^31 + 2*z^30 - 2*z^29 - 2*z^27 + z^25 - z^24 - 3*z^23 + 2*z^21
+ 2*z^17 - z^16 - 3*z^15 + 2*z^14 + z^12 + 2*z^11 + 2*z^10 + 3*z^9 +
```

4*z^8 - 2*z^6 - z^5 + 3*z^4 + z^2 + z - 2, 13*z^255 - 17*z^254 - 3*z^253
- 8*z^252 + 7*z^251 - 15*z^250 + 21*z^249 + 11*z^248 - 7*z^247 + 22*z^246
+ 3*z^245 - 10*z^244 - 8*z^243 - 12*z^242 + 5*z^241 + 8*z^240 + 4*z^239 -
6*z^238 + 16*z^237 - 2*z^236 + 21*z^235 - z^234 - 14*z^233 - z^232 +
11*z^231 - 5*z^230 + 3*z^229 + 12*z^228 + 11*z^227 + 5*z^226 + 2*z^225 +
18*z^224 - 13*z^222 + 14*z^221 + 6*z^220 - 3*z^219 + 13*z^218 + 2*z^217 -
11*z^216 - 3*z^215 + 8*z^214 + z^213 - 3*z^212 - 11*z^211 - 4*z^210 +
2*z^209 - 16*z^208 + 11*z^207 + 5*z^206 - 12*z^205 + 17*z^204 + 15*z^203
- 5*z^201 - 19*z^200 - 8*z^199 - 11*z^198 + 13*z^197 - 18*z^196 +
18*z^195 - 6*z^194 + z^193 + 6*z^192 - 11*z^191 + 2*z^190 + 7*z^189 -
4*z^188 + 2*z^187 - z^186 + 5*z^185 - 13*z^184 - 6*z^183 - 4*z^182 +
19*z^181 - 4*z^180 + 20*z^179 - 11*z^178 + 5*z^177 - 5*z^176 - z^175 +
z^174 + 2*z^172 + 4*z^171 + 3*z^170 + 3*z^169 - 7*z^168 - 9*z^167 -
14*z^166 + 13*z^165 + 2*z^164 - 4*z^163 - 3*z^162 + 7*z^161 + 17*z^160 +
8*z^159 + 6*z^158 - 16*z^157 + 11*z^156 + 17*z^155 + 2*z^154 - 4*z^153 -
5*z^152 + 5*z^151 - 4*z^150 - 2*z^149 + 3*z^148 - 3*z^147 - 7*z^146 +
12*z^145 + 5*z^144 + 5*z^143 - 11*z^142 - 5*z^141 - 9*z^140 - 3*z^139 +
11*z^137 - 7*z^136 + 14*z^135 + 5*z^134 + 12*z^133 - z^132 - z^131 -
14*z^130 - 3*z^129 - 8*z^128 + 7*z^127 + 11*z^126 + 12*z^125 - 2*z^124 +
6*z^123 - 6*z^122 - 2*z^121 - 20*z^120 - 6*z^119 + 2*z^118 + 11*z^117 -
4*z^116 - 9*z^115 - 5*z^114 + 3*z^113 + 5*z^112 + 4*z^111 + 6*z^109 -
8*z^108 + 12*z^107 - z^106 + 6*z^105 + z^104 - 11*z^103 - 9*z^102 -
5*z^101 + 4*z^100 + 5*z^99 + z^98 - 6*z^97 - 9*z^96 + 6*z^95 + 3*z^94 +
10*z^93 - 5*z^92 - 12*z^91 + 4*z^90 - 10*z^89 + 9*z^88 + z^87 + 3*z^86 +
z^85 + 17*z^84 + 4*z^83 - 25*z^82 - 4*z^81 - 3*z^80 + 23*z^79 - 6*z^78 -
4*z^77 - 5*z^76 + 7*z^75 + 2*z^74 + 7*z^73 + z^71 - 12*z^70 + 6*z^69 +
7*z^68 - 9*z^67 - 16*z^66 + 16*z^65 - 10*z^64 + 11*z^63 - 6*z^62 + 5*z^61
- 8*z^60 - 3*z^58 + z^57 - 5*z^56 + 11*z^54 - 4*z^53 + 4*z^52 + 7*z^51 -
17*z^50 - 11*z^49 + 14*z^48 + 4*z^47 + 8*z^46 - 7*z^45 - 5*z^44 + 5*z^43
+ 9*z^42 - 2*z^41 + 6*z^40 - 9*z^39 - 23*z^38 - 7*z^37 - 10*z^36 +
10*z^35 - z^34 + z^33 - z^32 + 12*z^31 + 9*z^30 - 5*z^29 - z^28 - 18*z^27
+ 2*z^26 - 2*z^25 - 6*z^24 + 11*z^23 - 5*z^22 + 5*z^21 + 4*z^20 + 3*z^19
- 2*z^18 - 5*z^17 - 8*z^16 + 8*z^15 + 14*z^14 + 5*z^12 - 12*z^11 -
27*z^10 + 20*z^9 + 10*z^8 - 12*z^7 - 3*z^6 + 8*z^5 - 7*z^4 + 12*z^3 +
4*z^2 - 20*z - 10, -z^255 + 5*z^254 - 4*z^253 - 5*z^252 + 4*z^251 +
3*z^249 - 13*z^248 - 16*z^247 - 7*z^246 + 3*z^245 + 2*z^244 + 3*z^243 +
10*z^242 + 7*z^241 - 15*z^240 - 11*z^239 - 7*z^238 - 5*z^237 + 3*z^236 +
10*z^235 + 3*z^234 - 16*z^233 + 3*z^232 - 13*z^230 + 10*z^229 + 9*z^228 +
14*z^227 - 4*z^226 + 9*z^225 - 18*z^224 - 23*z^223 + 4*z^222 - z^221 +
3*z^219 + 17*z^218 - 9*z^217 + 2*z^216 + 10*z^215 - 17*z^214 - 14*z^213 +
15*z^212 + 21*z^211 - 10*z^210 + 22*z^209 - 29*z^207 - 19*z^206 + 6*z^205
+ 5*z^204 - 12*z^203 + 15*z^202 + 12*z^201 - 3*z^200 - 6*z^199 + 14*z^198
- 6*z^197 - 4*z^196 + 11*z^195 - 9*z^194 - z^193 - 7*z^191 - 2*z^190 -
2*z^189 - 3*z^187 - 8*z^186 + 2*z^185 + 5*z^184 + 4*z^183 + 3*z^182 -
10*z^180 - 14*z^179 + 6*z^178 - 3*z^177 - 8*z^176 + 12*z^175 + 11*z^174 +
z^173 - 2*z^172 - 13*z^171 - 24*z^170 + 2*z^168 - 11*z^167 + 6*z^166 +
7*z^165 + 8*z^164 - 8*z^163 + 5*z^162 - 9*z^161 + 4*z^160 + 7*z^159 +
9*z^158 - 2*z^157 - 7*z^156 + z^155 - 18*z^154 - 3*z^153 - 5*z^152 +
19*z^151 - z^150 - 3*z^149 - 2*z^148 - 13*z^147 + 3*z^146 - z^145 +
4*z^144 - 7*z^143 + 5*z^142 + z^141 - 3*z^140 - z^139 - z^138 - 16*z^137

