R0tnoT13 Cryptography Write-Up

Challenge Description

A cryptographic subsystem leaked internal diagnostic values of the form S XOR ROTR(S, k), where S is a 128-bit internal state and k is a rotation offset. Several such values were captured along with their corresponding offsets. Additionally, two bits of the internal state were unspecified and a ciphertext encrypted using the internal state was provided. The objective was to reconstruct the internal state and recover the flag.

Key Observation

The expression S XOR ROTR(S, k) is linear over GF(2). Each leaked diagnostic frame produces 128 linear equations relating bits of the internal state. Because all rotation offsets are powers of two, the equations connect the entire bit space. As a result, the 128-bit internal state is reduced to only two unknown bits.

Attack Strategy

Each diagnostic value was translated into linear equations over GF(2). Gaussian elimination was applied to solve the system, recovering 126 bits of the internal state. The remaining two bits were brute-forced, resulting in four possible candidate states. Each candidate was tested against the ciphertext to identify the correct plaintext.

Solver Script

```
from Crypto.Cipher import AES
from itertools import product

leaks = {
    2: 163378902990129536295589118329764595602,
    4: 303499033263465715696839767032360064630,
    8: 183552667878302390742187834892988820241,
    16: 206844958160238142919064580247611979450,
    32: 230156190944614555973250270591375837085,
    64: 105702179473185502572235663113526159091,
}

ciphertext = bytes.fromhex("477eb79b46ef667f16ddd94ca933c7c0")
N = 128

equations = []
for k, v in leaks.items():
    for i in range(N):
        equations.append(((1 << i) ^ (1 << ((i + k) % N)), (v >> i) & 1))

rows = equations[:]
pivot = {}
r = 0

for col in range(N):
    for i in range(r, len(rows)):
        if (rows[i][0] >> col) & 1:
            rows[r], rows[i] = rows[i], rows[r]
            break
    else:
        continue

    m, b = rows[r]
    for i in range(len(rows)):
        if i != r and ((rows[i][0] >> col) & 1):
            rows[i] = (rows[i][0] ^ m, rows[i][1] ^ b)

    pivot[col] = r
    r += 1

free_vars = [i for i in range(N) if i not in pivot]
for fb in product([0, 1], repeat=2):
    S = [0] * N
    for i, b in zip(free_vars, fb):
        S[i] = b

    for col, ri in pivot.items():
        m, b = rows[ri]
        val = b
        for j in free_vars:
            if (m >> j) & 1:
                val ^= S[j]
        S[col] = val

    key = sum(S[i] << i for i in range(N))

    for endian in ("big", "little"):
        key_bytes = key.to_bytes(16, endian)
        plaintext = bytes(a ^ b for a, b in zip(ciphertext, key_bytes))

        if all(32 <= c < 127 for c in plaintext):
            print(plaintext)
```

Code Explanation

The script converts each leaked diagnostic value into linear equations over GF(2). Gaussian elimination is used to solve the system and recover most of the internal state. The two remaining unknown bits are brute-forced. Each reconstructed state is tested by XORing it with the ciphertext. Only one candidate produces readable ASCII output.

Recovered Flag

The correct reconstruction of the internal state reveals the following flag:

p_ctf{l1nyrl34k}

Conclusion

This challenge highlights how leaking linear relations of a secret state can completely undermine cryptographic security. Even a strong primitive becomes insecure when diagnostic data is improperly exposed.