strange_chain

这是仓促之中出的一道题，欠缺打磨。由于时间紧迫（ 太菜了 ），题目在混淆等各方面降低了不少难度，~~之后应该会出 2.0 版本？~~

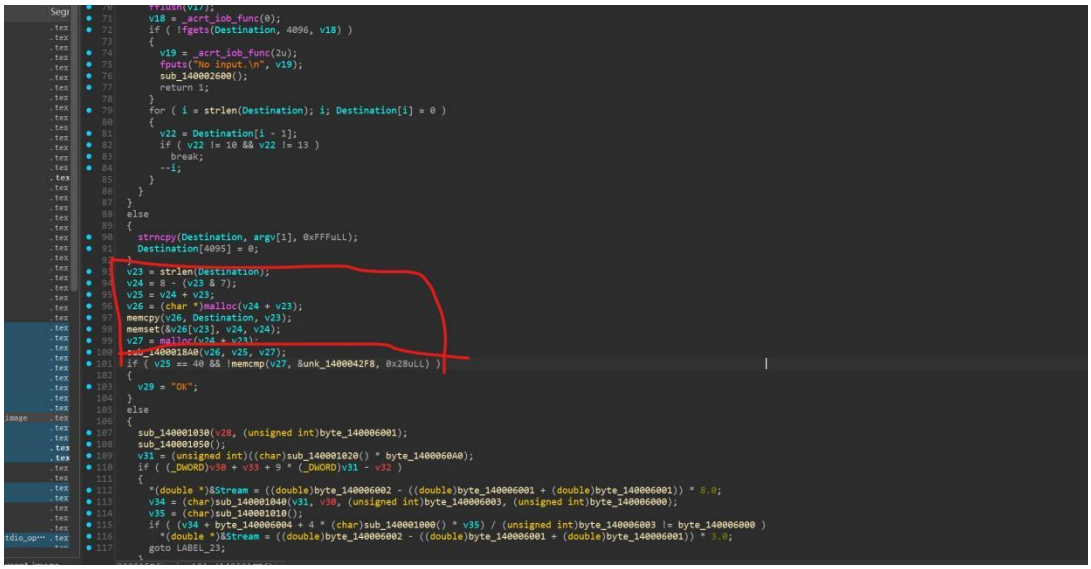
这题目的出题思路大概是把 魔改的rc5 拆成许多块状分区，使用 smc 来修改控制流，然后加上反调试与混淆。但是由于原先准备使用的混淆没有研究明白，为了及时上题，全部内容除了加密方式没变，其他内容全部进行了简单化。~~之后会补上的~~

第一次给校赛以外的比赛出题，水平有限，不足之处，恳请指正。

⊂(o･㋮･o）⊃

正式开始这道题：

ida 打开，函数还是比较清晰的，混淆不多，能看到很明显的对输入的 PKCS#7 padding 处理。



在 sub_140002660 函数里，能看到明显的结构体和链表的构建以及反调试和 smc 的存在。

sub_1400018A0 里就是具体的加密过程了。

逻辑是很明了的，思路也应该挺明确的，恢复出来控制流就行了。

反调试找调用 patch 掉就行，或者 scallyhide 应该也能过。

直接看流程：

注意到在 sub_140001840 里，这里分配了 24 * 函数数的空间，直接调试看:

```
void *__fastcall sub_140001840(__int64 a1)
{
  size_t v1; // rdi
  void *result; // rax
  void *v3; // rbx
  v1 = 24 * a1;
  result = VirtualAlloc(0LL, 24 * a1, 0x3000u, 0x40u);
  v3 = result;
  if ( result )
  {
    memset(result, 144, v1);
    return v3;
  }
  return result;
}
```

