CTF逆向总结（二）

原创

沐一 · 林 于 2021-10-21 17:45:31 发布 969 收藏 13
分类专栏： 笔记 文章标签： ctf

本文链接：https://blog.csdn.net/xiao__1bai/article/details/120891156

CTF 逆向总结

目录

攻防世界open-source：(argv[]外部调用输入参数)

攻防世界logmein：（地址小端存放与正向）

BUUCTF的reverse2：（原flag简单操作）

攻防世界666：（函数逻辑封装，函数名称暗示）

攻防世界Reversing-x64Elf-100：（函数逻辑封装、地址小端存放与正向、二维数组算法积累）

攻防世界EasyRE：（栈地址连续小字符串变量、栈中过渡变量反序字符串、/x7f截断字符串、运算符优先级注意）

攻防世界re-for-50-plz-50：

攻防世界IgniteMe：（函数逻辑封装、大小写字符转换算法）

攻防世界zorropub：（伪随机数加密算法、md5加密/解密算法、代码截断重写、函数积累、exe爆破传参、遍历字符加2算法积累）

main函数与迷宫结合类型：

攻防世界maze：（高低位分割数、函数逻辑封装、迷宫结合）

攻防世界easy_Maze：（迷宫结合、地址连续小数组、题目描述暗示、环境准备函数、IDA动态调试、GDB动态调试、IDA的Hex View图热键）

main函数与游戏结合类型：

攻防世界gametime：（游戏通关生成flag、）

main函数与数学算法结合：

攻防世界notsequence：（杨辉三角算法、函数逻辑封装、IDA对char型(byte)的4*计数）

攻防世界SignIn：（RSA加密/解密算法、函数积累、字符ASCII码做索引、ASCII码表相关、RSA的ASCII字符整数16进制拆分转换算法）

攻防世界ReverseMe-120：（base64加密/解密算法、可变参数混淆、寄存器传参、函数名称暗示、冗余中锁定关键代码、函数积累、数组首地址变化遍历字符串算法积累）

main函数中嵌入大量冗余代码，拆分代码混淆：

攻防世界Newbie_calculations：（非预期行为、不能直接运行、题目描述暗示、栈地址连续小数组、c语言写脚本、不同系统的特殊数、负数作循环条件）

攻防世界testre：（函数逻辑封装、冗余中锁定关键代码、base58加密算法、）

函数逻辑封装类型：

攻防世界的no-strings-attached：（函数名称暗示，GDB动态调试，小端）

攻防世界answer_to_everything：（函数名称暗示、函数逻辑封装、出人意料的flag、题目描述暗示）

攻防世界secret-galaxy-300：（函数名称暗示、题目描述暗示、字符串拆分算法积累）

攻防世界simple-check-100：（IDA动态调试、GDB动态调试）

攻防世界re1-100：（函数逻辑封装、出人意料的flag、非预期行为）

攻防世界elrond32：（argv[]外部调用输入参数符合条件、函数逻辑封装、递归调用算法）

攻防世界babymips：（多层加密操作、函数逻辑封装、移位算法积累、取限制位数算法、奇数偶数判断算法）

main函数中有与本地文件相关的操作类型：

攻防世界getit：（IDA动态调试、GDB动态调试）

攻防世界key：（不能直接运行、多层交叉引用查看、OD动态调试、寄存器传参、同地址变量、动调验证值猜想、冗余中锁定关键代码、运算符优先级注意、大量判断少赋值的字符串比较算法、不用输入类型）

main函数中有与本地环境变量相关的操作类型：

攻防世界catch-me：（不能直接运行、不用输入类型、main函数主逻辑平铺、冗余中锁定关键代码、运算符优先级注意、IDA动态调试、小端转大端存储算法）

main函数主逻辑分析（C++）

main函数中嵌入大量冗余代码，拆分代码混淆：

攻防世界dmd-50：（函数积累、地址小端存放与正向、md5加密/解密算法、出人意料的flag）

攻防世界crazy：（函数名称暗示、地址赋值算法积累、非预期行为、出人意料的flag）

main函数逻辑平铺：

攻防世界re2-cpp-is-awesome：（字符ASCII码做索引、函数逻辑封装、argv[]外部调用输入参数符合条件、align错误反汇编、模板复制操作）

攻防世界reverse-for-the-holy-grail-350：（冗余中锁定关键代码、模板赋值操作、多层加密、倍数条件算法积累、地址差值操作、超位数循环截取算法、正向爆破）

攻防世界easyCpp：（函数积累、函数逻辑封装、lambda自定义函数、IDA动态调试、动调验证值猜想、斐波那契数列算法）

无main函数分析（C语言）

主逻辑平铺一函数内：

攻防世界Mysterious：（地址小端存放与正向，出人意料的flag）

攻防世界流浪者：（多层交叉引用查看、函数逻辑封装、范围算法积累、函数积累）

攻防世界srm-50：

攻防世界hackme：（可变参数混淆、随机抽取比较、取限制位数算法）

无伪代码分析汇编：

攻防世界之76号：（F7和F8交叉使用、函数逻辑封装、寄存器传参、switch正向代入推导）

带壳题目类型

工具直接脱壳类型：

攻防世界simple-unpack脱壳：（工具脱壳）

攻防世界Windows_Reverse1：（工具脱壳、不能直接运行、寄存器传参、地址差值+数组组合遍历字符串、字符ASCII码做索引、ASCII码表相关）

攻防世界Replace：（工具脱壳、解题逆向模板、>> 和 % 运算符算法积累、正向爆破）

攻防世界Windows_Reverse2：（专业脱壳工具、大小写字符转换算法、16进制转10进制算法、遍历字符加1算法积累、同地址变量、base64加密/解密算法、多层加密操作）

攻防世界easyre-153：（函数积累、线程操作积累、IDA伪代码生成优化）

手动脱壳类型：

攻防世界crackme：（ESP脱壳定律、设立硬件访问断点、OD手动脱壳操作、工具脱壳、导入表修复）

攻防世界BabyXor：（ESP脱壳定律、OD动态调试、OD手动脱壳操作、导入表修复、函数逻辑封装、函数积累、环境准备函数、不用输入类型、正向爆破、地址差值操作、for空执行循环遍历字符串）

花指令题目类型（代码与数据混合）

自定义函数自修改：

攻防世界BABYRE：（函数名称暗示、IDA热键重新反汇编、IDA动态调试、栈地址连续小数组）

2021年10月广东强网杯，REVERSE的simplere：（迷宫结合、base64加密/解密算法、）

2021年9月广州羊城杯，REVERSE的RE-BabySmc：（函数积累、移位算法积累、IDA热键重新反汇编、单层交叉引用查看、base64加密/解密算法、正向爆破）

系统函数函数自修改：（HOOK，通常两次修改系统函数，一次改成自定义机器码，一次改回正常）

攻防世界EASYHOOK：（非预期行为、函数积累、手动机器码）

攻防世界梅津美治郎：（函数积累、非预期行为、环境准备函数、IDA动态调试、int3断点考察）

地址运算动态跳转：

攻防世界serial-150：（IDA热键重新反汇编、nop修补垃圾代码、动态地址运算处理、F7和F8交叉使用）

安卓java类逆向分析

java逻辑平铺：

攻防世界Guess-the-Number：（代码截断重写）

JS类逆向分析

攻防世界secret-string-400：（函数名称暗示、JS控制台操作、涉及虚拟机、字节码相关操作）

python逆向分析

pyc文件反编译：

攻防世界re4-unvm-me：（逆向解题模板、md5加密/解密算法、）

攻防世界handcrafted-pyc：（函数积累、涉及虚拟机、代码截断重写、字节码相关操作）

D语言逆向分析（.d后缀）

攻防世界deedeedee：

RC4解密脚本：

INT3断点：

ESP脱壳定律：

非预期行为：

指解题中出现与预想结果不符合的一系列非预期行为，这基本说明了在中间或前面存在其他自己还没分析的操作。

不同系统的特殊数：

指解题中遇到考察特定位数系统中特定的数的真实值的时候，需要辨认出对应的值才能继续解题。如：32位系统中100000000就是0了

冗余中锁定关键代码：

从后往前看，就是确定比较关键对象，从该对象开始排除其他无关变量，一步步找出与该对象有关的其它变量，最后串起找到的所有相关变量，然后开始逆向分析。

题目类型总结：

题目描述暗示：

指题目给出的描述中有解题的大方向思路，以及对解题过程中出现的一些疑惑点的解释。

不能直接运行：

指解题中下载的附件无法正常运行，这种文件通常不需要用户输入，可能是对外界本机环境又要求，需要文件相关操作等。也可能是脱壳后地址混乱，需要修复导入表或梳理地址，还有可能是算法混淆，增加了运算时间。

不用输入类型：

指解题中遇到程序在没有用户输入情况下直接执行，这种不用输入的程序都是直接利用程序外部的一些东西来作为条件继续执行的，如本地环境，文件读取等。这需要我们修改程序外的一些东西，来符合程序的执行。

游戏通关数生成flag：

指与游戏相关的可执行文件中，不是存储型flag，而是与用户输入相关的生成型flag，且以通关数生成flag。通常这种类型的题目改一下判断条件就可以全部通关获取flag了。

迷宫结合：

指解题过程中有类似于迷宫的每一步不能碰点或每一步必须符合在1维或多为字符串上的落点，如：
*A**********.***.....**.....***.*********..*********B**

这称之为与迷宫结合类型。算法走迷宫过于耗时，通常整理出迷宫维数后手动来走。

涉及虚拟机：

指解题中设计了个虚拟机，虚拟机的指令和数据来自预定义加载的大量字节码或数字。其实现原理是根据不同的字节码转换代码函数，进行特定的操作。通过调试和输出承接转换后代码的变量，可以呈现出代码的最终样式，有了代码就可以进行进一步解题了。

字节码相关操作：

指在涉及自定义虚拟机的题目类型中只能对字节码进行相关操作来把预定义字节码往代码方向进一步转换。

如：通过用bytes函数把字节码直接转换为字节串。因为bytes函数是以字节序列二进制的形式表示字节串，以字节为单位。可能对得上字节码吧。如：print(bytes(字节码列表).split(b'\x00'))

又如：python反编译中要把二进制字节码写入文件中，再通过相关工具反编译才能得到代码。

多层加密操作：

指解题中原代码逻辑中出现不止一层加密操作，多层加密之间可能彼此有特殊关联，逆向时需要反向一层层来解密，理清每一层解密的关系。

线程操作积累：

指解题代码中设计多线程的交叉，阻塞，共享内存等操作，由于线程知识积累较少，所以每次都要积累。

汇编操作类总结：

int3断点考察：

int 3是断点的一种，代码执行到此处会抛出异常，伪代码中通常会有__debugbreak()函数。因为这不是我们在OD之类的调试器下的断点，所以OD之类的调试器不会处理该断点的异常，而是交给系统处理，而系统的处理方式往往是强制退出。所以我们在动态调试中要改为nop，不然后面的代码就没法执行。

手动机器码：

指解题过程中遇到类似自修改代码的操作。

如HOOK原型：

byte_40C9BC = 0351;

dword_40C9BD = (char *)sub_401080 - (char *)lpAddress - 5; ；跳转到sub_401080地址处

这样写是因为汇编语言JMP address被编译后会变成机器指令码，E9 偏移地址，偏移地址=目标地址-当前地址-5(jmp和其后四位地址共占5个字节)。所以前面直接用E9，这里直接用偏移地址就省去编译生成机器码那一步。

ASCII码表类总结：

字符ASCII码做索引：

指解题中遇到如：*v1 = byte_402FF8[(char)v1[v4]]; 之类的字符做数组索引的表达式。

其中v1[v4]逐个取input_flag的单个字符，这个字符的ascii码作为偏移继续在byte_402FF8[]数组中寻址。(PS：这不是Python中list.index()函数可以用字符查找对应索引！)

ASCII码表相关：

指解题中遇到.data数据节中跟踪变量数组时显示的有大量0FFh这种不可识别字符后又有连续的可打印字符。

因为ASCII编码表里的可视字符就得是32往后了, 所以凡是位于32以前的数统统都是迷惑项，都会被显示成0FFh甚至乱码，不会被索引到的。然后后面32之后就有连续的字符串，这种就是ASCII码表。

逆向、脚本类总结：

解题逆向模板：

第一步确定Flag字符数量，第一个红框处得到flag数量是35。

第二步找到已确定的比较字符串作为基点来反推flag字符。

第三步找出逻辑中与flag直接相关的部分，该部分可以正向爆破或者从尾到头的反向逻辑。然后找到与flag没有直接关联的部分，该部分无需逆向逻辑，直接正向流程复现即可。

正向爆破：

指解题中采用枚举正向爆破的方法，让flag中的每一个字符遍历常用的字符（ascii码表中32-126），带入原伪代码中加密算法（不用逆向），如果成功，就把这个flag存入。

exe爆破传参：

指解题中能够枚举出所有符合条件的输入，但是对正确输入后程序进行的一系列操作不想正向逻辑复现。这是我们可以使用python的subprocess模块通过循环和线程给exe文件批量传入枚举出的所有符合条件的输入，然后检测输出结果是否符合即可。

C语言写脚本：

指解题中对于不需要逆向逻辑的单纯去除冗余代码算法的题目，需要仿写去除冗余代码后的逻辑，由于只是仿写，所以原本的伪代码很难用python复现，这时就需要复制粘贴修改成C语言脚本了。

代码截断重写：

指解题中在有较完整源代码的情况下flag生成与用户输入无关，而且源代码逻辑平铺，代码量少，没有过多函数封装，则可以单独截断提取出flag生成的函数或逻辑，然后运行截断程序输出flag。

出人意料的flag：

指在题目中获取到了flag，但是这个flag可能长得不像flag，或者flag还要经过进一步的脑洞处理，而不是常规的解密处理。

栈、参数、内存、寄存器类总结：

栈地址连续小数组：

指一些本来应该是大数组的变量被IDA识别成分割成两个或多个连续地址的小数组来使用，可以通过查看栈中的地址排列或循环中的循环数大于单个数组空间来发现，也是需要更加细致才能分析出来。

栈地址连续小字符串变量：

指一些本来应该是大字符串的单个变量被IDA识别成分割成两个或多个连续地址的小字符串变量来使用，可以通过查看栈中的地址排列来发现，也是需要更加细致才能分析出来。

同地址变量：

指IDA生成的伪代码中，多个不同名字的变量实际上指向了同一个地址。所以会遇到操作这个变量，取传入另一个变量的现象，这通常具有很大迷惑性，需要我么双击查看栈中的地址才行。

栈中过渡变量反序字符串：

指一些题目本来取输入的字符串变量的最后一位，但是IDA插入了过渡变量来使分析变得困难，如v5 = (char *)&v11 + 7;这里v11就是过渡变量，指向输入字符串input_flag的第16位，所以这里v5指向输入字符串input_flag的最后一位，栈中地址又是从下到上，高位到低位的，所以反序操作标志是v6 = *v5--;

地址小端存放与正向：

指字符串数字等在内存中是反向存放的，如v7 = ‘ebmarah’，如果用地址来取的话要反向，如果用数组下标来取的话才是正向。

高低位分割数：

指一些本来应该是两个小类型变量的数被IDA识别成一个大类型然后分成高位和低位来使用，需要更加细致才能分析出来。

可变参数混淆：

指解题中IDA伪代码显示出来的参数数量超出，不符合逻辑，也不知道多附加了什么操作。查看反汇编才发现并没有传入那么多的参数，原伪代码中之所以有那么多参数是因为C语言的可变参数的特性,有些参数显示了但是并没有用上。

地址差值+数组组合遍历字符串：

指解题中遇到地址减地址的操作如：v4 = input_flag - v1; 然后通过数组组合如：v1[v4]。

这里V1作为地址和v4作为数组v1[v4]执行的是v1+v4的操作，就是v4+v1=input_flag。因为数组a[b]本质就是在数组头地址a加上偏移量b来遍历数组的，所以这里是一种遍历input_flag的新操作。

寄存器传参：

指解题中涉及寄存器作为参数传入，但是有时候IDA无法反汇编出寄存器参数的传入。解题中发现异常如：传入参数为input_flag，但是比较的却是另一个变量，这时就可能是寄存器传参了，要通过查看汇编代码来发现。

/x7f截断字符串：

/x7f可以阻断字符串，在IDA中会把一个长字符串分隔成两行的短字符串。如：xlrCj~<r|2tWsv3PtI /x7f zndka

argv[]外部调用输入参数符合条件：

指解题中需要使用命令行传入参数。

如：int main(int argc, char *argv[])

$./a.out testing1 testing2

应当指出的是，argv[0] 存储程序的名称，argv[1] 是一个指向第一个命令行参数的指针，*argv[n] 是最后一个参数。如果没有提供任何参数，argc 将为 1，否则，如果传递了一个参数，argc 将被设置为 2。

地址差值操作：

指解题中用数组、字符串首地址、寄存器等地址类或栈地址类的差值来进行操作，有时直接用地址差值做循环条件。此类型可以双击IDA查看栈地址，也可以看IDA陈列在函数上面的类型定义中的esp和ebp相对偏移地址来算出地址差值。特别的rsi源地址寄存器是相当于rsp寄存器的。

函数类总结：

函数逻辑封装：

指关键逻辑被封装成自定义函数，需要自己双击跟进并总结出函数作用，需要通过动态调试验证猜想的作用。

函数名称暗示：

指题目给出的自定义函数名称有含义，可以概括该函数的大致作用，来给总结函数作用提出一些方向性的指导。

函数积累：

指题目中有没做笔记的函数需要终点重温和积累一下。

环境准备函数：

指在用户输入命令之前，是系统程序运行时的自执行代码，是为程序渲染环境做前期准备用的，没有必要弄懂它。如easy_Maze中的step_0和step_1是为程序做迷宫地图的，设计迷宫算法，弄懂它与解题没有太大关系，而且前期准备环境的算法函数也不会有故意出错的地方来设考点，毕竟考点是走迷宫。

lambda自定义函数：

指C++类解题中遇到lambda表达式了，这是C++自定义函数的关键字，函数名入口在前面紧接着std的部分那里，不同常规的C++长模板类名最后一个才是关键函数。不过前面的不过直接双击lambda也是可以跟踪的。

C++函数名前置：

指C++类型解题中，C++长模块关键类名函数中的关键函数名不是常规的在最后面，而是在最前面的std::xx中，除了C++自定义函数之外，C++有很多函数名都在前面std::xx处。最后面通常会有operator运算符搭配在一起。

IDA等软件类总结：

GDB动态调试：

指使用GDB来进行ELF文件的动态调试。

IDA动态调试：

指使用IDA来进行ELF或windows文件的动态调试。

IDA热键重新反汇编：

指解题中必须使用IDA热键对处理过或未处理的的错误反汇编代码重新分析，以至生成新的正确的反汇编代码，常用的有U取消代码定义、C重新定义反汇编、P重新生成函数等，多用在混淆和花指令区。

