关键点就是在 native 层实现的 so 加载

```
char *v8; // x0
char *v9; // x20
int v10; // w22
__int64 result; // x0
size_t v12; // x9
__int64 v13; // x11
FILE *v15; // x0
FILE *v16; // x21
size_t v17; // x22
void *v18; // x0
char v19[256]; // [xsp+0h] [xbp-120h] BYREF
__int128 v20; // [xsp+100h] [xbp-20h] BYREF
__int64 v21; // [xsp+118h] [xbp-8h]

v21 = *(_QWORD *)(_ReadStatusReg(ARM64_SYSREG(3, 3, 13, 0, 2)) + 40);
v2 = AAssetManager_fromJava(a1, a2);
v3 = AAssetManager_open(v2, "qwqer1", 3);
Length = AAsset_getLength(v3);
if ( Length >= 1
  && (v5 = Length, (Length & 0xF) == 0)
  && (Buffer = AAsset_getBuffer(v3)) != 0LL
  && (v7 = Buffer, (v8 = (char *)malloc(v5)) != 0LL) )
{
  v9 = v8;
  v20 = *(_OWORD *)"p0l1st";
  v10 = sub_14DC(&v20, 16LL, v7, v8, (unsigned int)v5);
  AAsset_close(v3);
  if ( v10 )
    goto LABEL_6;
```

So 自加载是一个 aes 魔改（可以不逆）

魔改就在这里在

```
v66.n64_u64[0] = vceqd_s64(v12 & 1, 0LL);
v67.n64_u64[1] = v77.n64_u32[1];
v9.n64_u64[0] = vzip1_s8(vceq_s32(v9, v49), (int8x8_t)v30.n128_u64[0]).n64_u64[0];
v67.n64_u32[0] = v51 & 0x40;
v68.n64_u64[0] = vceq_s32(v67, v49).n64_u64[0];
v69.n64_u64[0] = vbic_s8(v41, v66).n64_u64[0];
v70.n64_u64[0] = veor_s8(
                   veor_s8(vbsl_s8(v55, (int8x8_t)0x66006600660066LL, v34), vbic_s8(v78, v56)),
                   vbic_s8(v77, v57)).n64_u64[0];
v30.n128_u64[0] = vshl_n_s16(vzip1_s8((int8x8_t)v30.n128_u64[0], (int8x8_t)v30.n128_u64[0]), 8uLL).n64_u64[0];
v71.n64_u64[0] = vshr_n_s16(vshl_n_s16(v61, 8uLL), 8uLL).n64_u64[0];
v72.n64_u64[0] = vzip1_s8(v64, (int8x8_t)v30.n128_u64[0]).n64_u64[0];
v73.n64_u64[0] = vzip1_s8(v65, (int8x8_t)v30.n128_u64[0]).n64_u64[0];
v74.n64_u64[0] = veor_s8(vand_s8(v38, v66), v69).n64_u64[0];
v69.n64_u32[0] = v51;
```

