CatCTF 2023 Reverse WriteUp(Yellowbest)

高铁小姐的IDA

运行附件后发现输出了 `flag` 。

但将上述 `flag` 提交平台显示错误，说明输出的为 `fake flag` ,将附件放入IDA中查看，可以发现出现了两个 `flag` ，则推断其中另一个为 `true flag`

P.S. `Fake flag` 应该提示的更加明显，如 `flag{this_is_a_fake_flag}`

高铁小姐的迷宫

运行附件，发现其为输入一条路径。将附件放入IDA中查看，发现 `main` 函数的主要逻辑是根据输入进行一个判断，判断正确则返回 `flag` ，发现 `show_flag` 函数传入的参数是我们所需要输入的路径，因此本题只能从 `try_solve_maze` 判断函数下手:从返回条件 `return v6 == 7 && v7 == 7;` 以及初始化部分 `v6 = 0; v7 = 0;` ，再联想到本题的名称是迷宫，可判断出该迷宫的起始位置为 `(0, 0)` ，终点位置为 `(7, 7)` 。观察循环判断的主体部分，发现其主要判断语句为：

```
1 if ( v7 < 0 || v7 > 7 || v6 < 0 || v6 > 7 || (v8 >> (8 * (unsigned __int8)v7 + 7
2       return 0i64;
```

不难看出 `v6` 、 `v7` 表示坐标，而这迷宫的大小必然为 `8 * 8` 。观察后续的位运算，发现其比对 `ui64` 整数中对应 `(8 * i + 7 - j)` 的位置是否为 `1` ，由此可推断 `ui64` 整数的每一位均表示了迷宫指定位置的情况， `0` 表示可通行， `1` 则表示不可通行。同时，也能判断出，迷宫在该整数中存储的结构为最高的 `8` 字节存放第 `0` 行，最低的 `8` 字节存放第 `7` 行，每一行从高到低依序表示位置。根据传入的参数 `0x4612DA485B1AD218ui64` 可以推理出迷宫如下所示：

```
1  ooo##ooo
2  ##o#oo#o
3  ooo##ooo
4  o#o##o##
5  o#oo#ooo
6  ##o##o#o
7  ooo#oo#o
8  o#ooo##o
```

根据后续的判断语句可以看出移动方向分别用 `w`、`a`、`s`、`d` 表出，由此可得到对应的正确路径，输入路径即可得到 `flag`。

高铁小姐的压缩器

使用查壳软件发现附件加了 `upx` 的壳，能识别出来表示大概率没有魔改。使用 `upx -d` 对附件进行 `unpack`.将解压之后的附件放入IDA中查看。打开 `Strings subview`，发现存在提示字符，根据提示字符的引用情况可找到对应的 `main` 函数。

发现函数主体主要进行对输入的一个 `check`， `check` 之前对输入的字符串进行了处理。

```
Str2 = (char *)sub_1400015E2(Dst, v10);
if ( !strcmp(Str1, Str2) )
   sub_140001591("Correct!");
else
   sub_140001591("Incorrect!");
```

处理函数可以看出为分组处理，且每组为 `4` 个字符，推断为 `base64` 的一种魔改，查看 `byte14001A020`，发现为正常的base64字符表。观察到处理函数中，对于每组字符，根据组的序号 `i` 对每组字符进行了向后变动 `i` 位。可编写下列 `decode` 脚本：