IDA热键a生成数组：

指解题中对IDA零散的单个字符可以使用热键a生成数组，即长字符串。如果中间没有截断，则可以正常生成字符串。

IDA对char型(byte)的4*计数：

指解题中虽然IDA伪代码显示的 i 是 int 型，但是计算的时候通常会变成 4*i ，这通常会具有干扰性，所以我们要知道这是IDA默认把 i 当成byte类型即可，4*i 和 int 型的 i 是一样的。

单层交叉引用查看：

指在解题中只能确定一些少量的被调用函数，这些函数可能是自定义函数也可能系统函数。通过IDA的function call或Ctrl+x操作来查看改函数被谁调用，从而找到主逻辑所在的函数。

多层交叉引用查看：

指在解题中一开始获得的是比较深层次的被调用函数，需要多次查看交叉引用才能锁定最终的主逻辑所在函数。

IDA的Hex View图热键：

指在IDA的Hex View图右键菜单data format可以调成固定byte类型和有无符号类型。如调成4bytes和signed即可用±1简化显示。

align错误反汇编：

指IDA反汇编过程中自动把多个0判断成对齐操作了，然后会影响数据的分段和脚本的编写。如：align 8是因为前面dd 24h中本来是db 24 0 0 0 然后后面一个双字是dd 0 也就是db 0 0 0 0，IDA把这连着的7个0当成了间隔，那上一个数和下一个数间隔就是8了，所以IDA生成了align 8。我们只要鼠标右键undefine或把上面的dd 24改一下数据大小即可重定义align 8，重新生成数据了。

nop修补垃圾代码：

指花指令类型的题目中地址运算动态跳转情况下，被跳过的指令没有任何用处。但是代码和数据混杂在一起是无法重新F5反汇编生成函数的，垃圾代码中必须用0X90的nop填补才行。用脚本批量填补时通过U键查看垃圾代码的标志指令作为垃圾代码判断条件即可。

动态地址运算处理：

指花指令类型解题中遇到地址运算动态跳转类的反静态汇编模糊代码，如：jz short near ptr loc_400A54+1。需要协助IDA反汇编代码，通过用U取消错误的400A54代码段的定义，然后在400A55处用热键C重新生成反汇编代码后，前面jz short near ptr loc_400A54+1会直接变成jz short near ptr loc_400A55。

F7和F8交叉使用：

指花指令类型解题中不知道断点下在哪里，只能直接运行程序后在输入的地方按下暂停键来先把输入函数定位出来。但是由于输入函数是导入库的函数，所以这时会暂停在0x7开头的导入表地址处，直接单按F8或F7会发现陷入死循环的状态跳不出来，只有F7和F8交叉按才会慢慢跳出到0x4开头的真正代码处。

设立硬件访问断点：

指解题中遇到手动脱壳类型，如利用ESP脱壳定律手动脱壳。这时要根据ESP脱壳定律需要设立对应大小的硬件访问断点，使程序主代码解压后停在OEP处。

OD手动脱壳操作：

指手动脱壳中通过其它操作找到了程序主代码解压后的入口点OEP，然后通过OD插件OllyDump来脱壳当前调试的进程。

工具脱壳：

值传统的用工具压缩的文件，可以直接用工具来解压缩，即脱壳。

专业脱壳工具：

指用工具脱壳中如果exeinfope中有推荐的专业脱壳工具，则直接使用专业的工具。像万能脱壳工具这些可能会生成错误的伪代码，会造成解题错误，所以有专业的工具最好使用专业的工具。

导入表修复：

指加壳类题目中脱壳后程序出现错误，这通常是导入表出错。此时需要用importREC修复导入表，因为ImportREC是内存转储，所以要在程序运行时才能判断导入的库。

IDA伪代码生成优化：

指IDA反汇编生成伪代码中会像c语言的各种编辑器中采取的release版优化输出一样，汇编代码会因为O2优化而与源代码有较大差异。

如固定跳转中会发现ida不可能运行到的代码干脆不进行反编译，甚至如果后面代码操作和上面的处理没关系的话也不会反汇编上面的代码。

解决方法就是把固定跳转的地方用0x90的nop替换掉，注意保持指令大小不变。

动调验证值猜想：

指解题中对某个关键变量的取值或函数功能进行大致的猜想，然后这需要通过动态调试在该位下断点后查看结果是否符合猜想。

验证函数功能时记住在函数外追踪全局变量变化，而不是在函数内追踪局部变量。

双击追踪变量值的时候记住变量类型大小要对得上，有时候变量设置成正确类型大小后会显示一个地址，变量值全在地址内，而不是在双击追踪的.data地址处

算法类总结：

main算法逻辑平铺：

指主要算法代码都在main函数中，不涉及解题算法之外的其它操作，而且代码逻辑平铺、显而易见，没有把关键逻辑分成自定义函数形式，不需要频繁跟进函数。

范围算法积累：

指解题中有涉及用户输入的范围内判断以及逆向算法时对于范围处理的过程中值得注意和积累的地方。如：Str[i] > 90 || Str[i] < 65

地址赋值算法积累：

指解题中涉及对关键字符串如用户输入字符串的操作，原代码中会先把输入字符的地址赋值给变量，即让一个变量指向输入字符串然后再开始修改，这是两步操作，需要辨认。

二维数组算法积累：

指解题中涉及二维数组，用户输入与二维数组要取的下标相关，逆向时要明确是二维数组逻辑以及一维在哪里确定，举例如： *(char *)(v3[i % 3] + 2 * (i / 3)) - *(char *)(i + a1) != 1，(char *)(v3[i % 3] 确定了一维，+ 2 * (i / 3))继续在一维上面取字符串对应位，然后和*(char *)(i + a1) 字符串数组比较且要满足 = 1

数组首地址变化遍历字符串算法积累：

指解题中需要对数组进行赋值，但是数组首地址一直在变化，如：v7++ 。赋值的时候就会出现不断给v7[-1]赋值的现象，因为数组首地址一直往后移动了。但是像ebp一样的基址并没有移动，所以最后的比较不影响。

遍历字符加1算法积累：

指解题中遇到对变量每次加1的算法公式的新形式，如结合移位操作的加1算法，v3+=2 v7=v3 >> 2，加1算法通常与字符串遍历相结合。

遍历字符加2算法积累：

指解题中遇到对变量每次赋值 2 个字符的算法公式的新形式。

如：%02x，x 表示以十六进制形式输出，02 表示不足两位,，前面补0输出，如果超过两位，则以实际输出。

sprintf(&s1[2 * i], "%02x", (unsigned __int8)v11[i])的意思是把相当于char的__int8的两位输出到&s1[2*i]中，也就是一次输出两个__int8(char)类型的v11[i]到s1的偶数地址中，所以也相当于遍历赋值。

for空执行循环遍历字符串：

指解题源代码用空执行的for循环 i ，当 i 没有字符值时就会跳出循环，是一种遍历字符的新操作，通过地址差值用来计算字符串地址长度。

如：for(i=a1; *i; ++i)

; #空操作执行

字符串拆分算法积累：

指解题中IDA对多个连续的字符串按打乱的顺序以下标的方式分别按多组赋值，这种字符串拆分赋值的方式需要动态调试或耐性的一个个跟踪分析才能梳理清楚。

相同异或为0算法积累：

指解题中遇到特定异或为0的条件则可以采用上面的定律。如： *result = (108+argument[1]) ^ argument[2] = 0 即 argument[2]=(108+argument[1]) 因为相同异或才为0

可打印字符过滤算法积累：

指解题中遇到flag等关键字符在内的混杂的大量字符中，要通过多层过滤来一步步生成flag的算法，可打印字符范围内可用算法如：if ord(i)>=32 and ord(i)<=125:

大小写字符转换算法：

指解题中有一些算法范围波动比较少，看似逻辑相关，实际上只是大小写的ASCII值转换而已。

递归调用算法：

指解题中遇到函数内递归调用自己，传入参数也会在调用时修改的算法，当需要用python仿写递归算法时可以通过超范围的循环来实现递归，因为设置同样的条件，递归不满足时仿写的循环也会退出。

杨辉三角算法：

指解题中遇到对传统数学杨辉三角算法的代码实现，辨认的特征是通过关键代码判断是否符合杨辉三角算法的特性，如：在一维中用（n*(n+1)/2的前n行总数来遍历到特定行，又如：2^n来求第n行的和等。通常涉及等差、等比数列。

斐波那契数列算法：

指解题中涉及传统数学斐波那契数列算法的代码实现，用的是fib(j)的斐波那契数列函数直接生成，遇到该函数时要懂得辨认题目考察的内容与斐波那契数列相关。

>> 和 % 运算符算法积累：

指解题中遇到 >> 和 % 运算符的操作。>> 运算符其实是不带余数的除法 / 算法，单取整数部分。% 运算符其实是不带整数的求余运算，单取余数部分。

如： v6 = (v5 >> 4) % 16 是除以16后的整数部分， v7 = ((16 * v5) >> 4) % 16 是乘16后除16再取16内的余数，也就是直接取16内的余数。一个取整数，一个取余数，所以他们的逆向算法就是16 * v6 + v7

(既然v5 >> 4相当于v5除以16取整数部分，是不带余数的除法 / 算法。顺带说一下，v5 & 0xf 相当于v5除以16取余数部分，是完全的求余%算法，这里0xf是4位所以为16。)

移位算法积累：

指解题中遇到类似base64实现的位移动操作考点，特别注意4 * input_flag[i]这种向右移动的乘法变式，这其实是

2<<input_flag[i]操作，更关键的是逆向时也要注意限制位数为8位的 &0xff 。

运算符优先级注意：

指解题中应该要准确判断长运算式的优先级顺序，写脚本中也应该尽可能使用括号来固定优先级，否则会出现结果的错误。

负数作循环条件：

指解题中遇到负数作循环条件的情景需要明白这不是死循环，而是正数大循环。如：while(-1)，在32位里 -1 就是 FFFFFFFF，就是100000000 - 1。所以这一下子就转正了！如果是while(-a2)，所以就循环100000000 - a2次。

随机抽取比较：

指在解题中以随机数做基准，取各个对应的字符进行比较。其实就是在相同的字符中取随机但同样的位来比较，所以逆向是要顺序取。

取限制位数算法：

指解题中遇到源代码有__int8这样的限制，这是取前8位，python中可以使用&0xff这种方法，因为&在Python中是逻辑与运算，所以与的时候就保留了前8位，如：flag+=chr((v12^v15)&0xff)。取前16，32位都可以套用这个方法。

RSA的ASCII字符整数16进制拆分转换算法：

十六进制数组0~F中，有操作将输入的flag字符传入，以字符ASCII码操作后的结果为索引取十六进制内的数。

如：byte_202010[flag >> 4]和byte_202010[flag & 0xFF]。

一个整数一个余数你会发现这是把输入字符变成两个分开的十六进制存储起来，比如输入字符 '1'，它的整数是49，49除16的整数是3，余数是1，在byte_202010下标中分别对应3和1，构成的31就是字符 '1'的ASCII的十六进制形式，只不过是分开的十六进制，3 1 共两个字节。

base64加密/解密算法：

指题目中存在加密，，而且是传统的base64加密算法，需要自己去辨认算法的特点，能够在做题中判断出来。

md5加密/解密算法：

指解题中存在加密，而且是传统的md5加密算法，需要自己去辨认算法的特点，能够在做题中判断出来。

base58加密/解密算法：

指解题中存在加密，而且是传统的base58加密算法，右移位 >> 6和 >> 8，一个模58，一个除58是base58加密的关键。需要自己去辨认算法的特点，能够在做题中判断出来。

RSA加密/解密算法：

指解题中存在加密，而且是传统的RSA加密算法，需要自己去辨认算法的特点，能够在做题中判断出来。

奇数偶数判断算法：

原代码逻辑中出现 i&1 的判断，这其实是判断奇数还是偶数来的，以后遇到要懂得辨析。

倍数条件算法积累：

指解题中对输入flag的固定距离或倍数位做操作如：1、4、7。2、5、8。0、3、6。这种位数，通常的算法有除法符号除去非余数，如:3*(v3/3)==v3 和 ++v11==3 以及 v11 == 2这种类型。

超位数循环截取算法：

指解题中循环条件超过了输入flag的位数，从而照成循环结果会超出flag的长度。但是因为循环条件是符合逻辑的，而且不会因为位数超出就乱了结果，也就是说前面flag位数范围内还是符合flag生成的，所以这种题目最后都会有一个flag位数截取，来截取出flag位数内符合条件的前面的字符。

switch正向代入推导：

指解题中IDA生成的伪代码中有关键逻辑是switch的，那这部分无需逆向逻辑，直接正向推导每个满足case条件的字符即可，同迷宫一样都是手动。

16进制转十进制算法：

指解题中对于要求输入16进制数或者原本就存在16进制数中有让转为10进制数的操作。如v10=v16-55，v10=v16-'0'。

16进制范围A~F，减去55之后就是变成了实数的10 ~15。16进制范围0 ~9减去字符'0'之后就变成了实数0 ~9。

所以就是0 ~ F分别对应实数的0 ~15之所以减得不同是因为ASCII表中0 ~ 9和A ~ F并不相连。

IDA循环左右移动算法：

指花指令类型解题中数据变指令的修改是需要左右移动的，当我们选择静态修补的时候，就需要顺着原程序中自修改逻辑来左右移动或更进一步操作。

附上循环左右移动代码：

```
def ROR(i,index): #循环右移

tmp = bin(i)[2:].rjust(8,"0")

for _ in range(index):

tmp = tmp[-1] + tmp[:-1]

return int(tmp, 2)

def ROL(i,index): #循环左移

tmp = bin(i)[2:].rjust(8, "0")

for _ in range(index):

tmp = tmp[1:] + tmp[0]

return int(tmp, 2)
```

大量判断少赋值的字符串比较算法：

指解题中原代码分析部分遇到一堆的判断，但是却只有很少的赋值修改操作，这通常是某个系统函数的原型。

那没有修改操作，函数的作用基本就是判断每个字符是否相等，也就是对传入的字符串参数进行简单比较的函数。

伪随机数加密算法：

rand是伪随机数，要用srand生成随机数种子才行，不然产生的随机数列表都是一样的，而单独产生的随机数也不会在随机数列表用随意取值，而是固定的第一次这个值，第二次那个值。所以这里我们可以直接修改源代码或动态调试，打印出对应随机数的值。

小端转大端存储算法：

指解题中十六进制变量值本来应该是小端顺序逆序存储在内存中，但是源代码通过高低位截取替换把小端顺序转为大端顺序存储在数组中了。如：v3的0xB11924E1本来应该是小端顺序逆序存储在内存中的，但是通过HIBYTE、BYTE2、BYTE1、BYTE直接把v3按大端顺序存储了。

浏览器操作：

JS控制台操作：

指JS类解题过程中通过在源代码中嵌入console.log(变量.args)函数，打印输出对解题有用的信息，帮助进一步解题。并在控制台console窗口中一堆折叠数据中逐个展开来选择对应信息。

特殊语法积累：

模板复制操作：

指C++类型的解题中，题目原代码使用了：

template <class Ch, class Tr = char_traits <Ch>, class A=allocator <Ch>> class std::basic_string

类型的字符串模板实例化类 basic_string，来进行字符串复制操作。

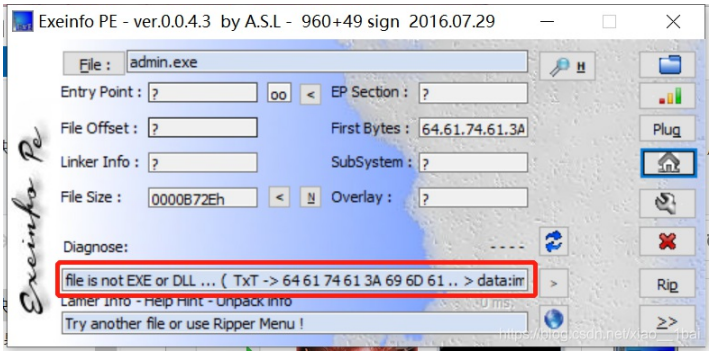
或者使用了：

template <class Ch, class Tr = char_traits <Ch>, class A=allocator <Ch>> class std::_M_construct<char *>

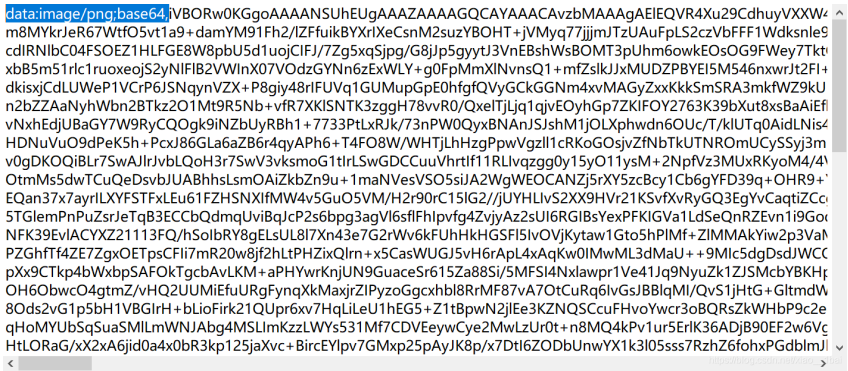
类型的结构体模板实例化来进行结构体的复制操作。总的来说都是复制操作。

非EXE文件类型

bugku 逆向入门：（实际TxT文件、不能直接运行）



直接去掉.exe后缀用记事本打开，直接搜索bugku，无果。看到文件开头：



是base64加密的图片，于是用在线网址(http://tool.chinaz.com/tools/imgtobase/)解密得到二维码：（扫码即得flag）

攻防世界parallel-comparator-200：（.c文件、大小写字符转换算法、函数积累、相同异或为0算法积累、线程操作积累、不能直接运行、伪随机数加密算法）