```python
- 13*z^136 + 10*z^135 + 14*z^134 + z^133 + 18*z^132 - 18*z^131 - 3*z^130
- 4*z^129 + 9*z^128 - 13*z^127 - z^126 + 4*z^125 + 3*z^124 + z^123 -
10*z^122 + 6*z^121 + 8*z^120 + 6*z^119 + 7*z^118 + 11*z^117 - 9*z^116 +
7*z^115 + 14*z^114 - 13*z^113 - 8*z^112 + 12*z^111 + 4*z^110 + 5*z^109 +
10*z^108 - 15*z^107 - 14*z^106 + 10*z^105 + 2*z^104 - z^103 - 13*z^102 +
12*z^101 + 7*z^100 + 4*z^99 - z^97 - 8*z^96 - 7*z^95 + 5*z^94 + 4*z^93 +
3*z^92 - 10*z^91 - 2*z^90 + 4*z^89 + 18*z^88 + 6*z^87 + 7*z^86 + 4*z^84 -
5*z^83 - 3*z^82 + 7*z^81 - 17*z^80 - 10*z^79 + 2*z^78 - 7*z^77 - 6*z^76 +
2*z^75 + z^74 - 15*z^73 + 14*z^72 + 14*z^71 - 6*z^70 + 16*z^69 + 6*z^68 +
2*z^67 - 4*z^66 + 5*z^65 - 10*z^64 + z^63 - 5*z^62 + 4*z^61 + z^60 -
5*z^59 + 5*z^58 + 8*z^57 - 2*z^56 - z^55 + 16*z^54 - 4*z^53 - 9*z^52 -
10*z^50 + 14*z^48 + z^47 - 21*z^46 - 6*z^45 + 15*z^44 + 9*z^43 + 4*z^42 +
11*z^41 + 4*z^40 - 6*z^39 + 2*z^37 - 15*z^36 + 3*z^35 + 13*z^34 + 7*z^33
- 14*z^32 - 9*z^31 - z^30 - 4*z^29 + 8*z^28 + 9*z^27 - z^26 - 6*z^25 +
19*z^24 + 14*z^23 - 15*z^22 + 4*z^21 + 4*z^20 - 5*z^19 + 4*z^18 + 14*z^17
+ 8*z^16 + z^15 - 3*z^14 - 12*z^13 + 3*z^12 + 3*z^11 + 12*z^10 + 7*z^9 +
4*z^8 + 11*z^7 - z^6 - 10*z^5 + 7*z^4 - z^3 - 16*z^2 + 3*z + 13)"""