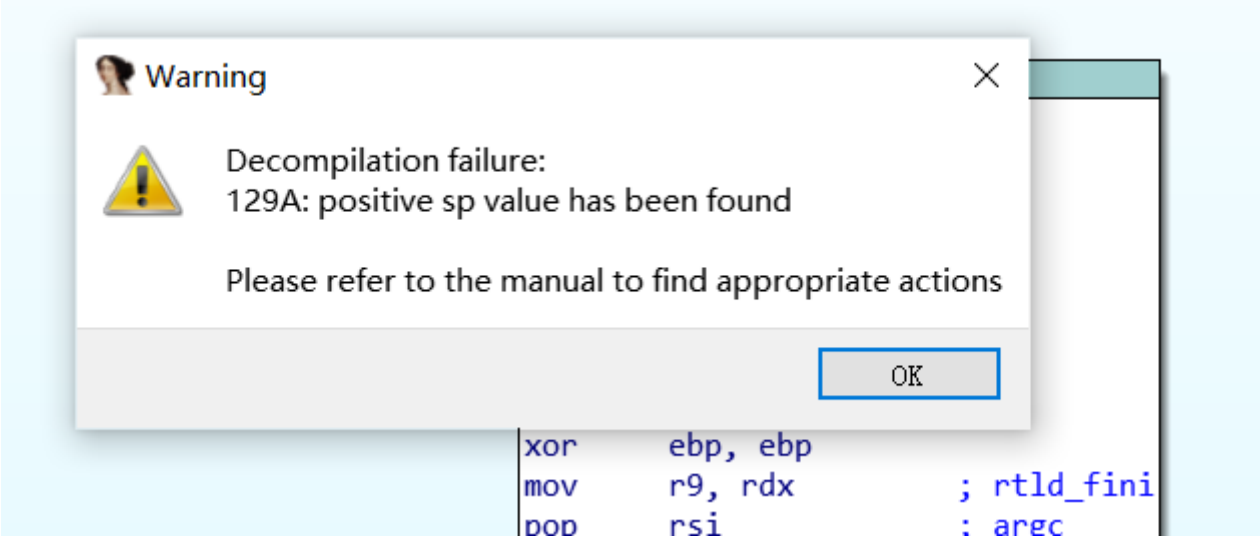
```
int get_location(char ch){
    for(int i = 0; i < 64; i++){
        if(ch == base64_map[i]){
            return i;
        }
    }
    return -1;
}
char *base64_decode(char *cipher, int length)
{

    char *plain = malloc(strlen(cipher) * 3 / 4);
    int i = 0;

    for (i = 0; i < length / 4; i++)
    {
        plain[3 * i]     = (((get_location(cipher[4 * i]) + 64 - i) % 64) << 2)
        plain[3 * i + 1] = (((get_location(cipher[4 * i + 1]) + 64 - i) % 64) <<
        plain[3 * i + 2] = (((get_location(cipher[4 * i + 2]) + 64 - i) % 64) <<
    }
    plain[strlen(cipher) * 3 / 4 - 1] = '\0';
    printf("%s", plain);
```

```
23      return plain;
24  }
```

输入 `main` 中作为基准比对的那串字符即可得到 `flag`。

高铁小姐的数码城

将附件放入IDA中，进行反编译时，发现无法正常反编译，推测可能存在反静态调试的技术。



发现 `start` 函数之前，调用了 `sub_1210`，查看函数 `sub_1210` ，发现其对地址位于 `0x1070 ~ 0x1210` 之间的部分进行了如下操作：

```
1     do
2     {
3       v1 = *(_DWORD *)v0;
4       v0 = (int (__cdecl *)(int, char **, char **))((char *)v0 + 4);
5       *((_DWORD *)v0 - 1) = (v1 - 1176221172) ^ 0xF39AB582;
6     }
```

发现每 `4byte` 进行了一次异或操作，可以看出使用了 `smc` 对该区间内的代码进行了保护。使用二进制编辑工具将地址位于 `0x1070 ~ 0x1210` 区域内的数据 `dump` 出来。可编写如下脚本进行解码：

```
1 def read_file_and_decode(file_path):
2     try:
3         with open(file_path, "rb") as file:
4             file_bytes = file.read()
5             result_bytes = []
6             for i in range(0, len(file_bytes), 4):
```

```
                chunk = file_bytes[i:i + 4]
                if len(chunk) == 4:
                    int_value = int.from_bytes(
                        chunk, byteorder='little', signed=False)
                    int_value = (int_value - 1176221172) ^ 0xF39AB582
                    int_value = int_value & 0xFFFFFFFF
                    bytes_value = int_value.to_bytes(4, byteorder='little')
                    result_bytes.extend(bytes_value)
            return result_bytes
    except FileNotFoundError:
        return None
```

将decode之后的数据替换掉原文件中的对应位置的数据，再次放入IDA，发现此时反编译正常，查看main函数，可发现关键部分进行了一次简单的异或加密，可编写如下解题脚本：

```
def decode():
    a1 = -88659107510083859
    a2 = 688237945405651195
    a3 = -2553973195072268547
    a4 = 1249800879

    int_list = []
    result_bytes = []
    int_list.append(a1 & 0xFFFFFFFF)
    int_list.append((a1 >> 32) & 0xFFFFFFFF)
    int_list.append(a2 & 0xFFFFFFFF)
    int_list.append((a2 >> 32) & 0xFFFFFFFF)
    int_list.append(a3 & 0xFFFFFFFF)
    int_list.append((a3 >> 32) & 0xFFFFFFFF)
    int_list.append(a4)
    for item in int_list:
        item = (item ^ 0xF39AB582) + 1176221172
        item = item & 0xFFFFFFFF
        bytes_value = item.to_bytes(4, byteorder='little')
        result_bytes.extend(bytes_value)

    print(bytes(result_bytes).decode('utf-8'))
```