下载附件，一个.c后缀的文件，devc++中查看代码。

这里犯下第一个错误：混淆代码太多，线程一开始没学，简单学了后发现也和解题逻辑没有太大关系，可以把线程划分为系统函数这一块：

```
#include <stdlib.h>

#include <stdio.h>

#include <pthread.h> //linux的线程库 ， 所以要在linux中才可运行

#define FLAG_LEN 20

void * checking(void *arg) {

char *result = malloc(sizeof(char));

char *argument = (char *)arg;

*result = (argument[0]+argument[1]) ^ argument[2]; //对 first_letter、 differences[i]、 user_string[i]进行简单操作

return result;

}

int highly_optimized_parallel_comparsion(char *user_string)

{

int initialization_number;

int i;

char generated_string[FLAG_LEN + 1];

generated_string[FLAG_LEN] = '\0';

while ((initialization_number = random()) >= 64); //无用循环

int first_letter;

first_letter = (initialization_number % 26) + 97; //initialization_number从0~25取值 +97后ASCII对应小写的a~z

pthread_t thread[FLAG_LEN]; //创建数组型的线程标识符 ， 20线程句柄

char differences[FLAG_LEN] = {0, 9, -9, -1, 13, -13, -4, -11, -9, -1, -7, 6, -13, 13, 3, 9, -13, -11, 6, -7}; //定义20个元素的char数组

char *arguments[20]; //定义20个char型的指针数组

for (i = 0; i < FLAG_LEN; i++) {

arguments[i] = (char *)malloc(3*sizeof(char)); //每个指针指向3个char字节划分的数组头

arguments[i][0] = first_letter; //first_letter由于 initialization_number = random()而未确定

arguments[i][1] = differences[i]; //已确定
```

```
arguments[i][2] = user_string[i]; //用户输入字符，未确定

pthread_create((pthread_t*)(thread+i), NULL, checking, arguments[i]); //调用上面checking函数对arguments三字节数组进行简单操作

}

void *result; //定义一个数组，用上面的异步线程赋值

int just_a_string[FLAG_LEN] = {115, 116, 114, 97, 110, 103, 101, 95, 115, 116, 114, 105, 110, 103, 95, 105, 116, 95, 105, 115}; //定义一个20个元素的数组

for (i = 0; i < FLAG_LEN; i++) {

pthread_join(*(thread+i), &result); //阻塞线程，让线程一个个执行

generated_string[i] = *(char *)result + just_a_string[i]; //把 just_a_string数组加到result中 赋值给generated_string数组

free(result);

free(arguments[i]);

}

int is_ok = 1;

for (i = 0; i < FLAG_LEN; i++) {

if (generated_string[i] != just_a_string[i]) //这里比较generated_string和 just_a_string数组，而generated_string数组在前面赋值中= *(char *)result + just_a_string[i]，所以result等于0才行

return 0;

}

return 1;

}

int main()

{

char *user_string = (char *)calloc(FLAG_LEN+1, sizeof(char)); //分配21个字符空间，除去0结尾就是20个字符

fgets(user_string, FLAG_LEN+1, stdin); //获取用户输入

int is_ok = highly_optimized_parallel_comparsion(user_string);

if (is_ok)

printf("You win!\n");

else

printf("Wrong!\n");

return 0;

}
```

关键代码判断有两条：（所以result是要为0了，因为0加任何数都为0。）

generated_string[i] = *(char *)result + just_a_string[i];

if (generated_string[i] != just_a_string[i])

给result赋值的语句中唯一不确定的就是argument[0]，也就是 first_letter = (initialization_number % 26) + 97：

*result = (argument[0]+argument[1]) ^ argument[2]; //对 first_letter、 differences[i]、 user_string[i]进行简单操作

在这里我查了很多资料，很多人直接用108代替argument[0]，也有人用first_letter的范围97~122来批量计算，这里我两种都说：

第一种：108，这里也是犯下的第二个错误，以前就听过rand是伪随机数，要用srand生成随机数种子才行，不然产生的随机数列表都是一样的，而单独产生的随机数也不会在随机数列表用随意取值，而是固定的第一次这个值，第二次那个值。所以这里我们可以直接修改源代码调试，打印出first_letter的值。

(PS：linux中C语言要使用gcc main.c -lpthread -o main编译方法来编译带pthread.h库的文件)

while ((initialization_number = random()) >= 64);

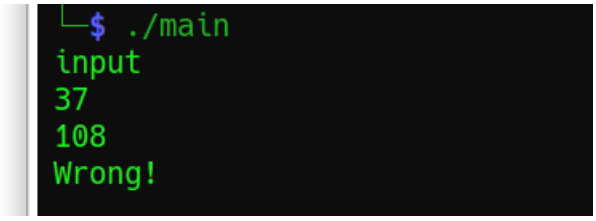
printf("%d\n",initialization_number); //打印initialization_number

int first_letter;

first_letter = (initialization_number % 26) + 97;

printf("%d\n",first_letter); //打印first_letter

结果可以看到，一个37，一个108，所以108就是调试过来的：



知道108后写脚本，这里犯下第三个错误：

因为源代码是*result = (argument[0]+argument[1]) ^ argument[2] = 0 即 *result = (108+argument[1]) ^ argument[2] = 0 即 argument[2]=(108+argument[1]) 因为相同异或才为0.(一开始我并不清楚这个逻辑，这里也可以说是一个算法积累了)

first_letter=108

differences=[0, 9, -9, -1, 13, -13, -4, -11, -9, -1, -7, 6, -13, 13, 3, 9, -13, -11, 6, -7]

flag=""

print(".join([chr(first_letter+i) for i in differences])) //这里借鉴了别人的博客，用的是列表的[ ]解析式，的确不错！也可以作为总结！

第二种方法就是从97~122的first_letter开始批量爆破计算，脚本：

differences=[0, 9, -9, -1, 13, -13, -4, -11, -9, -1, -7, 6, -13, 13, 3, 9, -13, -11, 6, -7]

for i in range(97,123):

flag=""

print(i)

for a in differences:

flag+=chr(i+a)

print(flag)

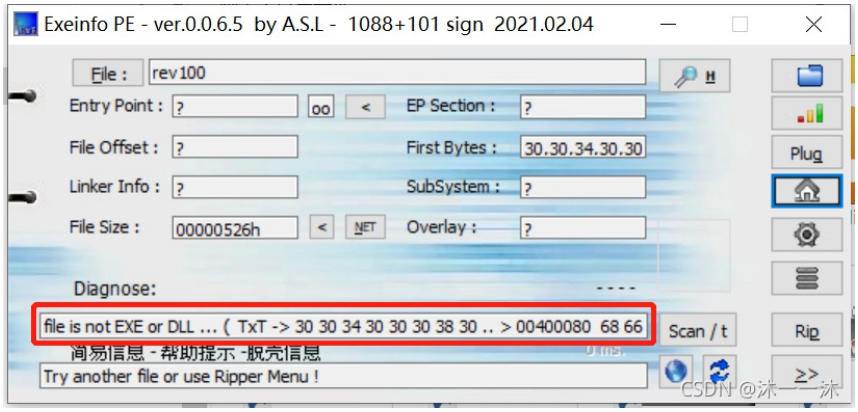
可以看到，能形成有文字含义字符串的就是108了：

```
differences=[0, 9, -9, -1, 13, -13, -4, -11, -9,
for i in range(97,123):
    flag=""
    print(i)
    for a in differences:
        flag+=chr(i+a)
    print(flag)
```

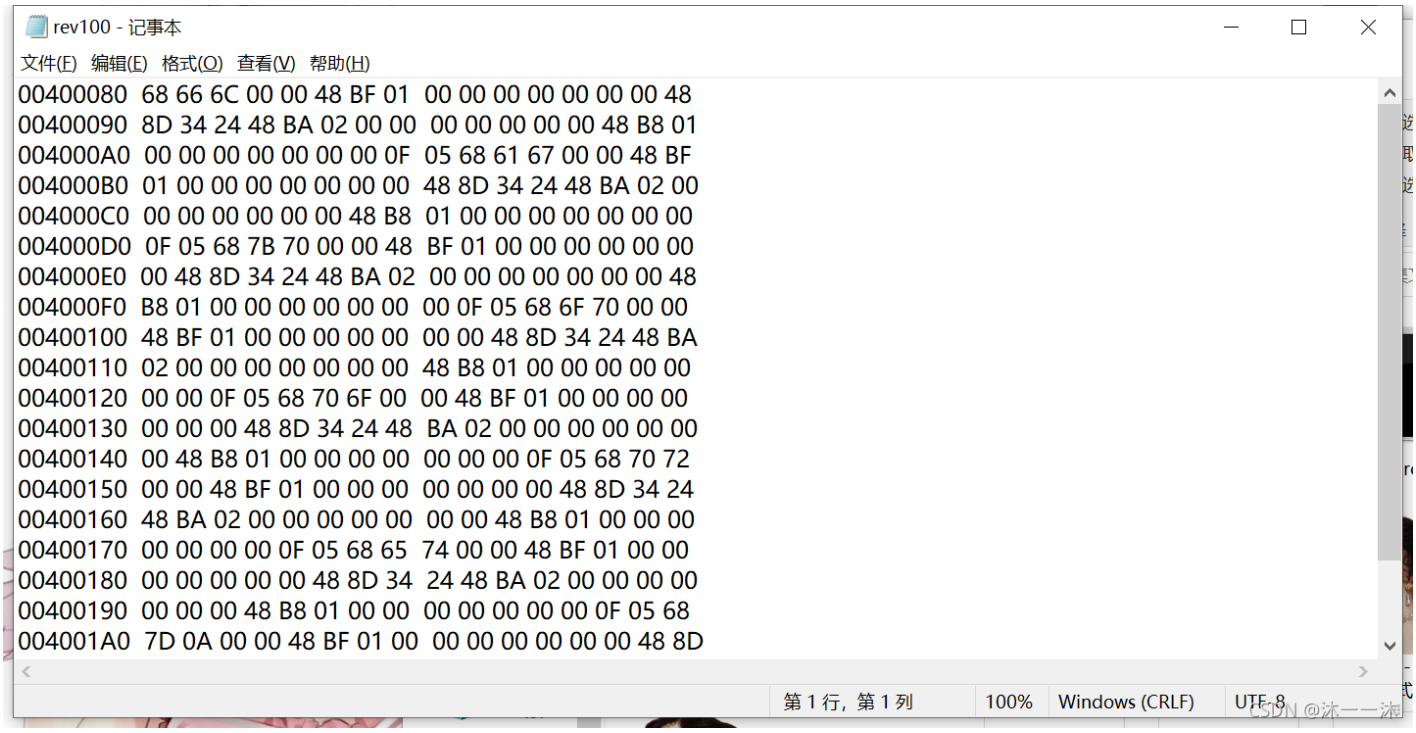
```
文件  动作  编辑  查看  帮助
bkYaoU^WYa[hUoekUWh[
99
clZbpV_XZb\iVpflVXi\
100
dm[cqW`Y[c]jWqgmWYj]
101
en\drXaZ\d^kXrhnXZk^
102
fo]esYb[]e_lYsioY[l_
103
gp^ftZc\^f`mZtjpZ\m`
104
hq_gu[d]_gan[ukq[]na
105
ir`hv\e^`hbo\vlr\^ob
106
jsaiw]f_aicp]wms]_pc
107
ktbjx^g`bjdq^xnt^`qd
108
lucky_hacker_you_are
109
mvdlz`ibdlfs`zpv`bsf
110
nwem{ajcemgta{gwactg
```


攻防世界tt3441810：（实际TXT文件、不能直接运行、出人意料的flag、可打印字符过滤算法积累）

下载附件，照例扔入exeinfope中查看信息：

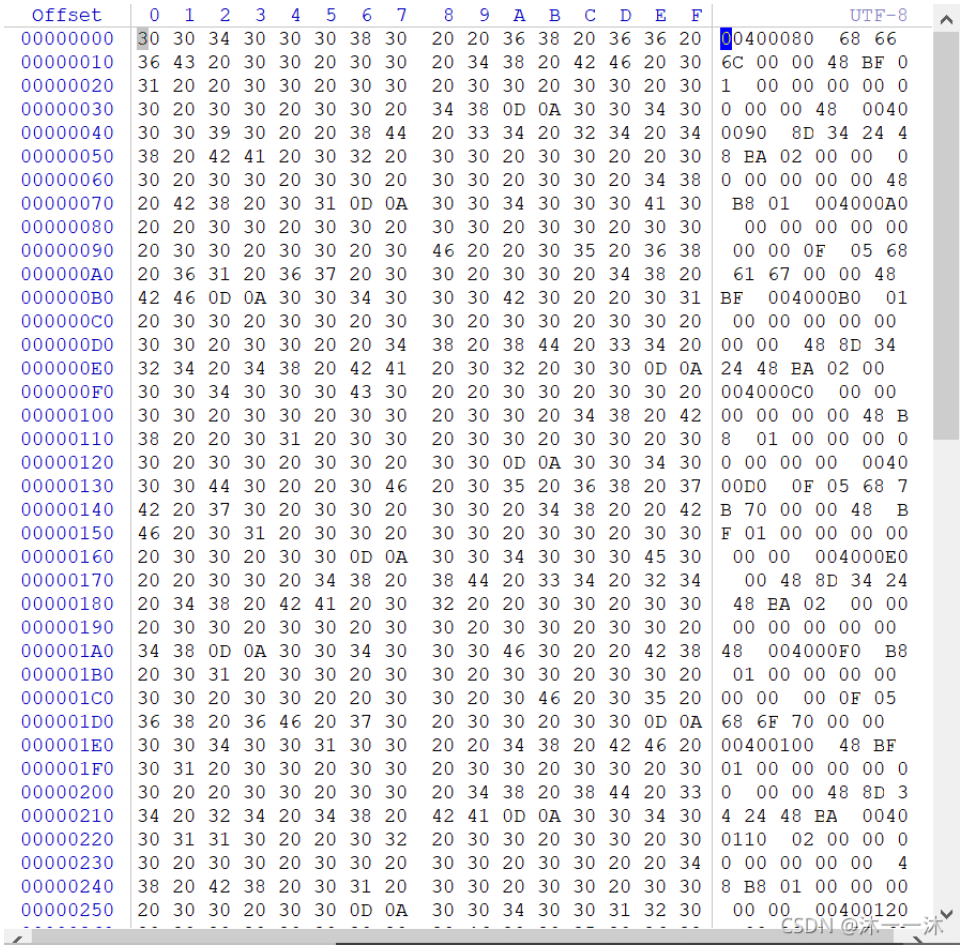


说是TXT文件，？？？记事本打开看一下：

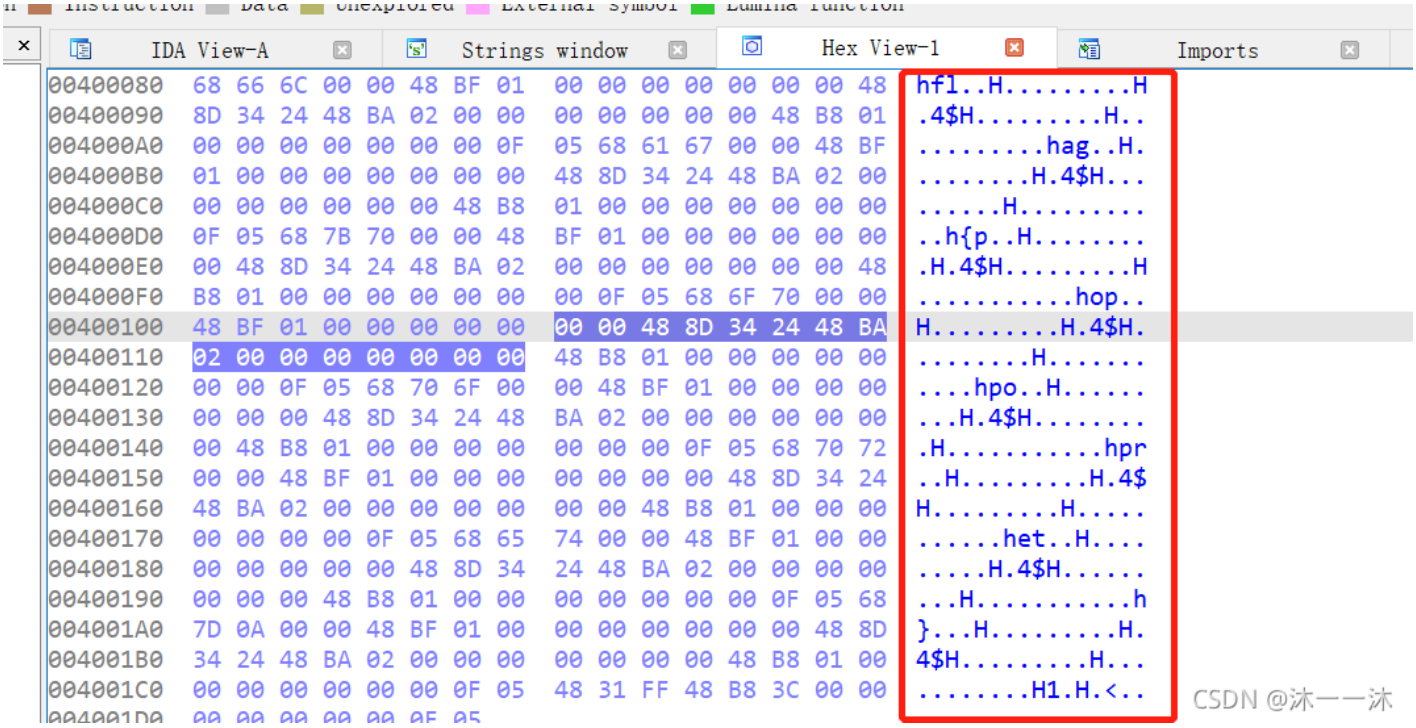


一下子懵住了，脑袋没转过来，查看了资料说Flag就混杂在这些十六进制里，winhex64打开看一下：

结果winhex64的文本显示不了字符，我还是看不出来什么。（换ASCII编码也是一样）



结果发现他们使用IDA打开的：



终于有一点字符了，之前了解过点字符是因为IDA识别不了不可打印字符，所以这里要写脚本过滤，像杂项或密码学一样的：(32~125是可显示字符)

key1=''' //这里积累第一个经验，多行的字符串可以用三引号，虽然我知道这个三引号，但是我要用时我还真想不到它。

hfl..H.........H

.4$H.........H..

.........hag..H.

........H.4$H...

......H.........

..h{p..H........

.H.4$H.........H

...........hop..

H.........H.4$H.

........H.......

....hpo..H......

...H.4$H........

.H...........hpr

..H.........H.4$

H.........H.....

......het..H....

.....H.4$H......

...H.....h

}...H.....H.

4$H.....H...

.....H1.H.<..

.....