# ==========================================
# 3. 解密
# ==========================================
try:
    # 生成解密密钥
    key = sha256(sk_str.encode()).digest()

    # 转换 hex 为 bytes
    iv = bytes.fromhex(iv_hex)
    enc = bytes.fromhex(enc_hex)

    # AES 解密
    cipher = AES.new(key=key, mode=AES.MODE_CBC, iv=iv)
    plaintext = cipher.decrypt(enc)

    # 去除填充并打印 Flag
    flag = unpad(plaintext, 16)
    print("[SUCCESS] Flag found:")
    print(flag.decode())

except Exception as e:
    print(f"[ERROR] Decryption failed: {e}")
```

运行后得到结果:

```
smith@jeff-ciallo:~/tasks/0xFunCTF2026/Hawk_II$ /bin/python3 /home/smith/tasks/0xFunCTF2026/Hawk_II/exp.py
[SUCCESS] Flag found:
0xfun{tOO_LLL_256_B_kkkkKZ_t4e_f14g_F14g}
smith@jeff-ciallo:~/tasks/0xFunCTF2026/Hawk_II$
```

The Fortune Teller

Gemini 得了 MVP，我就是躺赢狗（）。

与 **The Slot Whisperer** 类似，但因为数字很大，我们不能使用爆破了。因为泄露了高32位，同时给出了三个已知输出，很显然我们会思考去构造一个格出来，然后用**格基规约**求解。

具体代码如下:

```
# SageMath 脚本
def solve():
    # 1. 定义常数
    M = 2^64
    A = 2862933555777941757
    C = 3037000493

    # 题目给出的观测值 (高32位)
    glimpses = [1332353447, 4117841481, 4217378265]

    # 2. 准备构建格所需的参数
    # 计算 K_i (Bias)
    # 关系推导: x_i = A^i * x_0 + K_i (mod M)
    # K_i = (A^i * y0_full + Cumulative_C_i - yi_full) % M

    K = []

