**SMC技术**

SMC，即Self Modifying Code，动态代码加密技术，指通过修改代码或数据，阻止别人直接静态 分析，然

后在动态运行程序时对代码进行解密，达到程序正常运行的效果。

VirtualProtect 函数通常用于代码自加密的场景。代码自加密是一种保护代码不被轻易逆向分析 的技术， 通过在程序运行时动态地修改代码的内存保护属性，使得代码在执行时可以被修改和执行，但在不执 行时则

不能被读取或修改。

在linux系统中，可以通过mprotect函数修改目标内存的权限 在Windows系统中，VirtualProtect函数实现内存权限的修改 因此也可以观察是否有这俩个函数来判断是否进行了SMC

SMC一般有俩种破解方法，第一种是找到对代码或数据加密的函数后通过idapython写解密脚本。第 二种是动

态调试到SMC解密结束的地方dump出来。

**堆栈平衡**

// 堆栈不平衡问题出现原因 one：一般是程序代码有一些干扰代码，让IDA的反汇编分析出现错误。比如用push + n条指令 +

retn来 实际跳转，而IDA会以为retn是函数要结束，结果它分析后发现调用栈不平衡，因此就提示sp analysis failed.

two：还有一些比如编译器优化，因为ida是用retn指令来识别函数结束的，如果函数不是以这种 方式结束，

IDA就会分析为栈不平衡。也就是IDA找不到函数结束的位置。

// 堆栈平衡原理 1、概念解释

函数返回时，堆栈需要恢复到调用前的状态 2、平衡的重要性

避免内存访问错误和程序崩溃 保证程序的正常执行逻辑

3、平栈的方式 外平栈（cdecl）

在 cdecl 调用约定中，参数也是从右往左一次压入堆栈，但堆栈平衡由调用者在函数返回后负 责清理

内平栈（stdcall）

stdcall 是内平栈，既有被调用函数在返回前负责清理堆栈，通过调整栈指针来移除压入栈中 的参数

**下面简单举个例子，让大家更深刻的了解堆栈平衡的概念**

[汇编中的函数调用中栈的工作过程](https://www.bilibili.com/video/BV1Xt411x7Dr/?share_source=copy_web&vd_source=cb1799ab55ca5d1f2c72c3c2c11d5e74)

// 下面以汇编的角度来解释函数调用的过程，这里就以上面视频中的例子为例 A\_Func(5,6); // 这里main函数要调用A\_Func函数，参数为5和6，此时假设esp的值为

0x1000

push 6 // 参数先入栈，顺序为从右往左

push 5 // 此时 esp 因为加入了了两个int类型参数，所以变成了0x1000-8

call A\_Func // call指令做的操作相当于把当前的eip值压入栈中（也就是call指令的下一条 指令地址），并跳转到A\_Func函的入口地址

add esp, 8 // 这一步做的就是平栈操作，把刚才压入栈中的参数弹出，恢复esp的原值，此时 esp的值为0x1000

// 以上就是堆栈平衡的过程，简单来说就是调用函数前后堆栈的状态要保持一致。

// 在调用函数时，会先将参数从右到左依次压入栈中，然后 call 跳转到被调用函数（压入call 指令的

//下一个指令的地址），在被调用函数中在进行 push ebp; 等操作。我们可以 F9 步过call指 令，会发

// 现 ESP又恢复了参数压入之前的数值，这就是堆栈平衡。

**网鼎杯2020jocker**

// 利用Exeinfo查看文件的PE信息，可以知道文件是 32为无壳程序，利用IDA Pro打开 Shift+F12 进入字符串窗口 -> 看到关键字"please input you flag:"，双击进入 -> Ctrl+x查看交叉引用 -> 从而定位到main函数

// 利用IDA打开，出现 positive sp value has been detected, the output may be wrong!

// 出现了栈不平衡的问题，导致ida无法生成代码。 选项 -> 常规 -> 堆栈指针（查看栈帧变换）

可以看到，箭头指的两个地方栈偏移都出了问题，应该在call完之后都会平栈，也就是应该都是 0AC才对，

我们有快捷键 alt+k 将偏移改为0

然后F5返回main函数可以发现报错没了

先简单分析代码，可以知道输入字符串长度为24，这里还有 VirtualProtect 函数，可以猜测用到 了SMC技术， 加密了encrypt()函数，并在倒数第6行动态解密encrypt()函数。中间还有一个将输入的Str字符 串赋值给了

Destination变量。 然后就是，Str经过wrong()函数和omg()函数，接下来我们进入看看，具体做了那些操作。

首先，wrong()函数的主要操作就是，遍历输入的Str字符串，若下标i为奇数，该元素就与下标 做减法。相反

若是偶数，则该元素与下标做异或。 最后omg()函数就是比较，经过处理后的Str字符串与&unk\_4030c0的字符串进行比较，若相同则

输入正确，反 之错误。

但是得出来的flag是错误的，真正的加密在encrypt()函数中，我们需要找到这个函数。但是 encrypt()函数 由于堆栈不平衡反汇编不出来，且是用到SMC技术的，所以我们这里用动态调试的方式来解决这些问 题。

**正片开始**

根据上图下好断点后，进入动态调试之后，随便输入24位字符串aaaaaaaaaaaaaaaaaaaaaaaa。 点击上方的视图 -> 选择反汇编窗口 -> 然后按 F7 单步执行进入encrypt()函数。可以看到红线 下面还有一段

代码它不属于任何函数中，应该就是ida识别错误。

点击\_\_Z7encryptPc 先用U将其设为无定义

再按 c 声明为代码段（对黄色线条之内的数据），点击 F(强制) + yes，最后返回点击 \_\_Z7encryptPc 用 P 将

其设置为函数，然后 F5 / Tab 键来到反汇编界面，encrypted函数就被修复了，最后就是 finally函数，直接对

loc\_40159A按 P 即可

//sub\_40159A：

含义：sub\_40159A 通常表示一个完整的函数（subroutine），位于地址 0x40159A。 命名习惯：sub\_ 前缀加上地址的形式通常用于自动命名 IDA 识别的函数。例如 sub\_40159A 表 示 IDA 认为从

0x40159A 地址开始是一段独立的代码，符合函数的结构。

//loc\_40159A：

含义：loc\_40159A 表示一个代码标签（location），通常用于跳转或引用的位置，位于地址 0x40159A。

命名习惯：loc\_ 前缀加上地址表示 IDA 识别的某个特定位置，作为代码段内的标签或局部跳转目 标。

完成以上步骤，encrypt()函数和finally()函数就被修复了。函数代码如下：

encrypt()函数： 可以看到就是将输入字符串的前19位与Buffer数组进行一对一异或，异或后的结果与v2数组一对一 比较。

也就是说v2数组就是加密后的前19位字符串。

finally()函数： 不知道他在干嘛，但是根据网上资料知道，输入字符串后面5位应该是与某一个随机数进行异或得到 v3。

# 解密脚本

hh = 'hahahaha\_do\_you\_find\_me?'

v2 = [0x0E, 0x0D, 0x9, 0x6, 0x13, 0x5, 0x58, 0x56, 0x3E, 0x6,0x0C, 0x3C, 0x1F, 0x57, 0x14, 0x6B, 0x57, 0x59, 0x0D]

flag = []

for i in range(19): flag.append(chr(v2[i] ^ ord(hh[i])))

v3 = [37, 116, 112, 38, 58] key = ord('}') ^ 58

for i in range(5): flag.append(chr(v3[i] ^ key))

print(''.join(flag))

# flag{d07abccf8a410cb37a}