'''

flag=""

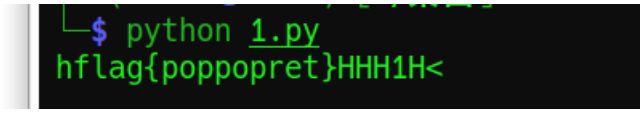
for i in key1:

if ord(i)>=32 and ord(i)<=125: //过滤在可打印字符范围内的字符

flag+=i

print(flag.replace('.','').replace('HH4$','').replace('HHh','')) //这里是我一层层看逻辑过滤的，因为出现多个. 、HH4$和HHh，所以这些都要过滤掉。

结果：



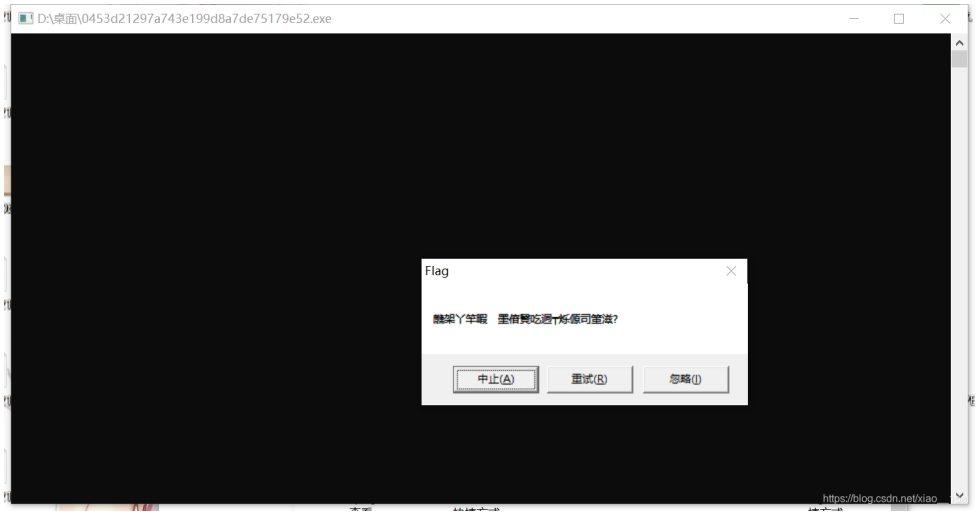
可以看到flag了，但是提交时说只提交{}内的部分，就是poppopret。这种提交方式已经见怪不怪了。

main函数主逻辑分析（C语言）

不能正常运行的EXE文件类型：

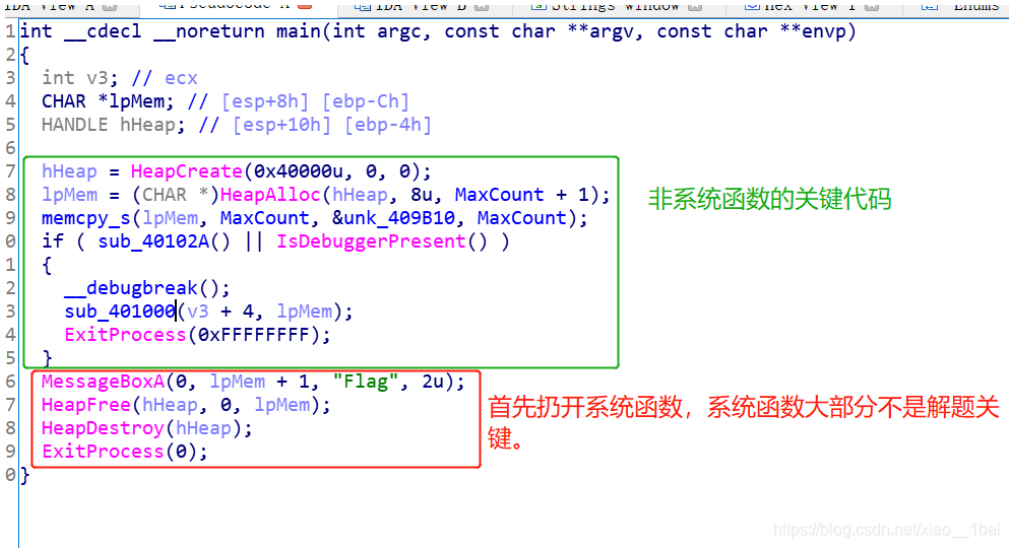
攻防世界的csaw2013reversing2：（运行乱码、int3断点考察、函数积累、不能直接运行）

win32无壳，那么既然是windows的直接双击运行一下看看：



这个就是乱码的flag了，乱码有好多种，base64加密等等这些，我们一个个排除，先扔入IDA中查看伪代码。要先看C或C++伪代码再分析反汇编结构图最后才看反汇编文本！！！！

main主函数伪代码如图：



那么解题关键就在前面了：

memcpy_s(lpMem, MaxCount, &unk_409B10, MaxCount);//这个是复制函数，把&unk_409B10处的字符串赋值给lpMem,分析后可知这是乱码的flag，双击跟踪&unk_409B10也可以看到是弹框中输出的乱码。

if (sub_40102A() || IsDebuggerPresent()) //这是判断函数，如果判断是调试器运行就执行这个解密

{

__debugbreak();

sub_401000(v3 + 4, lpMem);//这个双击跟踪进去后发现是一个运算函数，那么只能是解密算法所在了

ExitProcess(0xFFFFFFFF); //解密后就退出了，就没有后面的弹框了，需要我们自己想办法。

}

分析完了，我们开始解题，也是好几种方法：

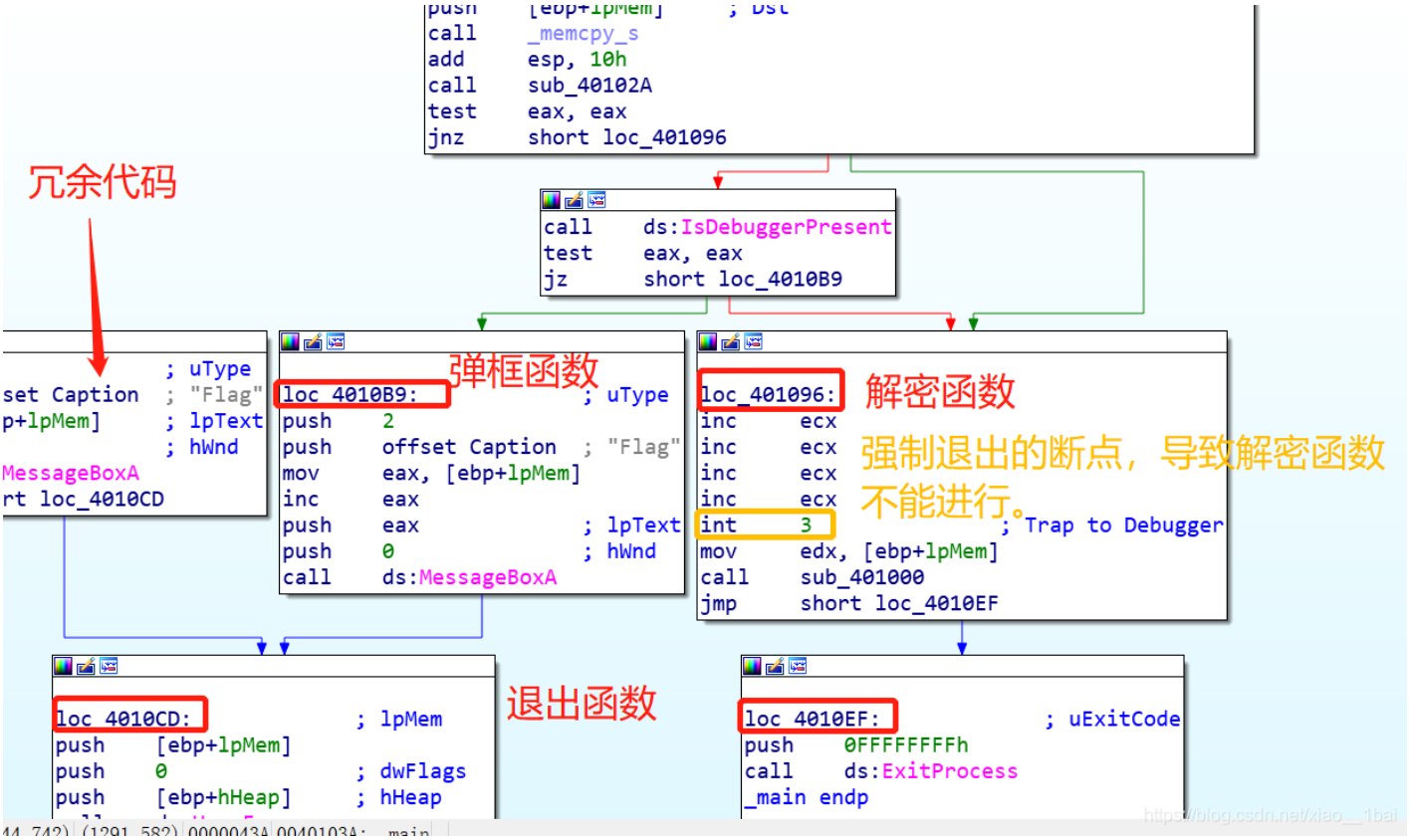
1：静态调试：根据sub_401000的解密算法自己仿照C语言或python脚本解密，因为参数都可以跟踪到。

2：动态调试：在onllydbg中进入解密流程内，解密后查看寄存器或跳转到messageboxA中进行弹出即可。

这里我用的动态调试：按照流程看完C语言伪代码后我们来看反汇编结构图：

有了前面分析基础就看懂得多了：loc_4010B9:是弹框函数所在，loc_401096:是解密函数所在，最左边那个应该是冗余代码，loc_4010EF和loc_4010CD: 都是退出函数所在。

注意这里loc_401096有个int 3;这是断点的一种，代码执行到此次会抛出异常，因为这不是我们在OD之类的调试器下的断点，所以OD之类的调试器不会处理该断点的异常，而是交给系统处理，而系统的处理方式往往是强制退出。所以我们在动态调试中要改为nop，不然后面的代码就没法执行



那么我们上onllydbg修改int 3;断点为nop：

可以看到我改了几个地方：

1：

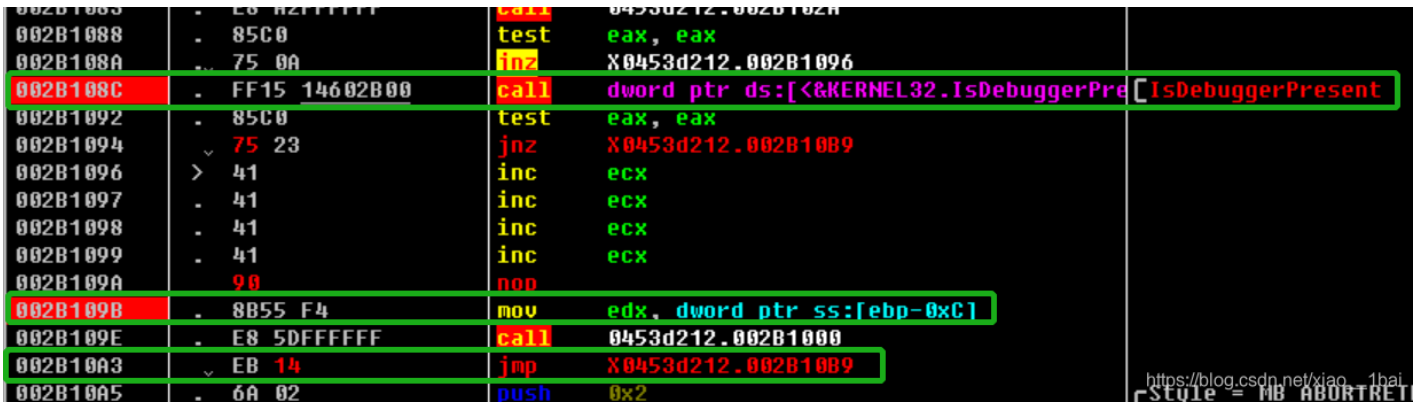
002B1094的 jz short loc_4010B9改成jnz short loc_4010B9，虽然我也不知道为什么我用onllydbg调试还是进不去解密函数，可能onllydbg被认为不是调试器吧。

2：

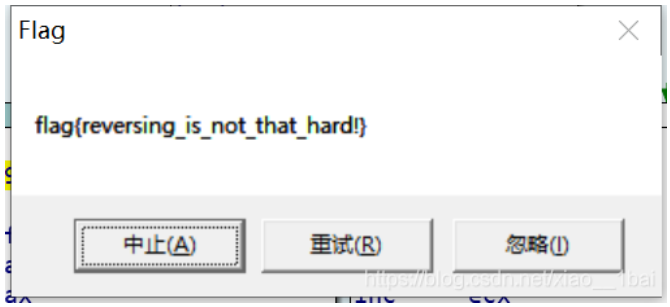
002B109A 的int 3;断点强制退出被我改成了002B109A nop，标识空操作，避免退出。

3：

002B10A3的 jmp 0453d212.002B10EF改成jmp 0453d212.002B10B9 因为这里原来解完密后就退出的，我把它转到原来loc_4010B9:的messageboxA函数去作为弹框内容输出了。(ps：我本来是跳转到最IDA反汇编结构图的左边那个冗余函数的，因为我看它也是MessageboxA函数，结果弹出个空框，对比后才发现它比第二个MessageboxA少了几行代码，原来是个坑，难怪。)，当然也可以解完密之后下断点读取寄存器内容也行。



这样就弹出flag了：



main算法逻辑平铺类型：

攻防世界逆向入门题Hello, CTF：（简单比较）

对汇编不太熟悉，只能分析伪代码：

```
  if ( !strcmp(&v10, &v13) )
    sub_40134B(aSuccess, v7);
  else
    sub_40134B(aWrong, v7);
}
sub_40134B(aWrong, v7);
result = stru_408090._cnt-- - 1;
if ( stru_408090._cnt < 0 )
  return _filbuf(&stru_408090);
++stru_408090._ptr;
return result;
```

伪代码显示用用户输入的v10和v13比较，sub_40134B是我在OD中认定的字符串输出函数puts。

所以终于找到了v13这个被比较变量了，查看与它相关的操作：

```
qmemcpy(&v13, a437261636b4d65, 0x20u);
```

一个复制函数，那么那个a4~开头的就是我们要找的了，双击跟踪：

```
.data:00408066                  align 4
.data:00408068 a437261636b4d65 db '437261636b4d654a757374466f7246756e',0
.data:0040808B                  align 10h
```

数据域发现一串十六进制字符，解码得到flag。

攻防世界open-source：(argv[]外部调用输入参数)

#include <stdio.h>

#include <string.h>

int main(int argc, char *argv[]) {//外部调用输入参数

if (argc != 4) {//输入三个参数，因为第一个是程序自己的名称

printf("what?\n");

exit(1);

}

unsigned int first = atoi(argv[1]);

if (first != 0xcafe) {//第一个参数的十六进制为0xcafe

printf("you are wrong, sorry.\n");

exit(2);

}

unsigned int second = atoi(argv[2]);

if (second % 5 == 3 || second % 17 != 8) {//第二个参数满足条件我口算有42，余数是不足才补的数，不是整除后剩的数。也就是5*9余3

printf("ha, you won't get it!\n");

exit(3);

}

if (strcmp("h4cky0u", argv[3])) {//第三个参数直接就是h4cky0u

printf("so close, dude!\n");

exit(4);

}

printf("Brr wrrr grr\n");

unsigned int hash = first * 31337 + (second % 17) * 11 + strlen(argv[3]) - 1615810207;//这里的结果hash与前面输入参数有关，鄙人不才，曾一度想修改源码不输入参数直接输出这句话，当然，没有参数的这句话就会报错。

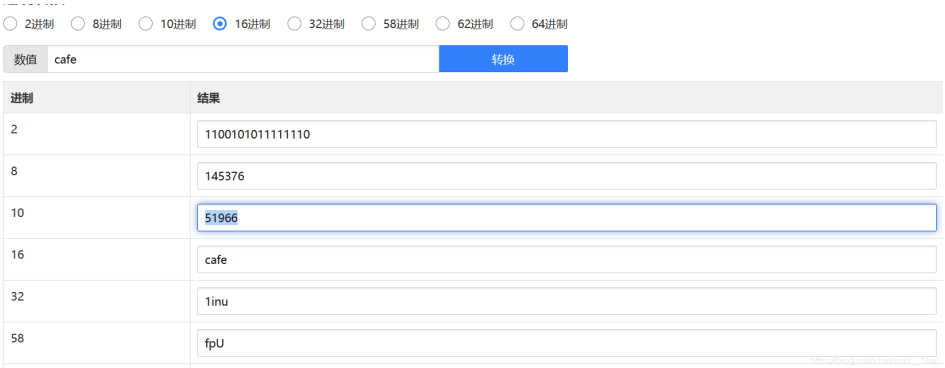
printf("Get your key: ");

printf("%x\n", hash);

return 0;

}

一开始第二个条件停了会，毕竟做题经验太少了，atoi返回的是字符串的整形，0xcafe是十六进制，整形和十六进制比较C语言内部会进行进制转换：



所以到此所有参数都解出来了，第一个是51966，第二个是42，第三个是h4cky0u.在kali虚拟机中编译，命令行接受参数执行即可：

gcc 1.c

./1.c 51966 42 h4cky0u

后来看别人做法还发现了其他解法，第一个是直接修改源码，其实也对，源码在手当然是充分利用源码的优势才对，直接把hash输出语句替换成：

unsigned int hash = 0xcafe * 31337 + (second % 17) * 11 + strlen(argv[3]) - 1615810207;

即可，反正C语言内部会自己转换，记得把第二个0xcafe处的判断语句用/**/注释掉即可。

攻防世界logmein：（地址小端存放与正向）

ELF的linux文件，在kali虚拟机中查看位数，是64位，扔到64位IDA中查看信息，主要查看伪代码：

```
size_t v3; // rsi
int i; // [rsp+3Ch] [rbp-54h]
char s[36]; // [rsp+40h] [rbp-50h]
int v6; // [rsp+64h] [rbp-2Ch]
__int64 v7; // [rsp+68h] [rbp-28h]
char v8[8]; // [rsp+70h] [rbp-20h]
int v9; // [rsp+8Ch] [rbp-4h]

v9 = 0;
strcpy(v8, ":\"AL_RT^L*.?+6/46");
v7 = 28537194573619560LL;
v6 = 7;
printf("Welcome to the RC3 secure password guesser.\n", a2, a3);
printf("To continue, you must enter the correct password.\n");
printf("Enter your guess: ");
__isoc99_scanf("%32s", s);
v3 = strlen(s);
if ( v3 < strlen(v8) )
  sub_4007C0(v8);
for ( i = 0; i < strlen(s); ++i )
{
  if ( i >= strlen(v8) )
    ((void (*)(void))sub_4007C0)();
  if ( s[i] != (char)(*((_BYTE *)&v7 + i % v6) ^ v8[i]) )
    ((void (*)(void))sub_4007C0)();
}
sub_4007F0();
}
```

在内存中都是小端存放，但是v7是按地址取，v8是按数组下标取。所以v8可以正向取。

主要的加密操作得出flag

很常规的题型，关键输入判断如下：

```
if ( s[i] != (char)(*((_BYTE *)&v7 + i % v6) ^ v8[i]) )
```

在IDA中v7按R键转换为v7 = ‘ebmarah’; (_BYTE *)&v7表示将原本是_int64类型的v7转换地址形式，转成byte型地址形式来实现1位一位读取字符串。

这里还要注意的是这里的内存是小端存放的，也就是说我们要逆着来比较v7的字符串，然后直接上python脚本：

key1=":\"AL_RT^L*.?+6/46"

key2="ebmarah"[::-1]

key3=""

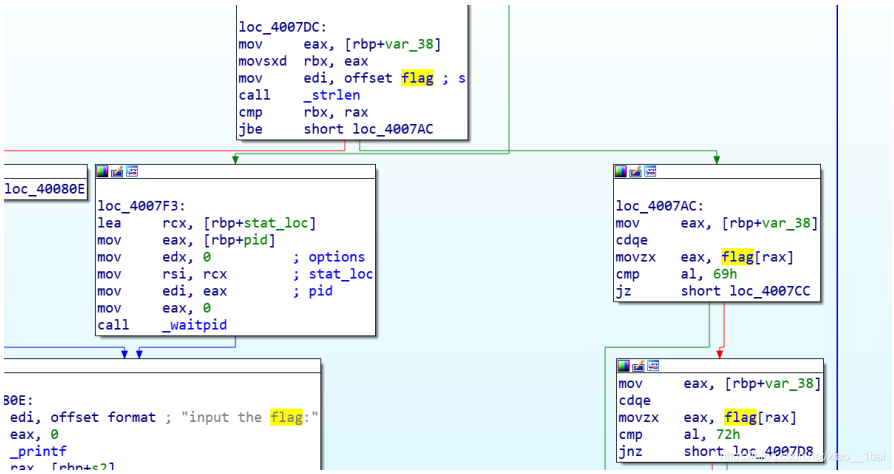
for i in range(len(key1)):

key3+=chr(ord(key2[i%7]) ^ ord(key1[i]))

print(key3)