    # 计算 A^i 和 累积 C
    # S_i = A^i * S_0 + term_c
    # term_c[0] = 0
    # term_c[1] = C
    # term_c[2] = A*C + C

    term_c = [0, C, (A*C + C) % M]
    term_a = [1, A, (A^2) % M]

    base_shift = 2^32
    y0_full = glimpses[0] * base_shift

    # 计算 K1, K2 (对应 glimpse[1], glimpse[2])
    # 注意：对于 i=0, x0 = 1*x0 + 0, 所以 K0=0，我们在矩阵中不需要 K0，只处理
K1, K2
    for i in range(1, 3):
        yi_full = glimpses[i] * base_shift
        # K_i = A^i * (y0_full) + term_c[i] - yi_full
        val = (term_a[i] * y0_full + term_c[i] - yi_full) % M
        K.append(val)

    # 3. 构建矩阵
    # 目标寻找短向量: (x0, x1, x2, 1)
    # Matrix B (4x4):
```

```python
# [ 1,     A,     A^2,   0 ]
# [ 0,     M,     0,     0 ]
# [ 0,     0,     M,     0 ]
# [ 0,     K1,    K2,    1 ]

B = Matrix(ZZ, 4, 4)
B[0, 0] = 1
B[0, 1] = term_a[1]
B[0, 2] = term_a[2]
B[0, 3] = 0

B[1, 1] = M
B[2, 2] = M

B[3, 1] = K[0] # K1
B[3, 2] = K[1] # K2
B[3, 3] = 1    # Constant constrol

print("Executing LLL reduction...")
L = B.LLL()

# 4. 从规约后的基中提取解
real_x0 = None

for row in L:
    # LLL 可能产生正负号翻转的向量，检查最后一维是 1 或 -1
    vec = list(row)
    if abs(vec[3]) == 1:
        potential_x0 = vec[0] * vec[3] # 修正符号

        # 验证 x0 范围
        if 0 <= potential_x0 < 2^32:
            # 进一步验证是否符合 glimpse 序列
            s0 = y0_full + potential_x0

            # 模拟推导
            curr = s0
            valid = True
            for g in glimpses:
                if (curr >> 32) != g:
                    valid = False
                    break
                curr = (curr * A + C) % M

            if valid:
                real_x0 = potential_x0
                print(f"\n[+] Found x0: {real_x0}")
                print(f"[+] Recovered Initial State S0: {s0}")
                break
```

```python
    if real_x0 is not None:
        # 5. 预测后5个状态
        # 当前我们验证完 glimpses 后，curr 已经是 S3 (即第3个输出之后的下一个状态)
        # 题目要求预测接下来 5 个

        state = (y0_full + real_x0)

        # 先空转前3次 (对应已知的3个输出)
        for _ in range(3):
            state = (state * A + C) % M

        print("\n[+] Next 5 full 64-bit states:")
        results = []
        for i in range(5):
            print(f"State {i+1}: {state}")
            results.append(str(state))
            state = (state * A + C) % M

        print("\nFinal Answer String:")
        print(" ".join(results))
    else:
        print("[-] LLL failed.")

solve()
```

运行后得到结果:

```
Executing LLL reduction...

[+] Found x0: 1162129244
[+] Recovered Initial State S0: 5722414482739998556

[+] Next 5 full 64-bit states:
State 1: 172645297224485511
State 2: 7633516833898572440
State 3: 5969584701298477925
State 4: 12291851195862230526
State 5: 13161189394887810867

Final Answer String:
172645297224485511 7633516833898572440 5969584701298477925 12291851195862230526 13161189394887810867
```

远程提交后即可得到 flag :

```
smith@jeff-ciallo:~$ nc chall.0xfun.org 64067
1332353447
4117841481
4217378265
Predict the next 5 full 64-bit states (space-separated): 172645297224485511 7633516833898572440 5969584701298477
925 12291851195862230526 13161189394887810867
IMPOSSIBLE! You've pierced the Fortune Teller's heart!
0xfun{trunc4t3d_lcg_f4lls_t0_lll}
```