能看到这个函数构造了如是

```
mov rax, <func_ptr>
call rax
mov rax, <next>
jmp rax
```

的结构，所以写个脚本 trace 脚本，抓出来就行，然后就能得到程序的控制流了。

然后就能复原了，其实所有的数据都是可以 dump 下来的，比如：

薅的别人的脚本：

```
import ida_name
import ida_bytes
import ida_kernwin
import ida_idaapi
def get_ea2(name: str) -> int:
    ea = ida_name.get_name_ea(ida_idaapi.BADADDR, name)if ea
    raise RuntimeError(f"????? {name}")
    return ea
ADDR RC = get_ea2("dword_7FF7F3436148")
ADDR K = = get_ea2("xmmword_7FF7F34360B0")
```

```
RC = [ida_bytes.get_dword(ADDR_RC + 4*i) for i in range(14)]K  = [ida_
print(K)
```

类似这样的，剩下只逆加密部分就想 …

这题只要会动调，会写 dump 脚本就能出。~~不会，能想到问 ai 也能秒~~

```
import ctypes

class Context:
    def __init__(self):
        self.S = [] # 2*(ROUNDS+1) 个uint32
        self.L = [0]*4  # 4个uint32
        self.A = 0
        self.B = 0
        self.i = 0
        self.j = 0
        self.t = 0
        self.c = 4
        self.kpos = 0
        self.spos = 0
        self.rc = []  # ROUNDS+2 个uint32

        self.EA = 0
        self.EB = 0
        self.round = 0
        self.blk_in = None
        self.blk_out = None

        self.page = None
        self.nstubs = 0
        self.pc = 0

ROUNDS = 12
KEYLEN = 16
WORDMASK = 0xFFFFFFFF
key16 = [0x02,0x00,0x02,0x05,0x01,0x01,0x04,0x05,
         0x01,0x04,0x0A,0x0B,0x0C,0x0D,0x0E,0x0F]
target_ct = [0x08,0x21,0x43,0x56,0x3d,0x50,0xd0,0x6d,0xf2,0x07,0xee,0x
             0x73,0xc4,0x92,0x5d,0xf8,0xe3,0xb9,0x16,0x78,0xcc,0xf0,0x
             0xef,0x5d,0x5b,0xec,0xc6,0xf2,0x63,0x9b]

def rol32(x, r):
    r &= 31
    return ((x << r) | (x >> (32 - r))) & WORDMASK
```

```
def ror32(x, r):
    r &= 31
    return ((x >> r) | (x << (32 - r))) & WORDMASK

def Fmix(x):
    return rol32(x, 5) ^ rol32(x, 13)

def mb_ks_L_acc_byte(ctx):
    k = ctx.kpos
    ctx.kpos += 1
    idx = k >> 2  # k//4
    ctx.L[idx] = (ctx.L[idx] << 8) | key16[k]
    ctx.L[idx] &= WORDMASK

def mb_ks_S_seed0(ctx):
    ctx.t = 2 * (ROUNDS + 1)
    ctx.c = 4
    ctx.S = [0] * ctx.t
    ctx.S[0] = (0xB7E15163 ^ 0xC3EFE9DB) & WORDMASK
    ctx.spos = 1

def mb_ks_S_seed_next(ctx):
    Qr = rol32(0x9E3779B9, 11)
    ctx.S[ctx.spos] = (ctx.S[ctx.spos - 1] + Qr) & WORDMASK
    ctx.spos += 1

def mb_ks_mix_A(ctx):
    ctx.A = rol32((ctx.S[ctx.i] + ctx.A + ctx.B) & WORDMASK, 3)
    ctx.S[ctx.i] = ctx.A

def mb_ks_mix_B(ctx):
    r = ((ctx.A ^ ctx.B) & 31) ^ (ctx.rc[ctx.i % (ROUNDS + 2)] & 31)
    ctx.B = rol32((ctx.L[ctx.j] + ctx.A + ctx.B) & WORDMASK, r)
    ctx.L[ctx.j] = ctx.B

def mb_ks_mix_inc(ctx):
    ctx.i = (ctx.i + 1) % ctx.t
    ctx.j = (ctx.j + 1) % ctx.c

def mb_en_preA(ctx):
    ctx.EA = (ctx.EA + (ctx.S[0] ^ ctx.rc[0])) & WORDMASK

def mb_en_preB(ctx):
    ctx.EB = (ctx.EB + (ctx.S[1] ^ ctx.rc[0])) & WORDMASK
    ctx.round = 1
```

```python
def mb_en_Axor(ctx):