BUUCTF的reverse2：（原flag简单操作）

一进门看到这个，还以为真的这么简单，认为offset flag就是flag地址，直接跟踪结果是假的flag，一开始还以为是提交格式有问题，比如多了个空格或者复制错了什么啊，果然还是太年轻了，见识少，就是假的flag。



查看伪代码(由于汇编能力比较差)：

```
if ( pid )
{
  argv = (const char **)&stat_loc;
  waitpid(pid, &stat_loc, 0);
}
else
{
  for ( i = 0; i <= strlen(&flag); ++i )  对原flag的操作
  {
    if ( *(&flag + i) == 105 || *(&flag + i) == 114 )
      *(&flag + i) = 49;
  }
}
printf("input the flag:", argv);
__isoc99_scanf("%20s", &s2);
if ( !strcmp(&flag, &s2) )
  result = puts("this is the right flag!");
else
  result = puts("wrong flag!");
return result;
```

可以看到前面有对flag内容的替换，就是把ASCII码等于105和114的替换成ASCII码49。直接写脚本：

flag="{hacking_for_fun}"

for i in range(len(flag)):

if((ord(flag[i])==105) or (ord(flag[i])==114)):

flag=flag.replace(flag[i],chr(49))

print(flag)

写脚本时一开始还有点问题，顺便记在这里提醒一下自己：

一开始写成flag="{hacking_for_fun}"[::-1]，受了以前的小端顺序影响，这里我们看到的就是从601081到601091的地址顺序，也就是已经按小端的来了，所以不用再反序，还有就是flag.replace()这种python字符串内置函数都是暂时的，要想保留改变还是要用赋值语句赋值成：flag=flag.replace(flag[i],chr(49))

攻防世界666：（函数逻辑封装，函数名称暗示）

64位ELF文件，无壳，扔入IDA64中查看伪代码，因为有main函数，所以直接main函数：

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char s[240]; // [rsp+0h] [rbp-1E0h] BYREF
  char input_flag[240]; // [rsp+F0h] [rbp-F0h] BYREF

  memset(s, 0, 0x1EuLL);
  printf("Please Input Key: ");
  __isoc99_scanf("%s", input_flag);
  encode(input_flag, (__int64)s);
  if ( strlen(input_flag) == key )                    // 长度为18
  {
    if ( !strcmp(s, enflag) )                         // 密文enflag为izwhroz""w"v.K".Ni
      puts("You are Right");
    else
      puts("flag{This_1s_f4cker_flag}");
  }
  return 0;
}
```

encode是名称暗示的封装函数

判断题目类型，flag是与用户输入有关的明文密文加密型，给了密文(双击跟踪)，那么根据加密逻辑函数用密文逆向逻辑解出明文即可，加密逻辑如下：

```
int __fastcall encode(const char *a1, __int64 a2)
{
  char v3[104]; // [rsp+10h] [rbp-70h]
  int v4; // [rsp+78h] [rbp-8h]
  int i; // [rsp+7Ch] [rbp-4h]

  i = 0;
  v4 = 0;
  if ( strlen(a1) != key )                            // 18长度
    return puts("Your Length is Wrong");
  for ( i = 0; i < key; i += 3 )                      // 3位一个，循环6次
  {
    v3[i + 64] = key ^ (a1[i] + 6);
    v3[i + 33] = (a1[i + 1] - 6) ^ key;               // a1改变v3,v3改变a2，返回a2为s
    v3[i + 2] = a1[i + 2] ^ 6 ^ key;
    *(_BYTE *)(a2 + i) = v3[i + 64];
    *(_BYTE *)(a2 + i + 1LL) = v3[i + 33];
    *(_BYTE *)(a2 + i + 2LL) = v3[i + 2];
  }
  return a2;
}
```


这逻辑挺简单的，所以直接上脚本即可：

a2="izwhroz\"\"w\"v.K\".Ni"

key=18

v3=""

flag=""

#print(len(a2))

for i in range(0,18,3):

v3=a2[i]

flag+=chr((ord(v3)^key) - 6)

v3=a2[i+1]

flag+=chr((ord(v3)^key) +6)

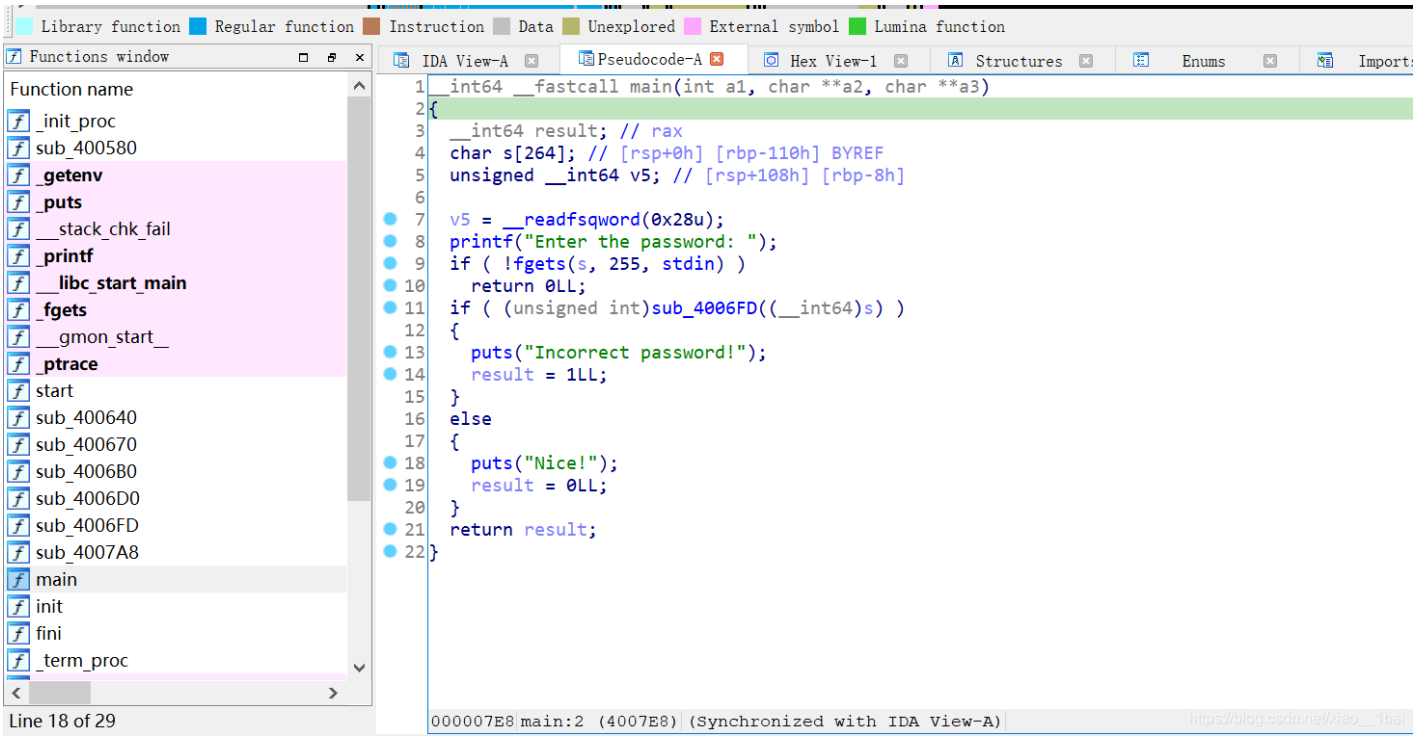
v3=a2[i+2]

flag+=chr((ord(v3)^key)^6)

print(flag)

攻防世界Reversing-x64Elf-100：（函数逻辑封装、地址小端存放与正向、二维数组算法积累）

64位ELF文件，无壳，扔入64位IDA中，有主函数从主函数开始：（PS：这里犯下第一个错误，我一开始以为没有主函数，跳到start函数中去分析了，结果有个栈指针错误，但是我还不会调，迷惘了，后来一看才发现原来有主函数）



跟踪进入判断函数并分析代码：

__int64 __fastcall sub_4006FD(__int64 a1)

{

int i; // [rsp+14h] [rbp-24h]

__int64 v3[4]; // [rsp+18h] [rbp-20h]

v3[0] = (__int64)"Dufhbmf"; //这里犯下第二个错误，我以为是普通字符串，在内存中应该小端顺序反序才对，结果是数组，数组的话从首地址开始的确是正序的了，吸取经验，以后要是不确定是不是反序就直接双击跟踪看内存即可。

v3[1] = (__int64)"pG`imos";

v3[2] = (__int64)"ewUglpt";

for (i = 0; i <= 11; ++i)

{

if (*(char *)(v3[i % 3] + 2 * (i / 3)) - *(char *)(i + a1) != 1) //这里犯下第三个错误，一开始没看见最左边的取地址符*的范围是一整个(char *)(v3[i % 3] + 2 * (i / 3))，搞到脚本编写出来障碍，这里应该这样理解，(char *)(v3[i % 3]取这v3[0]、v3[1]、v3[2]、中的第几个完整数组，+ 2 * (i / 3)是为了在确定的v3[0]、v3[1]、v3[2]中继续深入取对应数组的字符进行操作，这里的逆向逻辑也简单，就是*(char *)(v3[i % 3] + 2 * (i / 3)) - 1 = *(char *)(i + a1)

return 1LL;

}

return 0LL;

}

分析完毕，脚本：

key1="Dufhbmf"

key2="pG`imos"

key3="ewUglpt"

flag=""

key4=[key1,key2,key3]

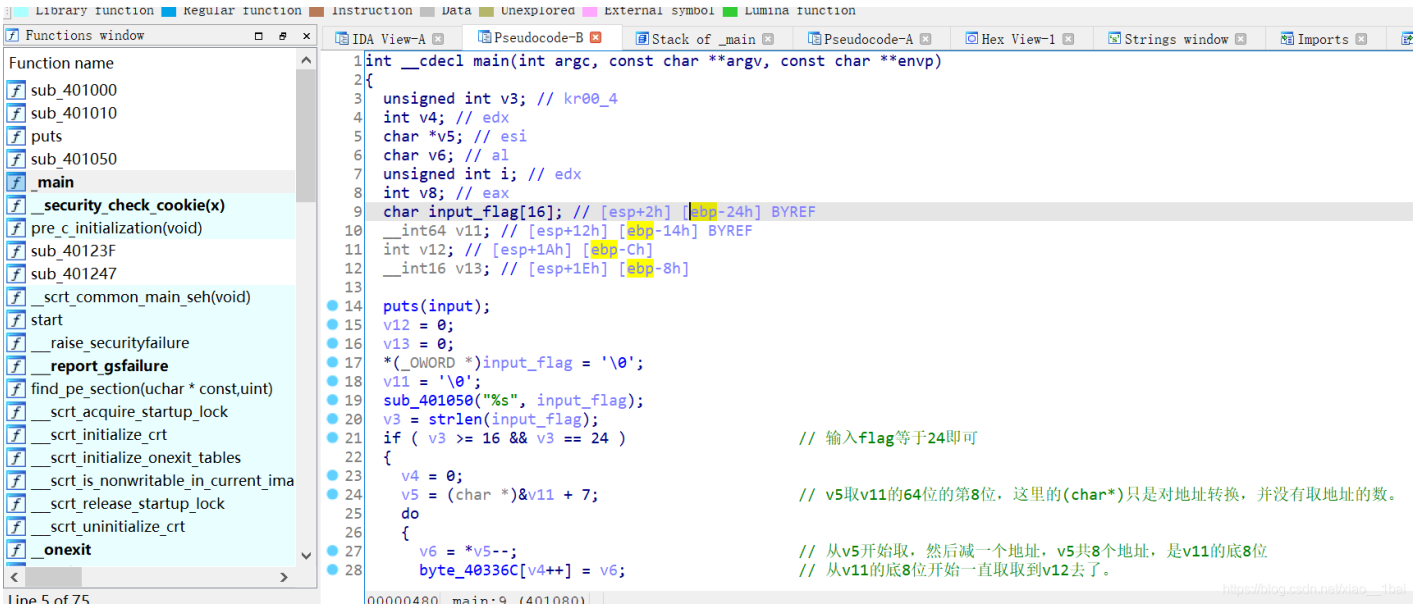
for i in range(12):

flag+=chr(ord(key4[i%3][(2*int(i/3))]) -1)

print(flag)

攻防世界EasyRE：（栈地址连续小字符串变量、栈中过渡变量反序字符串、/x7f截断字符串、运算符优先级注意）

64位ELF文件，无壳，照例扔入IDA64中查看信息，有Main函数就看main函数：(PS：下面被我注释了一下内容，不过不影响)



照例分析代码：

puts(input); //这里名称被我重命名过，判断依据是程序运行时的Input字符串在这里被引用，所以这里是输出函数

v12 = 0;

v13 = 0;

*(_OWORD *)input_flag = '\0';

v11 = '\0';

sub_401050("%s", input_flag);

v3 = strlen(input_flag);

if (v3 >= 16 && v3 == 24) // 这里两个条件其实是多余的，输入flag等于24即可

{

v4 = 0;

v5 = (char *)&v11 + 7; // v5取v11的64位地址下面的7位地址处，这里的(char*)只是对地址转换，并没有取地址的数，也就是说v5还是地址，而且是以char类型8位为单位的地址。而且后7位刚好是input_flag开始的24个字节范围的末尾，就是我们输入24字节flag的最后一个。(一个字符8位)

PS：这里之所以是v11地址的下面7位处是因为这是在main函数内的栈，栈是从下到上的(从高地址到底地址)，栈变量都是从第一个分配地址往下划分内存的，后面会再讲~

do

{

v6 = *v5--; // 从v5开始取，然后减一个地址(一次减8位)，v5现在是input_flag的起始地址开始的24字节的末尾指针了，也就是指向用户输入字符串的最后一个字符，这里循环24次，符合前面我们输入input_flag长度为24的判断条件。

byte_40336C[v4++] = v6; //这里把v6的值也就是我们inpu_flag末尾位置的值开始，一个个赋值给v4开头，这就造成了v4是我们输入24位flag的反向字符，后续对v4数组操作也是对用户输入的24字符flag的反向操作。

}

while (v4 < 24);

for (i = 0; i < 24; ++i)

byte_40336C[i] = (byte_40336C[i] + 1) ^ 6;// 赋完值之后有对自身进行异或操作，进一步修改，注意这里是我们输入的24位字符的反向数组

v8 = strcmp(byte_40336C, aXircjR2twsv3pt); // 异或完后简单的比较，aXircjR2twsv3pt双击跟踪后一个被/x7f截断的字符串，前18位是xlrCj~<r|2tWsv3Ptl，发现不满足24位后再查看才发现后面还有，第19位是/x7f，20~24是zndka。这里留个心眼，/x7f可以阻断字符串

if (v8)

v8 = v8 < 0 ? -1 : 1;

if (!v8)

{

puts("right\n");

system("pause");

}

}

return 0;

}

分析完了，附上图回顾一下以前犯错的思路，给自己日后提个醒：

犯下第一个错误是对关键数组地址修改的地方不敏感，一开始我只看到了对v4数组异或的代码，没有注意到前面的反序操作：

for (i = 0; i < 24; ++i)

byte_81336C[i] = (byte_81336C[i] + 1) ^ 6;

结果逆向逻辑出来后的flag是反的，大概长这样：}NsDkw9sy3qPto4UqNx{galf，可能还是能看出来是反的flag，单要是换其他字符串就不一定了。所以我们应该要注意前面还有对v4操作的代码，也要分析：(分析在前面)

if (v3 >= 16 && v3 == 24)

{

v4 = 0;

v5 = (char *)&v11 + 7;

do

{

v6 = *v5--;

byte_81336C[v4++] = v6;

}

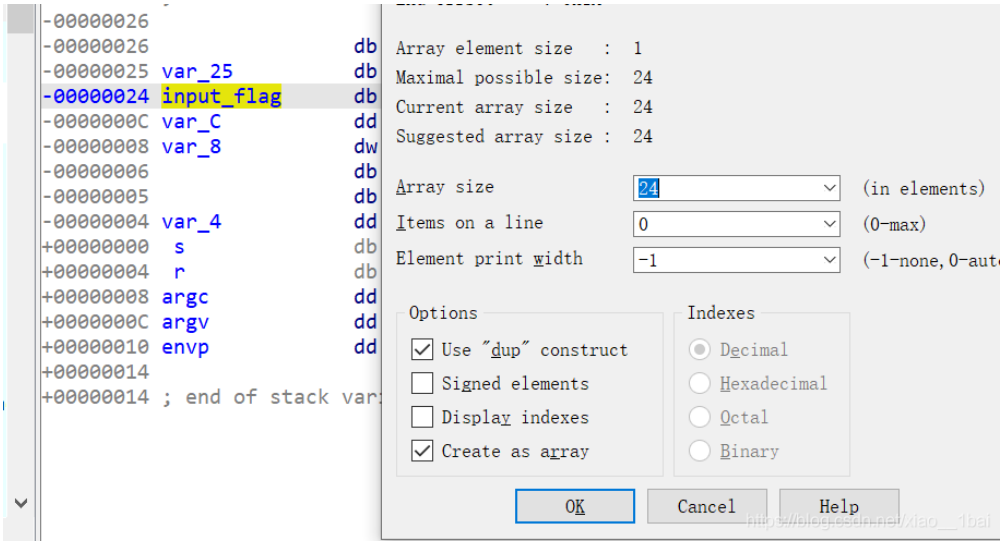
while (v4 < 24);

犯下第二个错误就是对栈不熟练，就是基于对前面v4数组操作的分析，才发现这里有个栈操作让v4数组反向获取用户输入的字符串：