我们可以通过 frida hook 的方式来获取加密的 so

或者

这个 so 加载是实现的落地加载，我们可以 nop 掉

Unlink 然后运行程序就能在程序目录里面得到 so 文件了

```
{
  off_40A8 = dlsym(v18, "Java_com_qwq_myapplication
  if ( off_40A8 )
  {
    unlink(v19);
    result = 1LL;
    byte_40B0 = 1;
    return result;
  }
  dlclose((void *)qword_40A0);
  qword_40A0 = 0LL;
}
}
unlink(v19);
}
}
else
```

```
:/data/data/com.qwq.myapplication # ls
cache  code_cache  files  validator_decrypted_3444.so
:/data/data/com.qwq.myapplication #
```

```
14  v15 = *(_QWORD *)(_ReadStatusReg(ARM64_SYSREG(3, 3, 13, 0, 2)) + 40);
15  v5 = (_BYTE *)(*(__int64 (__fastcall **)(__int64, __int64, _QWORD))(*(_QWORD *)a1 + 1352LL))(a1, a3, 0LL);
16  v6 = v5;
17  if ( *v5 )
18  {
19    sub_92C(v5, &unk_670, v14, &v12);
20    memset(s1, 0, sizeof(s1));
21    if ( v12 >= 1 )
22    {
23      v7 = s1;
24      v8 = v14;
25      if ( 2 * v12 <= 1 )
26        v9 = 1LL;
27      else
28        v9 = (unsigned int)(2 * v12);
29      do
30      {
31        v8 += 4;
32        v7 = (_QWORD *)((char *)v7 + (int)sub_888(v7, -1LL));
33        --v9;
34      }
35      while ( v9 );
36    }
37    v10 = strcmp(
38            (const char *)s1,
39            "c2f6aade913f38fa407aceea7679a9db017ea632769a2c530f9917cbf2388a8dc89f80174a0d961820ebf4b20328e9ae") == 0;
40    (*(void (__fastcall **)(__int64, __int64, _BYTE *))(*(_QWORD *)a1 + 1360LL))(a1, a3, v6);
41  }
42  else
43  {
44    (*(void (__fastcall **)(__int64, __int64, _BYTE *))(*(_QWORD *)a1 + 1360LL))(a1, a3, v5);
45    return 0;
46  }
47  return v10;
48}
00000AF8 Java_com_qwq_myapplication_MainActivity_checkFlag:11 (AF8)
```

```
  result = (unsigned int)a2[2];
  v28 = 32;
  v29 = -1640531527;
  do
  {
    --v28;
    v27 += (*a2 + 16 * v26) ^ (v26 + v29) ^ (a2[1] + (v26 >> 5));
    v30 = v29 + v27;
    v29 -= 1640531527;
    v26 += (result + 16 * v27) ^ v30 ^ (a2[3] + (v27 >> 5));
  }
  while ( v28 );
  v31 = 8 * v23++;
  v25[1] = v26;
  *(_DWORD *)&v32[v31] = v27;
  *(_DWORD *)(a3 + v31) = v27;
  *(_DWORD *)(a3 + (v31 | 4)) = v26;
}
while ( v23 != v24 );
return result;
}
```

真正的 check 就是个 tea

直接写脚本就行

```
#include <stdint.h>
#include <string.h>
#include <stdio.h>

/* TEA 解密函数 */
void tea_decrypt(uint32_t v[2], const uint32_t k[4]) {
    uint32_t v0 = v[0], v1 = v[1], sum = 0xC6EF3720; // delta * 32
    const uint32_t delta = 0x9e3779b9;

    for (int i = 0; i < 32; i++) {
        v1 -= ((v0 << 4) + k[2]) ^ (v0 + sum) ^ ((v0 >> 5) + k[3]);
        v0 -= ((v1 << 4) + k[0]) ^ (v1 + sum) ^ ((v1 >> 5) + k[1]);
```

```
        sum -= delta;
    }

    v[0] = v0; v[1] = v1;
}

/* 解密字符串 */
void decrypt_string(uint32_t* blocks, int block_count, uint32_t key[4], char* output) {
    uint32_t temp_blocks[16];

    // 复制数据到临时数组
    for (int i = 0; i < block_count * 2; i++) {
        temp_blocks[i] = blocks[i];
    }

    // 对每个 64 位块进行解密
    for (int i = 0; i < block_count; i++) {
        tea_decrypt(&temp_blocks[i * 2], key);
    }

    // 将解密后的数据复制到输出字符串
    memcpy(output, temp_blocks, block_count * 8);
    output[block_count * 8] = '\0'; // 添加字符串结束符
}

int main() {
    uint32_t key[4] = {0xa56babcd, 0x1f1234f1, 0x12345678, 0xfedcba98};

    // 加密的数据块
    uint32_t encrypted_blocks[] = {
            0xc2f6aade,
            0x913f38fa,
            0x407aceea,
            0x7679a9db,
            0x017ea632,
            0x769a2c53,
            0x0f9917cb,
            0xf2388a8d,
            0xc89f8017,
            0x4a0d9618,
            0x20ebf4b2,
            0x0328e9ae
    };
```

```
    int block_count = sizeof(encrypted_blocks) / sizeof(encrypted_blocks[0]) / 2;



    // 解密
    char decrypted[128];
    decrypt_string(encrypted_blocks, block_count, key, decrypted);

    printf("Decrypted string: %s\n", decrypted);

    return 0;
}
```