高铁小姐的下午茶

将附件放入IDA中，打开 `Strings subview` ，发现存在提示字符串，根据该提示字符串的引用情况，可以找到 `main` 函数。观察 `main` 函数，发现其也是对输入进行处理后再进行一个简单的匹配。找到关键处理函数 `sub_140026A50` 。

发现该函数执行完之后，调用了另一个函数，再另一个函数中再次调用了另一个函数……且这些函数的主体部分大致相同，猜测可能将循环展开成函数进行了简单混淆，共计32个函数。其中，调用的第一个函数如下：

```
void __fastcall sub_140026A50(_DWORD *a1, _DWORD *a2, _DWORD *a3)
{
  _DWORD *v3; // [rsp+48h] [rbp+18h]

  v3 = a2;
  *a1 += (*v3 - 1640531527) ^ (*a3 + 16 * *v3) ^ ((*v3 >> 5) + a3[1]);
  *v3 += (*a1 - 1640531527) ^ (a3[2] + 16 * *a1) ^ ((*a1 >> 5) + a3[3]);
  sub_140026980(a1, a2, a3);
}
```

可以发现该函数对 `a1` 和 `a2` 为一组，并与 `a3` 进行了一系列的加和异或操作，发现关键整数 `-1640531527` 即 `0x9E3779B9`，猜测为采用了 `TEA` 加密，加密轮次为 `32`，并通过传入的 `a3` 可以找到对应加密的 `key` 值数组。可以编写出如下 `Decrypt` 函数：

```
void Decrypt(uint32_t *Data, uint32_t *Key)
{
    uint32_t x = Data[0];
    uint32_t y = Data[1];

    uint32_t sum = 0;
    uint32_t delta = 0x9E3779B9;
    sum = delta << 5;
    for (int i = 0; i < 32; i++)
    {
        y -= ((x << 4) + Key[2]) ^ (x + sum) ^ ((x >> 5) + Key[3]);
        x -= ((y << 4) + Key[0]) ^ (y + sum) ^ ((y >> 5) + Key[1]);
        sum -= delta;
    }
    Data[0] = x;
    Data[1] = y;
}
```

通过 `check` 函数 `sub_140015F5` 可以找到加密之后的数据，可编写如下解题脚本：

```
int main()
{

```

```
    uint32_t Data[34] = {4280888139, 2037429180, 3281899677, 1294517364, 1473761
                         3643456943, 2815164043, 429546175, 4253649492, 37046028
                         2961196347, 696785352, 1444447161, 1929565336, 33395647
                         4251074430, 2218216128, 3816058564, 2576672822, 3989319
                         1294679424, 1295458946};

    uint32_t key[4]   = {268468224, 268550144, 268492800, 268460032};

    printf("value after decrypt : ");
    for (int i = 0; i < 17; i++)
    {
        Decrypt(Data + 2 * i, key);
        printf("%u %u ", Data[2 * i], Data[2 * i + 1]);
    }
    printf("\n");

    for(int i = 0; i < 34; i++){
        printf("%c", Data[i]);
    }
    printf("\n");

    return 0;
}
```

高铁小姐的Obfuscation Beginner Challenge

此题为CFF混淆，但并未使用 `ollvm` 工具进行混淆，写了个简单的 `code_generator` 和 `obfuscator` ，因此没有工具可以进行使用，只能通过手动来进行翻译。

source.cpp 如下：

```
#include <cstdio>
#include <cstdint>
#include <cstring>

int main()
{
    char flag[40] = {'\0'};
    uint32_t hash_flag[9] = {4249032922, 2873892813, 3395819401,
                             3238152401, 4251462384, 2926067706,
                             2875334543, 2942515706, 3809875849};
    uint32_t i, j, x, base;
    uint32_t xor_base = 0x9E3779B9;

    printf("Please input your flag: ");
```

```
    scanf("%39s", &flag);
    if (strlen(flag) != 36)
    {
        printf("Incorrect!\n");
        return 0;
    }

    for (i = 0; i < 9; i++)
    {
        x = 0;
        base = 1;
        for (j = 0; j < 4; j++)
        {
            x += flag[4 * i + j] * base;
            base = base * 256;
        }
        x ^= xor_base;
        if (x != hash_flag[i])
        {
            printf("Incorrect!\n");
            return 0;
        }
    }

    printf("Correct!\n");
}
```