```
-00000024 input_flag      db 16 dup(?)              ; string(
-00000014 var_14          db 8 dup(?)               ; string(
-0000000C var_C           dd ?
-00000008 var_8           dw ?
-00000006                 db ? ; undefined
-00000005                 db ? ; undefined
-00000004 var_4           dd ?
+00000000  s              db 4 dup(?)
+00000004  r              db 4 dup(?)
```


之前在IDA权威指南中了解过栈视图，这里v11是var_14，input_flag就是我们输入24位字符串的首地址，这里给了一个混淆就是v11，前面v5 = (char *)&v11 + 7;就是在v11地址往下取7位char类型，就是0D地址了，刚好在var_C前面。

input_flag首地址24到var_C前面0D处就是完整的24位input_flag地址，所以v5就是取input_flag最后一个字符，这里v11的过渡作用混淆了我，我们可以直接在栈中把v11删除，改input_flag为24位字符串，这里也就解释了v4数组取input_flag反向字符的原因了：



犯下第三个错误是在写脚本中发现的，减号的优先级高于^符号：

下面脚本中 flag+=chr((ord(data[i]) ^ 6)-1) #要是写成chr((ord(data[i]) ^ 6 - 1)那就GG了，由于优先级不同所以结果会不同，给的警示是最好什么都用括号括起来，毕竟这种优先级问题是很难发现的，还以为是自己逻辑梳理错误呢。

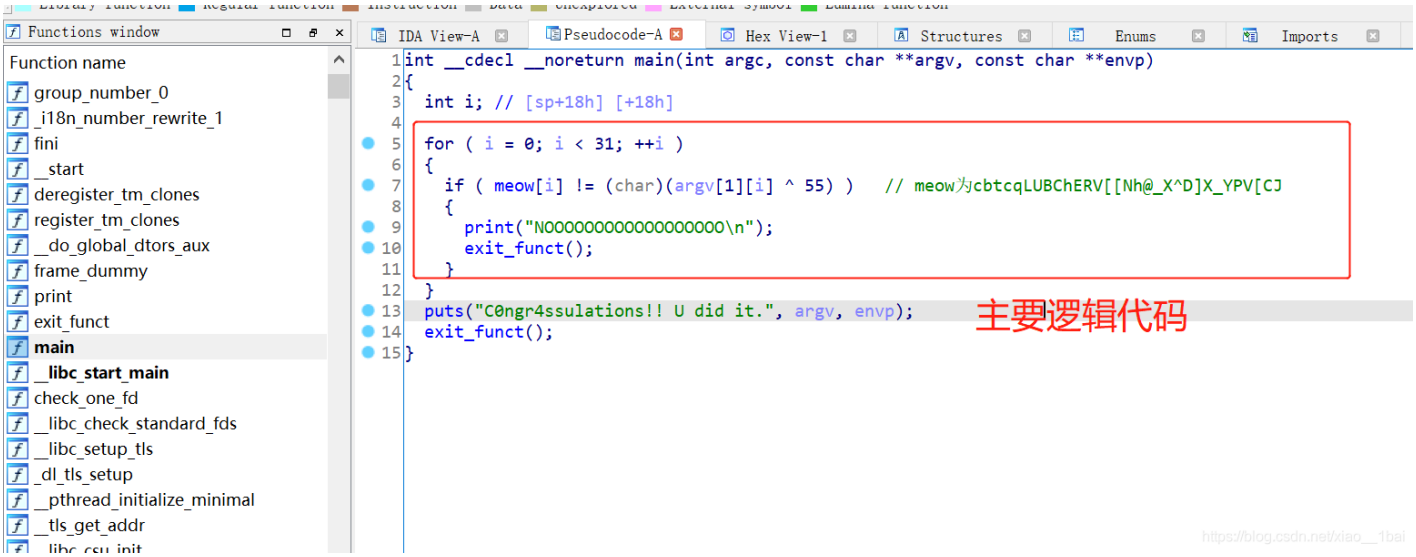
```
data="xlrCj~<r|2tWsv3Ptl\x7Fzndka"

flag=""

for i in range(24):

flag+=chr((ord(data[i])^6)-1)

print(flag)

print(flag[::-1])
```

结果：

```
└─$ python 2.py
}NsDkw9sy3qPto4UqNx{galf
flag{xNqU4otPq3ys9wkDsN}
```

攻防世界re-for-50-plz-50：

32位ELF文件，无壳，照例扔入IDA32中查看伪代码，有main函数看main函数：(图中有点注释。不过不影响)



meow双击跟踪是cbtcqLUBChERV[[Nh@_X^D]X_YPV[CJ ，右边 argv[1][i]是命令行传入的参数：(下面是我以前的笔记)

int main(int argc, char *argv[]) ：

(还可以写成int main(int test_argc, char *test_argv[]))

调用时：

$./a.out testing1 testing2

应当指出的是，argv[0] 存储程序的名称，argv[1] 是一个指向第一个命令行参数的指针，*argv[n] 是最后一个参数。如果没有提供任何参数，argc 将为 1，否则，如果传递了一个参数，argc 将被设置为 2。

所以逻辑很简单，就是传入参数后异或的值与本身存在的数组比较，也就是说题目类型是与用户输入相关的非存储型flag：

key1="cbtcqLUBChERV[[Nh@_X^D]X_YPV[CJ"

flag=""

for i in range(len(key1)):

flag+=chr(ord(key1[i])^55)

print(flag)

```
└─$ python 1.py
TUCTF{but_really_whoisjohngalt}
```

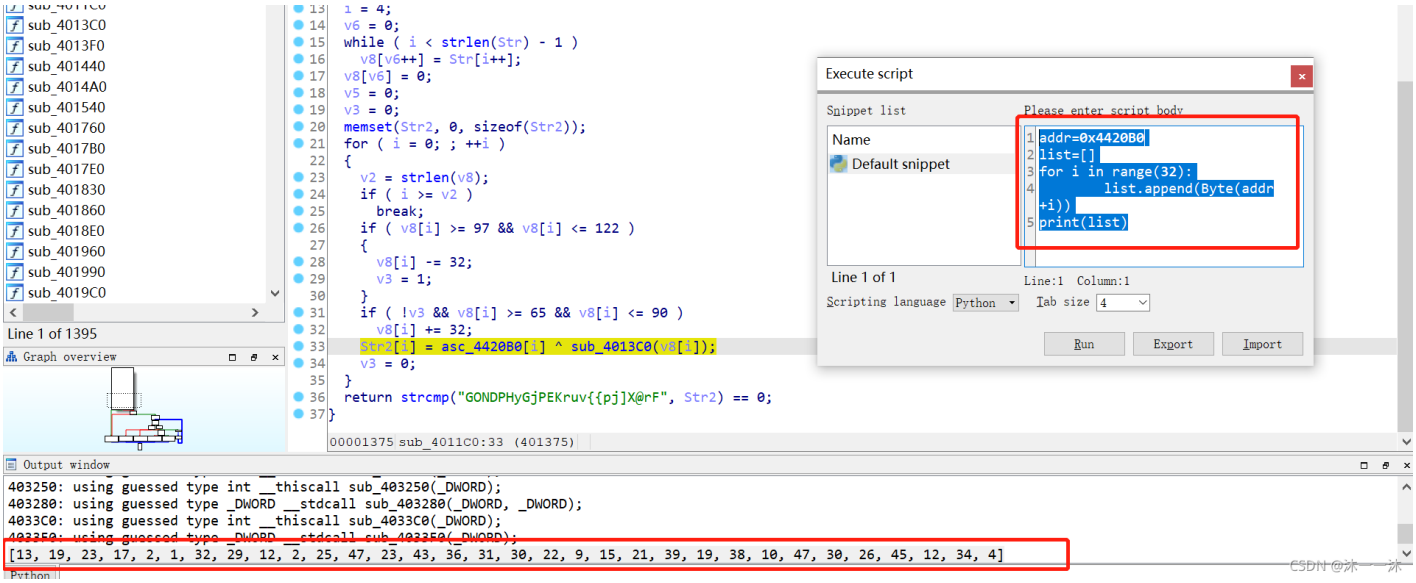
攻防世界IgniteMe：（函数逻辑封装、大小写字符转换算法）

32位windows文件，无壳，照例扔入IDA32中查看伪代码信息，有Mian函数看Main：

由初始信息可以知道，前4个是EIS{ ，最后一个是 }，判断函数在 if 那里，双击跟踪sub_4011C0(Str)函数：

可以看到结果字符串有了，是GONDPHyGjPEKruv{{pj]X@rF，逆向逻辑有了，是简单的一次循环加判断，这里注意一下不带花括号的判断是只判断紧接着的下一条语句而已。

最后这里Str2[i] = asc_4420B0[i] ^ sub_4013C0(v8[i]);用到了asc_4420B0[i]数组来异或，在IDA中嵌入脚本打印一下：



写逆向逻辑脚本：

```
key1="GONDPHyGjPEKruv{{pj]X@rF"

list1=[13, 19, 23, 17, 2, 1, 32, 29, 12, 2, 25, 47, 23, 43, 36, 31, 30, 22, 9, 15, 21, 39, 19, 38, 10, 47, 30, 26, 45, 12, 34, 4]

flag=[]

v3=0

for i in range(len(key1)):

flag.append(((ord(key1[i])^list1[i])-72)^85)

if flag[i] >= 65 and flag[i] <= 90:

flag[i]+=32

elif flag[i] >= 97 and flag[i] <=122:

flag[i]-=32

print(''.join([chr(i) for i in flag]))

print(len(''.join([chr(i) for i in flag]))) #也可以用map(chr,flag)递归装换成字符
```

攻防世界zorropub：（伪随机数加密算法、md5加密/解密算法、代码截断重写、函数积累、exe爆破传参、遍历字符加2算法积累）

64位ELF文件无壳，kali上运行不了，可是由于我的linux只有kali，所以只能结合别人的资料来分析了。照例扔入IDA64中查看伪代码，有main函数看main函数：

```
v15 = __readfsqword(0x28u);

seed = 0;

puts("Welcome to Pub Zorro!!");

printf("Straight to the point. How many drinks you want?");

__isoc99_scanf("%d", &v5);

if ( v5 <= 0 )

{

printf("You are too drunk!! Get Out!!");

exit(-1);

}

printf("OK. I need details of all the drinks. Give me %d drink ids:", (unsigned int)v5);

for ( i = 0; i < v5; ++i )

{

__isoc99_scanf("%d", &v6);
```

```
if ( v6 <= 16 || v6 > 65535 )
{
puts("Invalid Drink Id.");
printf("Get Out!!");
exit(-1);
}
seed ^= v6;
}
i = seed;
v9 = 0;
while ( i )
{
++v9;
i &= i - 1;
}
if ( v9 != 10 )
{
puts("Looks like its a dangerous combination of drinks right there.");
puts("Get Out, you will get yourself killed");
exit(-1);
}
srand(seed);
MD5_Init(v10);
for ( i = 0; i <= 29; ++i )
{
v9 = rand() % 1000;
sprintf(s, "%d", v9);
v3 = strlen(s);
MD5_Update(v10, s, v3);
v12[i] = v9 ^ LOBYTE(qword_6020C0[i]);
}
```

v12[i] = 0;

MD5_Final(v11, v10);

for (i = 0; i <= 15; ++i)

sprintf(&s1[2 * i], "%02x", (unsigned __int8)v11[i]);

if (strcmp(s1, "5eba99aff105c9ff6a1a913e343fec67"))

{

puts("Try different mix, This mix is too sloppy");

exit(-1);

}

return printf("\nYou choose right mix and here is your reward: The flag is nullcon{%s}\n", v12);

}

先分析代码前半部分，输入的v5决定了v6的数量，但是v6又是用于生成seed的，所以v5和v6其实可以联系在一起，都是用于生成seed的。然后生成的seed又要满足条件才可以作为srand的随机数种子，所以这里是一个限制条件。

```
v15 = __readfsqword(0x28u);
seed = 0;
puts("Welcome to Pub Zorro!!");
printf("Straight to the point. How many drinks you want?");
__isoc99_scanf("%d", &v5);
if ( v5 <= 0 )
{
  printf("You are too drunk!! Get Out!!");
  exit(-1);
}
printf("OK. I need details of all the drinks. Give me %d drink ids:", (unsigned int)v5);
for ( i = 0; i < v5; ++i )
{
  __isoc99_scanf("%d", &v6);
  if ( v6 <= 16 || v6 > 65535 )
  {
    puts("Invalid Drink Id.");
    printf("Get Out!!");
    exit(-1);
  }
  seed ^= v6;
}
i = seed;
v9 = 0;
while ( i )
{
  ++v9;
  i &= i - 1;
}
if ( v9 != 10 )
{
  puts("Looks like its a dangerous combination of drinks right there.");
  puts("Get Out, you will get yourself killed");
  exit(-1);
}
srand(seed);
```

输入v5，v5决定了v6的数量，v6要在范围内，然后seed伪随机数种子由v6异或生成

上面生成的seed种子又要在循环中限制v9，且v9要等于v10

然后分析后半部分，这里先补充一些函数和以前积累的知识：

int MD5_Init(MD5_CTX *c)函数：

初始化 MD5 Contex, 成功返回1,失败返回0

int MD5_Update(MD5_CTX *c, const void *data, size_t len)函数：

循环调用此函数,可以将不同的数据加在一起计算MD5,成功返回1,失败返回

int MD5_Final(unsigned char *md, MD5_CTX *c)函数：

输出MD5结果数据,成功返回1,失败返回0

unsigned char *MD5(const unsigned char *d, size_t n, unsigned char *md)函数：

MD5_Init,MD5_Update,MD5_Final三个函数的组合,直接计算出MD5的值

void MD5_Transform(MD5_CTX *c, const unsigned char *b)函数：

内部函数,不需要调用

%02x：

x 表示以十六进制形式输出

02 表示不足两位,，前面补0输出，如果超过两位，则以实际输出。

sprintf(&s1[2 * i], "%02x", (unsigned __int8)v11[i]);的意思是把相当于char的__int8的两位输出到&s1[2*i]中，也就是一次输出两个__int8(char)类型的v11[i]到s1的偶数地址中，所以相当于一个个赋值而已。

继续往后分析：

v9是用特定seed随机数种子生成的特定的列表，v12就是flag。黄框就是特定的v9列表不断拼凑出的md5加密列表，只要v9 md5加密后等于5eba99aff105c9ff6a1a913e343fec67，那么v9与LOBYTE(qword_6020C0[i])异或后就是flag：

```
  srand(seed);
  MD5_Init(v10);
  for ( i = 0; i <= 29; ++i )
  {
    v9 = rand() % 1000;
    sprintf(s, "%d", v9);
    v3 = strlen(s);
    MD5_Update(v10, s, v3);
    v12[i] = v9 ^ LOBYTE(qword_6020C0[i]);
  }
  v12[i] = 0;
  MD5_Final(v11, v10);
  for ( i = 0; i <= 15; ++i )
    sprintf(&s1[2 * i], "%02x", (unsigned __int8)v11[i]);
  if ( strcmp(s1, "5eba99aff105c9ff6a1a913e343fec67") )
  {
    puts("Try different mix, This mix is too sloppy");
    exit(-1);
  }
  return printf("\nYou choose right mix and here is your reward: The flag is nullcon{%s}\n", v12);
}
```

用v10初始化 MD5 Contex, 成功返回1,失败返回0

用特定seed随机数种子生成的特定的列表赋值给v9

将s加到v10中一起计算MD5,成功返回1,失败返回

v12就是flag

输出MD5结果数据

md5加密后与密文比较，成功就输出flag

(这里积累第二个经验)

所以总的逻辑梳理一下，用户输入的两个数生成满足i &= i - 1和v9==10条件的seed种子，然后这个这个符合条件的种子生成特定的v9列表群，最后挑选一个列表md5加密后满足5eba99aff105c9ff6a1a913e343fec67的v9异或LOBYTE(qword_6020C0[i])就是flag：

所以参照别人的博客和理解python subprocess模块后写下自己的脚本：（模块可以对程序输入进行遍历）

（注意：kali是运行不了的，会爆error while loading shared libraries: libcrypto.so.1.0.0: cannot open shared object file libc错误）

模块知识博客地址：python subprocess模块 - lincappu - 博客园

```
import subprocess

c=0

seed=[] #v5和v6都是用于生成seed种子，所以可以合并成一步

for i in range(16,65535,1): #源代码中是先i=seed再验证i的符合性，所以逆向的时候就要先验证i的符合性再seed=i

while(i):

c+=1

i&=i-1

if(c==10):

seed.append(i) #获取符合条件随机数种子列表，种子固定后rand生成的随机数就会固定。

flag=""

for i in seed: #传入符合的i值，其实就是传入一定的seed值，

proc=subprocess.Popen(['./zorropub'],stdin=subprocess.PIPE,stdout=subprocess.PIPE); #用subprocess的Popen方法开启proc子进程并用stdin和stdout设置子进程的输入和输出，并用communicate不断向缓存区传入参数。

out=proc.communicate(('1\n%s\n'%i).encode('utf-8'))[0] #传入参数，第一个传入1即可，第二个传入符合的seed种子，第一个无论传入多少生成的seed都只有一个，所以传入1即可。用.encode('utf-8')属性可以传入字符串参数，不然就要传入bytes类型的参数了

if "nullcon".encode('utf-8') in out:

print(out) #打印符合的输出字符串

print(i) #打印符合的seed值
```

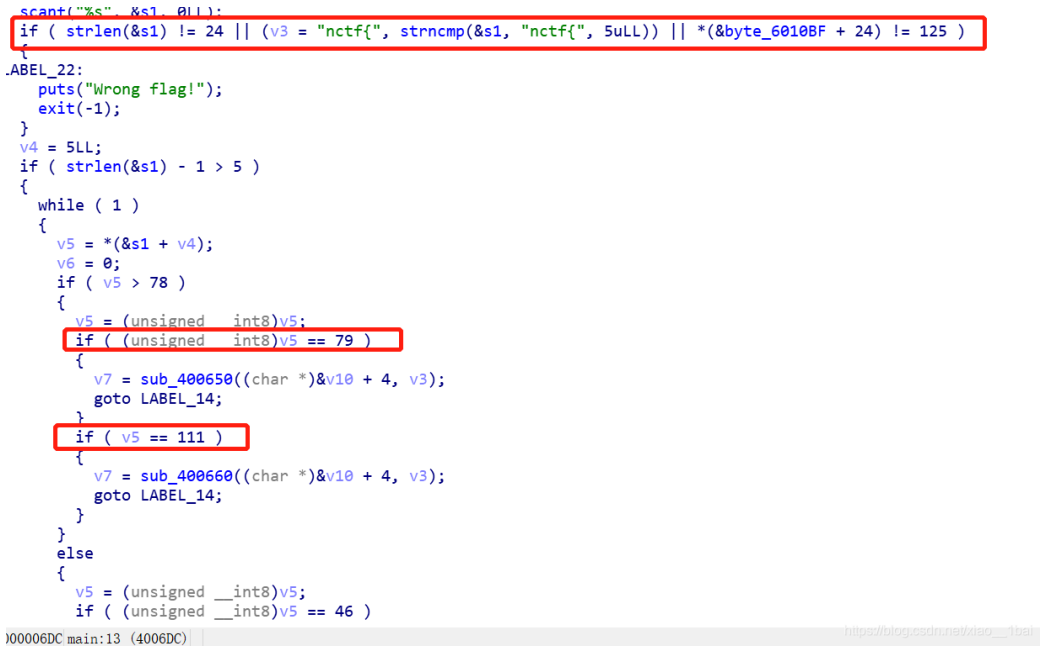
结果：(我运行不了，所以没有结果截图)

nullcon{nu11c0n_s4yz_x0r1n6_1s_4m4z1ng}

main函数与迷宫结合类型：

攻防世界maze：（高低位分割数、函数逻辑封装、迷宫结合）

64位ELF文件，无壳，先扔入IDA中查看伪代码：



从这里犯下第一个错误，我竟然对第一个判断语句的！=125的125不知所云，还去查了ASCII表，对后面的79，46这些竟然也想查。真的得给自己个一巴掌，flag基本都是字符和数字混合，而且在IDA里数字转ASCII字符直接快捷键R啊！！！！

然后判断题目类型是本身就有的存储型flag还是用用户输入一个个生成的生成型flag。答案是后者，那gdb调试就没法用了，直接静态分析代码即可。

转了字符后基本就明白了，现在开始代码分析了：

puts("Input flag:");

scanf("%s", &input_flag, 0LL);

if (strlen(&input_flag) != 24 || strncmp(&input_flag, "nctf{", 5uLL) || *(&byte_6010BF + 24) != '}') //这里要求输入的flag是24个字符，且前5个和最后一个都确定了，一开始的125真的搞得我都不知道啥意思。后面的Oo.0也是如此

{

LABEL_22:

puts("Wrong flag!");

exit(-1);

}

v3 = 5LL;

if (strlen(&input_flag) - 1 > 5)

{

while (1)

{

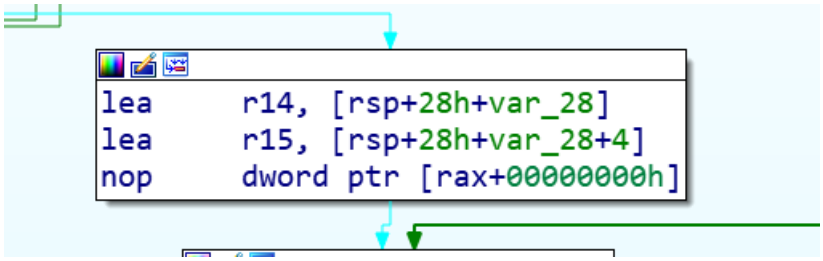
singleflag = *(&input_flag + v3); // 这里v3是从5开始递增的数，目的是从第5个字符开始判断是否符合下述条件

v5 = 0;