    ctx.EA ^= (ctx.EB + Fmix(ctx.EB)) & WORDMASK

def mb_en_Arol(ctx):
    r = (ctx.EB & 31) ^ (ctx.rc[ctx.round] & 31)
    ctx.EA = rol32(ctx.EA, r)

def mb_en_Aadd(ctx):
    ctx.EA = (ctx.EA + ctx.S[2 * ctx.round]) & WORDMASK

def mb_en_Bxor(ctx):
    ctx.EB ^= (ctx.EA + Fmix(ctx.EA)) & WORDMASK

def mb_en_Brol(ctx):
    r = (ctx.EA & 31) ^ (ctx.rc[ctx.round] & 31)
    ctx.EB = rol32(ctx.EB, r)

def mb_en_Badd(ctx):
    ctx.EB = (ctx.EB + ctx.S[2 * ctx.round + 1]) & WORDMASK

def mb_en_swap3(ctx):
    if (ctx.round % 3) == 0:
        ctx.EA, ctx.EB = ctx.EB, ctx.EA

def mb_en_inc_round(ctx):
    ctx.round += 1

def mb_en_store(ctx):
    o = ctx.blk_out
    o[0] = ctx.EA & 0xFF
    o[1] = (ctx.EA >> 8) & 0xFF
    o[2] = (ctx.EA >> 16) & 0xFF
    o[3] = (ctx.EA >> 24) & 0xFF
    o[4] = ctx.EB & 0xFF
    o[5] = (ctx.EB >> 8) & 0xFF
    o[6] = (ctx.EB >> 16) & 0xFF
    o[7] = (ctx.EB >> 24) & 0xFF


def build_targets_KS_correct():
    """构建正确的KS函数链"""
    targets = []
    for _ in range(KEYLEN):
        targets.append(mb_ks_L_acc_byte)
    targets.append(mb_ks_S_seed0)
    for _ in range(1, 2*(ROUNDS+1)):
        targets.append(mb_ks_S_seed_next)
```

```
    t = 2*(ROUNDS+1)
    c = 4
    mix = 3 * max(t, c)
    for _ in range(mix):
        targets.append(mb_ks_mix_A)
        targets.append(mb_ks_mix_B)
        targets.append(mb_ks_mix_inc)
    return targets

def build_targets_ENC_correct():
    targets = []
    targets.append(mb_en_preA)
    targets.append(mb_en_preB)
    for _ in range(1, ROUNDS+1):
        targets.append(mb_en_Axor)
        targets.append(mb_en_Arol)
        targets.append(mb_en_Aadd)
        targets.append(mb_en_Bxor)
        targets.append(mb_en_Brol)
        targets.append(mb_en_Badd)
        targets.append(mb_en_swap3)
        targets.append(mb_en_inc_round)
    targets.append(mb_en_store)
    return targets

def derive_round_const(k):
    s = 0x7F4A7C15
    for i in range(KEYLEN):
        s = (s ^ k[i]) + 0x9E37 + (s << 5)
        s &= WORDMASK
    rc = [0]*(ROUNDS+2)
    for i in range(ROUNDS+2):
        s ^= s << 13
        s ^= s >> 17
        s ^= s << 5
        rc[i] = rol32(0x9E3779B9 ^ s, i & 31)
        rc[i] &= WORDMASK
    return rc

def rc5v_key_schedule_via_chain(ctx):
    ctx.rc = derive_round_const(key16)
    ctx.L = [0]*4
    ctx.A = ctx.B = 0
    ctx.i = ctx.j = 0
    ctx.kpos = ctx.spos = 0

    ks_chain = build_targets_KS_correct()
```

```
    for func in ks_chain:
        func(ctx)

def reverse_mb_en_store(ctx, ct_block):
    ctx.EA = (ct_block[0] | (ct_block[1] << 8) | (ct_block[2] << 16) |
    ctx.EB = (ct_block[4] | (ct_block[5] << 8) | (ct_block[6] << 16) |

def reverse_mb_en_inc_round(ctx):
    ctx.round -= 1

def reverse_mb_en_swap3(ctx):
    if (ctx.round % 3) == 0:
        ctx.EA, ctx.EB = ctx.EB, ctx.EA