```
if ( singleflag > 78 ) //这里给个范围，ASCII码大于78的划为第一类
{
singleflag = (unsigned __int8)singleflag;
if ( (unsigned __int8)singleflag == 'O' ) //如果第一个取O
{
v6 = sub_400650((_DWORD *)&v9 + 1); // 这里犯下第二个错误，64位的v9分成取高底32字节其实是分到r14和r15两个寄存器的，底32位在r14，高32位在r15才有后面根据寄存器的分开操作，因为在两个不同寄存器中。
goto LABEL_14;
}
if ( singleflag == 'o' )//如果第一个取o
{
v6 = sub_400660((int *)&v9 + 1); // 有符号32位高字节操作，r15寄存器，_DWORD就是int就是32位。
goto LABEL_14;
}
}
else
{
singleflag = (unsigned __int8)singleflag;
if ( (unsigned __int8)singleflag == '.' )//如果取到.
{
v6 = sub_400670(&v9); // 无符号底字节32位操作，r14寄存器
goto LABEL_14;
}
if ( singleflag == '0' )
{
v6 = sub_400680((int *)&v9); // 有符号底字节32位，r14寄存器
LABEL_14:
v5 = v6;
goto LABEL_15;
}
}
```

```
LABEL_15:
if ( !(unsigned __int8)sub_400690((__int64)asc_601060, SHIDWORD(v9), v9) )
goto LABEL_22;
if ( ++v3 >= strlen(&input_flag) - 1 ) //在flag范围内v3加1，对应前面singleflag取第6、7、8~个一个个比较
{
if ( v5 ) //如果flag取完了，且sub_这些函数没有返回flase，也就是没有越界，就可以判断是否抵达终点了
break;
LABEL_20:
v7 = "Wrong flag!";
goto LABEL_21;
}
}
}
if ( asc_601060[8 * (signed int)v9 + SHIDWORD(v9)] != '#' ) //判断是否为#这个终点。
goto LABEL_20;
v7 = "Congratulations!";
LABEL_21:
puts(v7);
return 0LL;
}
```

第二个错误看IDA反汇编结构图，底双字在r14寄存器，高双字在r15寄存器：



这里犯下的第三个错误就是对sub_400650、sub_400660、sub_400670、sub_400680、sub_400690、asc_601060、这些IDA自己命名的函数不敢去看！总是觉得自己看不懂，害怕！！！后来才发现其实不应该害怕的！！要逼自己一把！！！

```
bool __fastcall sub_400650(_DWORD *a1)
{
int v1; // eax
```

```
v1 = (*a1)--;
return v1 > 0;
}
bool __fastcall sub_400660(int *a1)
{
int v1; // eax
v1 = *a1 + 1;
*a1 = v1;
return v1 < 8;
}
bool __fastcall sub_400670(_DWORD *a1)
{
int v1; // eax
v1 = (*a1)--;
return v1 > 0;
}
bool __fastcall sub_400680(int *a1)
{
int v1; // eax
v1 = *a1 + 1;
*a1 = v1;
return v1 < 8;
}
```

这四个函数点开之后是对传入参数+1 -1操作而已，真的不难，而且附带返回的比较后来查资料说是判断有没有越出迷宫边界，false就是越出了，就不用玩了，为true就是没越出，继续玩。

```
(unsigned __int8)sub_400690((__int64)asc_601060, SHIDWORD(v9), v9) //主函数中的样式
__int64 __fastcall sub_400690(__int64 a1, int a2, int a3) //双击后中的函数样式
{
__int64 result; // rax
result = *(unsigned __int8 *)(a1 + a2 + 8LL * a3);
LOBYTE(result) = (_DWORD)result == ' ' || (_DWORD)result == '#';
```

return result;

}

这里sub_400690点进去分析后的(__int64)asc_601060如图是一串字符串，后来知道了是迷宫的图，sub_400690函数里传入v9的有符号高双字r15寄存器，和v9底双字的r14寄存器，然后运算表达式result = *(unsigned __int8)(a1 + a2 + 8LL * a3); 就是在asc_601060字符串数组内取字符而已。

可以看出a3*8，所以这是8个字符为一行，也就是说r14寄存器的底双字表示行，r15高双字表示列，+1-1分别对应着向上向下，向左向右移动。（因为这里把2维的迷宫平铺成1维了，所以向上向下走要变*8才行）

O是左移，o是右移，0是下移，.是上移

所以这里可以写出asc_601060的迷宫图形：

******

* * *

*** * **

** * **

* *# *

** *** *

** *

****

现在分析最后一段：

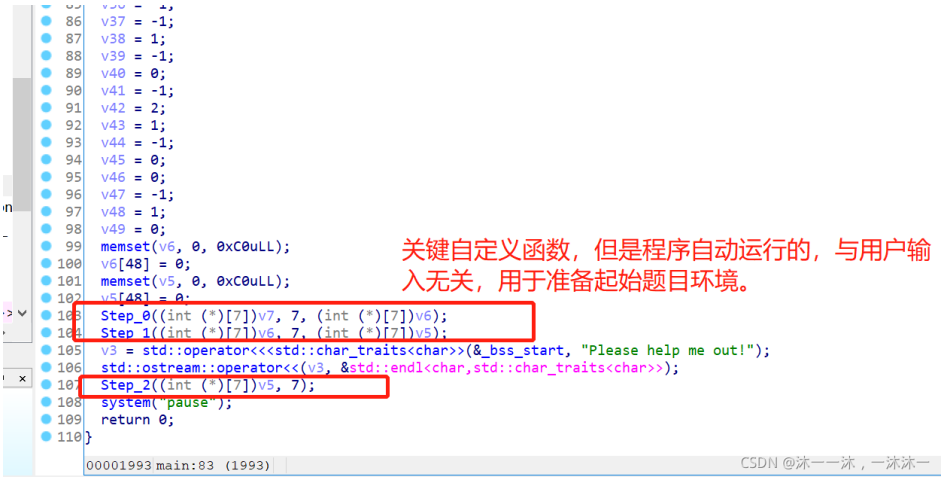
这里就是看最后跳出的flag末尾时是不是到了#这个字符，如果是就表示通关。

所以是：右下右右下下左下下下右右右右上上左左

就是o0oo00O000oooo…OO

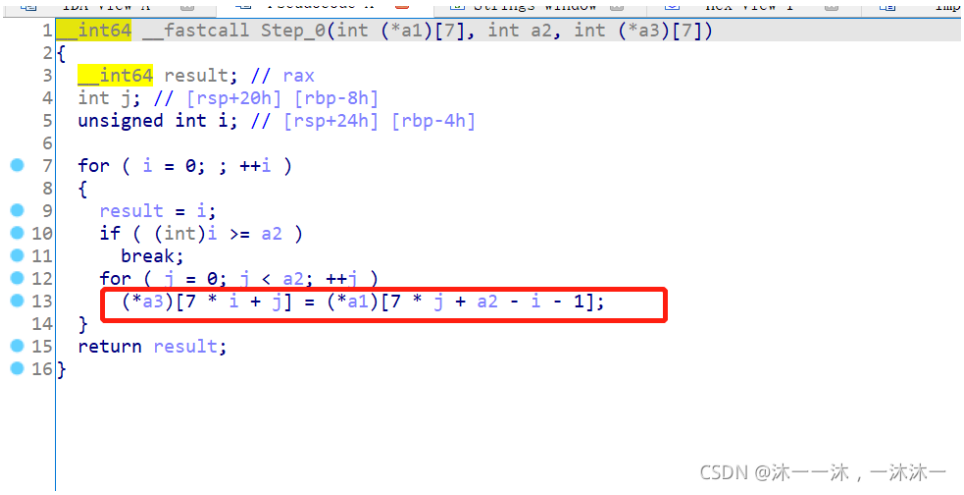
攻防世界easy_Maze：（迷宫结合、地址连续小数组、题目描述暗示、环境准备函数、IDA动态调试、GDB动态调试、IDA的Hex View图热键）

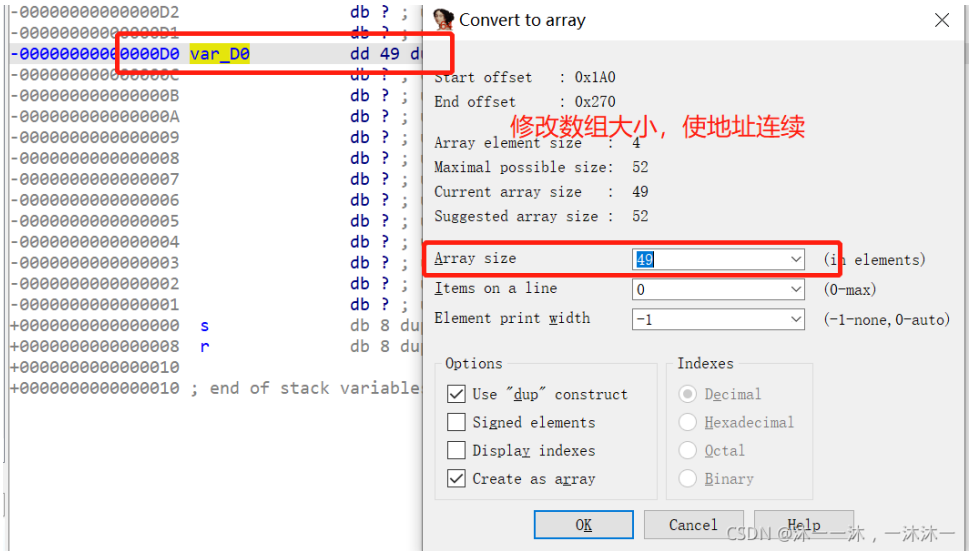
64位ELF文件无壳，照例扔入IDA中查看伪代码信息，有main函数看main函数：



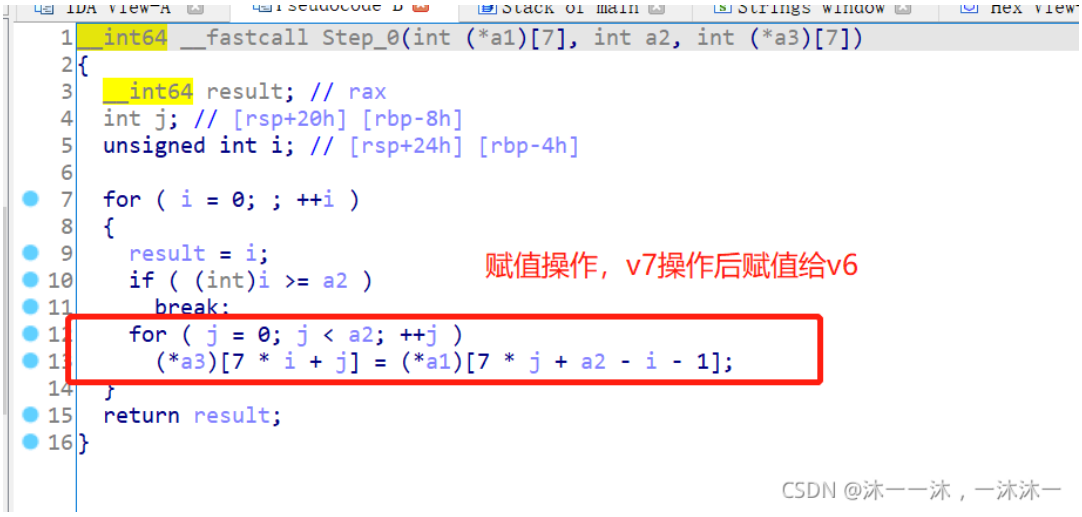
这里积累第一个经验：一进去看到一堆变量，三个自定义函数，跟踪进step_0看看代码：

发现对v7有超过[7]下标的操作，怀疑主函数中一堆变量是v7的数组分出来的，看了一下堆栈，发现是连续的，于是调整主函数栈中局部变量构造：





结合题目暗示maze迷宫类型，开始分析三个step函数，第一个step_0函数，第一个红框显示是程序自运行操作，大概是把v7数组按条件给到v6数组：



第二个step_1函数，也是简单的v6条件赋值给v5，但是这里中间经过两个函数。

```
1  __int64 __fastcall Step_1(int (*a1)[7], int a2, int (*a3)[7])
2 {
3   int v5[7]; // [rsp+20h] [rbp-D0h] BYREF
4   int v6; // [rsp+E4h] [rbp-Ch]
5   int j; // [rsp+E8h] [rbp-8h]
6   int i; // [rsp+ECh] [rbp-4h]
7
8   v6 = getA(a1, a2);
9   if ( !v6 )
10    return 0LL;
11  getAStart(a1, a2, (int (*)[7])v5);
12  for ( i = 0; i < a2; ++i )
13  {
14    for ( j = 0; j < a2; ++j )
15      (*a3)[7 * i + j] = v5[7 * i + j] / v6;
16  }
17  return 1LL;
18 }
```

又是赋值操作，还涉及两个迷宫算法的自定义函数getA和getAStart

跟踪第一个函数getA，是三层循环嵌套加递归。

```
9   unsigned int v9; // [rsp+ECh] [rbp-4h]
10
11  if ( a2 == 1 )
12    return (*a1)[0];
13  v9 = 0;
14  memset(v4, 0, 0xC0uLL);
15  v4[48] = 0;
16  for ( i = 0; i < a2; ++i )
17  {
18    for ( j = 0; j < a2 - 1; ++j )
19    {
20      for ( k = 0; k < a2 - 1; ++k )
21      {
22        if ( k < i )
23          v3 = k;
24        else
25          v3 = k + 1;
26        v4[7 * j + k] = (*a1)[7 * j + 7 + v3];
27      }
28    }
29    v5 = getA((int (*)[7])v4, a2 - 1);
30    if ( (i & 1) != 0 )
31      v9 -= v5 * (*a1)[i];
32    else
33      v9 += v5 * (*a1)[i];
34  }
35  return v9;
36 }
00001226 Z4getAPA7_ii:9 (1226)
```

三层嵌套加递归

跟踪getAStart函数，也是三层嵌套，到时其中还调用getA函数。

```
      for ( j = 0; j < a2; ++j )
      {
        for ( k = 0; k < a2 - 1; ++k )
        {
          for ( l = 0; l < a2 - 1; ++l )
          {
            if ( k < i )
              v4 = 7LL * k;
            else
              v4 = 7LL * (k + 1);
            v5 = &(*a1)[v4];
            if ( l < j )
              v6 = l;
            else
              v6 = l + 1;
            v10[7 * k + l] = v5[v6];
          }
        }
        v7 = &(*a3)[7 * i];
        v8 = getA((int (*)[7])v10, a2 - 1);
        v7[i] = v8;
        if ( (j + i) % 2 == 1 )
          (*a3)[7 * j + i] = -(*a3)[7 * j + i];
      }
    }
  }
  return result;
}
```

有时三层嵌套加递归

这里积累第二个经验：

一开始受前面一道EASYHOOK题目的影响，我以为这里会有类似的虚实代码替换。跟踪进去后发现step_1不断嵌套而且没有系统函数替换进程地址，判断不是HOOK。但是由于step_1函数不断的嵌套，而且是那种循环算法的嵌套，我又以为是Newbie_calculations这类有冗余代码代码需要自己简化函数的题目，结果发现也是错的，因为Newbie_calculations的冗余代码作用是消耗时间，而这题也没有消耗时间。

最后查了资料我才明白，这种step_0和step_1代码的确是算法，但是它们在用户输入命令之前，也就是说这些是系统自执行代码，是为程序渲染环境做前期准备用的，没有必要弄懂它。比如这里step_0和step_1是为程序做迷宫地图的，设计迷宫算法，弄懂它与解题没有太大关系，而且前期准备环境的算法函数也不会有故意出错的地方来设考点，毕竟考点是走迷宫。

理解完这些之后我们看step_2，发现输入cin 函数方法在这里，那么我们只要观察输入函数cin之后的逻辑和在内存中截取出前面step_0和step_1函数自动运算生成的地图来参考即可：