def reverse_mb_en_Badd(ctx):
    ctx.EB = (ctx.EB - ctx.S[2 * ctx.round + 1]) & WORDMASK

def reverse_mb_en_Brol(ctx):
    r = (ctx.EA & 31) ^ (ctx.rc[ctx.round] & 31)
    ctx.EB = ror32(ctx.EB, r)

def reverse_mb_en_Bxor(ctx):
    ctx.EB ^= (ctx.EA + Fmix(ctx.EA)) & WORDMASK

def reverse_mb_en_Aadd(ctx):
    ctx.EA = (ctx.EA - ctx.S[2 * ctx.round]) & WORDMASK

def reverse_mb_en_Arol(ctx):
    r = (ctx.EB & 31) ^ (ctx.rc[ctx.round] & 31)
    ctx.EA = ror32(ctx.EA, r)

def reverse_mb_en_Axor(ctx):
    ctx.EA ^= (ctx.EB + Fmix(EB)) & WORDMASK

def reverse_mb_en_preB(ctx):
    ctx.EB = (ctx.EB - (ctx.S[1] ^ ctx.rc[0])) & WORDMASK

def reverse_mb_en_preA(ctx):
    ctx.EA = (ctx.EA - (ctx.S[0] ^ ctx.rc[0])) & WORDMASK

def build_reverse_ENC_chain():
    reverse_chain = []
    reverse_chain.append(reverse_mb_en_store)
    for _ in range(ROUNDS):
        reverse_chain.append(reverse_mb_en_inc_round)
        reverse_chain.append(reverse_mb_en_swap3)
        reverse_chain.append(reverse_mb_en_Badd)
```

```
            reverse_chain.append(reverse_mb_en_Brol)
            reverse_chain.append(reverse_mb_en_Bxor)
            reverse_chain.append(reverse_mb_en_Aadd)
            reverse_chain.append(reverse_mb_en_Arol)
            reverse_chain.append(reverse_mb_en_Axor)
        reverse_chain.append(reverse_mb_en_preB)
        reverse_chain.append(reverse_mb_en_preA)
        return reverse_chain

    def rc5v_decrypt_block_via_chain(ctx, ct_block):
        reverse_enc_chain = build_reverse_ENC_chain()
        for idx, func in enumerate(reverse_enc_chain):
            if idx == 0:
                func(ctx, ct_block)
            else:
                func(ctx)

        pt_block = bytearray(8)
        pt_block[0] = ctx.EA & 0xFF
        pt_block[1] = (ctx.EA >> 8) & 0xFF
        pt_block[2] = (ctx.EA >> 16) & 0xFF
        pt_block[3] = (ctx.EA >> 24) & 0xFF
        pt_block[4] = ctx.EB & 0xFF
        pt_block[5] = (ctx.EB >> 8) & 0xFF
        pt_block[6] = (ctx.EB >> 16) & 0xFF
        pt_block[7] = (ctx.EB >> 24) & 0xFF
        return pt_block

    def pkcs7_unpad(data):
        pad_len = data[-1]
        if 1 <= pad_len <= 8:
            for i in range(1, pad_len+1):
                if data[-i] != pad_len:
                    return data
            return data[:-pad_len]
        return data

    def main():
        ctx = Context()
        rc5v_key_schedule_via_chain(ctx)
        plaintext = bytearray()
        for i in range(0, len(target_ct), 8):
            ct_block = target_ct[i:i+8]
            pt_block = rc5v_decrypt_block_via_chain(ctx, ct_block)
            plaintext.extend(pt_block)
```

```
plaintext_unpadded = pkcs7_unpad(plaintext) print(f"falg:
{plaintext unpadded.decode('ascii')}")
```

VCTF{Have1ng_a_goOd_t1me_with_SMC}