```
  while ( v8 <= 29 && (*a1)[7 * v10 + v9] == 1 )
  {
    std::operator>><char,std::char_traits<char>>(&std::cin, &v7);
    v1 = v8++;
    v6[v1] = v7;
    if ( v7 == 'd' )
    {
      ++v9;
    }
    else if ( v7 > 'd' )
    {
      if ( v7 == 's' )
      {
        ++v10;
      }
      else
      {
        if ( v7 != 'w' )
          goto LABEL_14;
        --v10;
      }
    }
    else if ( v7 == 'a' )
    {
```

走迷宫步数为29步

输入函数

d和a字符走一步，打横走，因为+v9

s字符和w字符走一步，打竖走，因为7*v10

上图第一个框标识走够29步，而且每一步在数组中对应的都是1才行。第二个框是cin输入框，是我们定位的关键，后面的框就是打游戏常见的aswd这样来移动，移动到1上面才行。

所以我们现在要在内存中截断出step_0和step_1前期自动运行出的地图出来即可。

下面图中第一个框表示制作的是7*7的地图。(结合我们前面粗略看得step逻辑也可以判断出来出入7是一边界，我们的v7也是49个元素的数组)

这里积累第三个经验：

第二个框表示送入v5作为最终地图，值得注意的是这个v5是int型的，v5也是数组，但是我们查看内存时是以1byte为单位的，所以我们要转成4bytes来看内存才行。

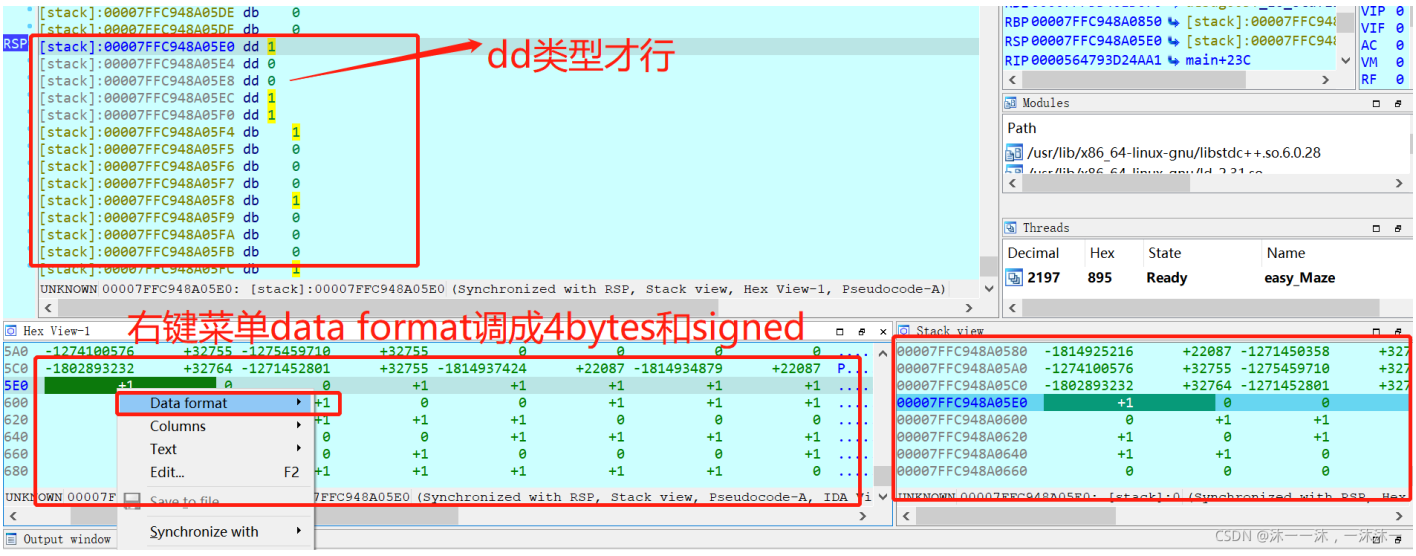
```
57   memset(v6, 0, 0xC0uLL);
58   v6[48] = 0;
59   memset(v5, 0, 0xC0uLL);
60   v5[48] = 0;
61   Step_0((int (*)[7])v7, 7, (int (*)[7])v6);
62   Step_1((int (*)[7])v6, 7, (int (*)[7])v5);
63   v3 = std::operator<<<std::char_traits<char>>(&_bss_start, "Please help me out!");
64   std::ostream::operator<<(v3, &std::endl<char,std::char_traits<char>>);
65   Step_2((int (*)[7])v5);
66   system("pause");
67   return 0;
68 }
```

int类型的4byte注意，单位对准了才有对的地图。

第一种方法：

IDA直接调试，在step_2处下断点，IDA中直接双击查看v5，转4bytes，取49个单位77排列。下面三个框描述的都是一样的数据，记住要转32位int型，即dd类型！按77排列即可，可以看出第一排是1001111，后面继续往下看。

(这里用右键菜单data format调成4bytes和signed即可用±1简化显示。这里有个坑，我现在选择的是8列一行，但是如果选择auto自动或4列一行的话就会出错，好像是隐藏了一些列，然后导致整个地图就画错了，这里注意一下。)



总的地图就是：(借别人的图)

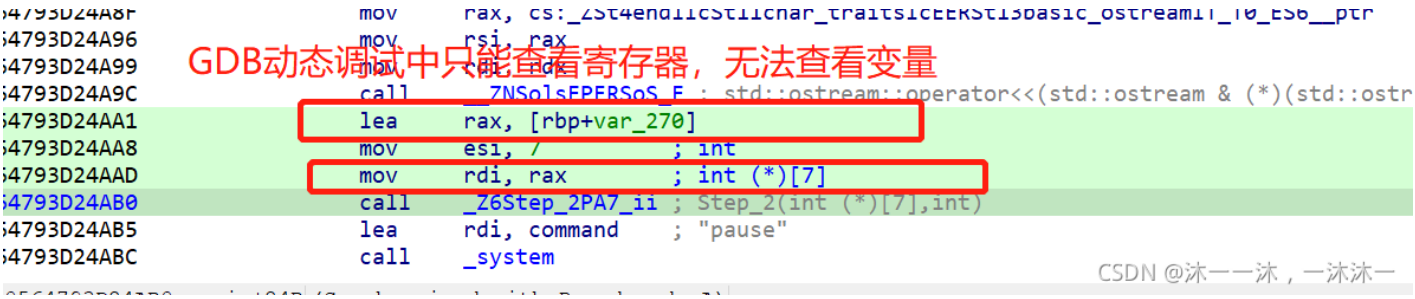
	A	B	C	D	E	F	G
1	1	0	0	1	1	1	1
2	1	0	1	1	0	0	1
3	1	1	1	0	1	1	1
4	0	0	0	1	1	0	0
5	1	1	1	1	0	0	0
6	1	0	0	0	1	1	1
7	1	1	1	1	1	0	1
8							

那么每一步走到1上面就是：

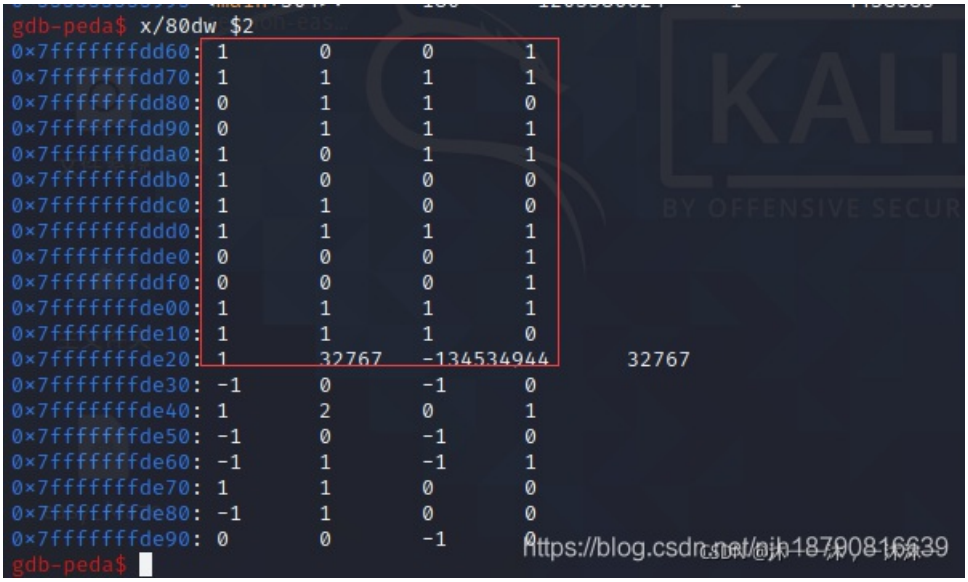
ssddwdwdddssaasasaaassddddwdds

第二种方法GDB动态调试：

首先看汇编代码，v5给了eax寄存器，那么有确定的内存位置就可以直接打印内存了，如果不知道v5放在哪里的话是没办法打印内存的。这里v5给了rax和rdi，所以两个都可以查。



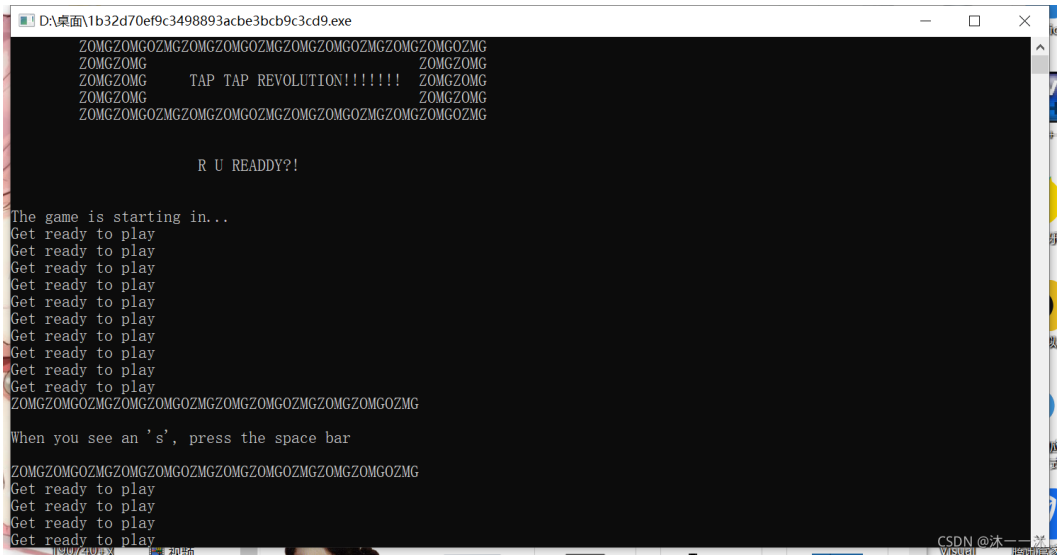
根据上图的汇编地址，断点断在564793D24AB0即可：(这是别人的图，我自己找不到了，知道用./80dw命令显示rax寄存器即可，w是双字，d是整数打印。)



main函数与游戏结合类型：

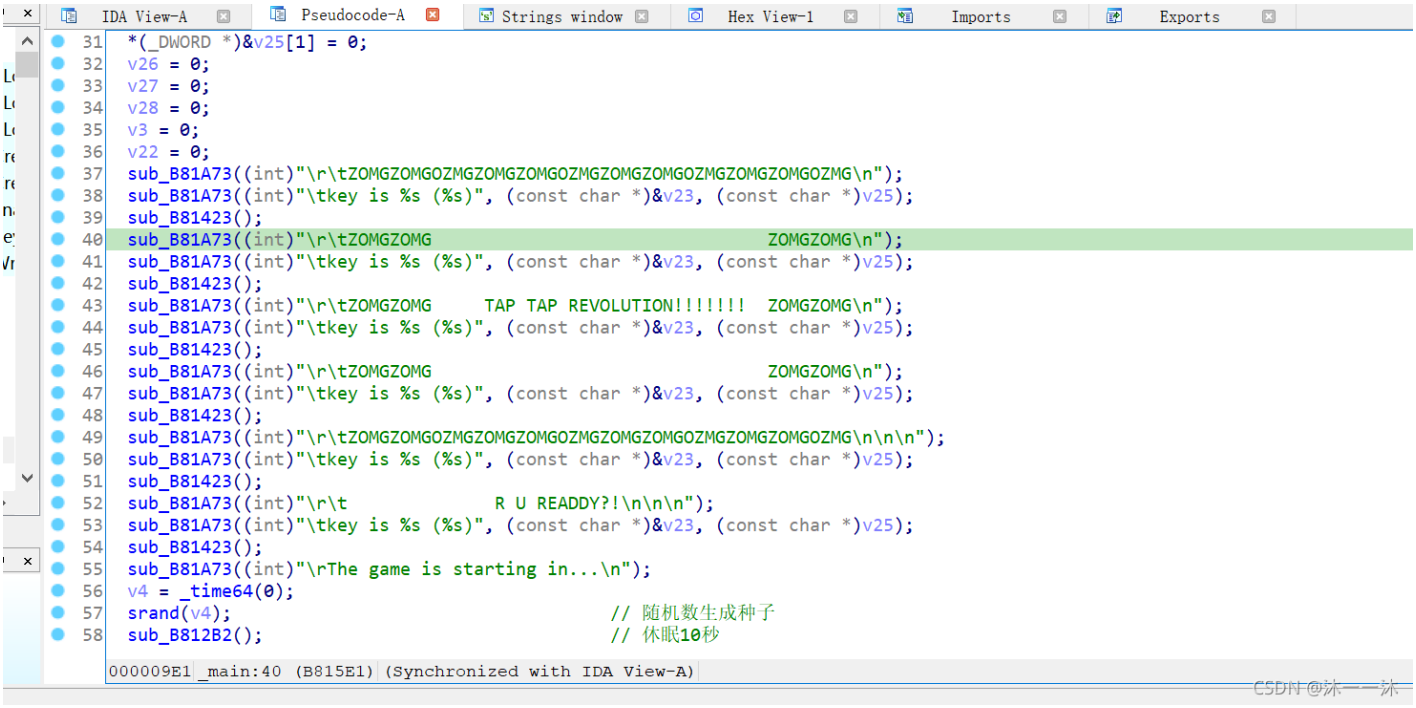
攻防世界gametime：（游戏通关生成flag、）

32位无壳，运行一下程序看看主要信息：



说实话我一开始没看懂怎么玩，所以扔入IDA32中查看伪代码信息，有main函数看Main函数：

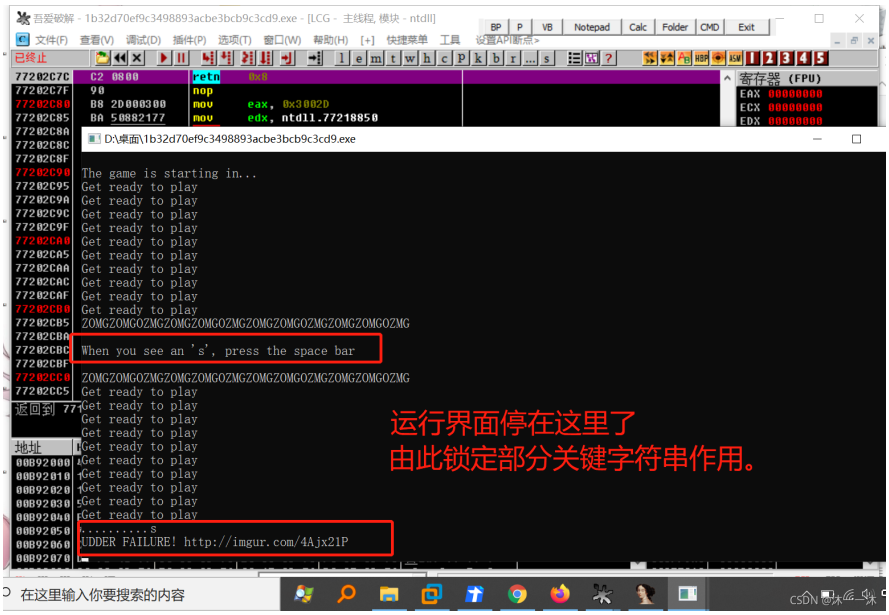
哇，眼花缭乱，代码太多了。



这里积累第一个经验，游戏题一定要玩懂才行：

没那么难玩的，如果游戏文字跳转太快看不清，很难玩，就看着反汇编代码来玩。用OD等动态调试器在游戏结束时保持最后界面，以此来用最后结束时的界面信息根据伪代码判断在哪里退出的，从而找到第一个判断函数。

上OD动态调试：



终于可以看清游戏规则了，出现s就按空格，不然就退出：(后面还有按x和m的)，在IDA伪代码中查看对应信息：

```
49  sub_B81A73((int)"\r\tZOMGZOMGOZMGZOMGZOMGOZMGZOMGZOMGOZMGZOMGZOMGOZMG\n\n\n");
50  sub_B81A73((int)"\tkey is %s (%s)", (const char *)&v23, (const char *)v25);
51  sub_B81423();
52  sub_B81A73((int)"\r\t              R U READDY?!\n\n\n");
53  sub_B81A73((int)"\tkey is %s (%s)", (const char *)&v23, (const char *)v25);
54  sub_B81423();
55  sub_B81A73((int)"\rThe game is starting in...\n");
56  v4 = _time64(0);
57  srand(v4);                                    // 随机数生成种子
58  sub_B812B2();                                 // 休眠10秒
59  sub_B812D5(200u);                             // 10秒输出10个Get ready to play\n
60  if ( !sub_B81435(0x1F4u, 32, 10, v5, (const char *)v25, (const char *)&v23) )
61    return 0;
62  if ( !sub_B81435(0x12Cu, 120, 8, v6, (const char *)v25, (const char *)&v23) )
63    return 0;
64  if ( !sub_B81435(0x12Cu, 109, 5, v7, (const char *)v25, (const char *)&v23) )
65    return 0;
66  sub_B81A73((int)"key is %s (%s)", (const char *)&v23, (const char *)v25);
67  sub_B81A73((int)"\rTRAINING COMPLETE!                                  \n");
68  v8 = Sleep;
69  v9 = 20;
70  do
71  {
72    Sleep(0xC8u);
73    sub_B81A73((int)"\n");
```

终于在众多代码中找到判断函数了，双击跟踪sub_B81435


```
6   sub_B81423();
7   sub_B81A73((int)"\rZOMGZOMGOZMGZOMGZOMGOZMGZOMGZOMGOZMGZOMGZOMGOZMG\n");
8   if ( a2 == 32 )
9     sub_B81A73((int)"\nWhen you see an 's', press the space bar\n\n");
10  else
11    sub_B81A73((int)"\nWhen you see an '%c', press the '%c' key\n\n", a2, a2);
12  sub_B81A73((int)"key is %s (%s)", a6, a5);
13  sub_B81423();
14  sub_B81A73((int)"\rZOMGZOMGOZMGZOMGZOMGOZMGZOMGZOMGOZMGZOMGZOMGOZMG\n");
15  sub_B812D5(a1);
16  v8 = a3;
17  if ( a3 > 0 )
18  {
19    do
20    {
21      sub_B81A73((int)".");                     // 这里输出点.  所以出现的字符判断在它后面
22      Sleep(200u);
23      --v8;
24    }
25    while ( !v8 );
26  }
27  if ( !(unsigned __int8)panduan(a2, 100000) )  // 这里是判断语句，因此我改了名
28    return 1;
29  sub_B81A73((int)"key is %s (%s)\r", a6, a5);
30  sub_B81423();
31  sub_B81A73((int)"\rUDDER FAILURE! http://imgur.com/4Ajx21P \n");
32  return 0;
33}
```

跟踪函数过程中又找到判断函数

000008D1 sub_B81435:27 (B814D1) (Synchronized with IDA View-A)

结合刚才结束界面的回显信息，进一步缩小了判断函数的范围。

然后这里积累第二个经验：

游戏类题目，有些是存储型flag，就是flag本来就在那里，你解出游戏就会显示。而有一些是与用户输入相关的生成型flag，就是用户通关的每一步影响着flag的生成，比如通一关给一部分flag这样。

这道题明显是后者，但是生成型flag中又要看输入到底怎么影响flag生成，如果是那种以通关数生成flag的话，我们改一下判断条件就可以全部通关了。但如果是那种通关的时候要靠用户输入字符，并考输入的对应字符来生成甚至是加密后再生成一部分flag的话，这种题就要一个个找到对应的通关字符然后再逆向逻辑才行。

而这题比较简单，是只判断通关数即可生成flag，为什么我会知道呢，其实我猜的。(笑~) 所以我们用OD修改判断条件即可。

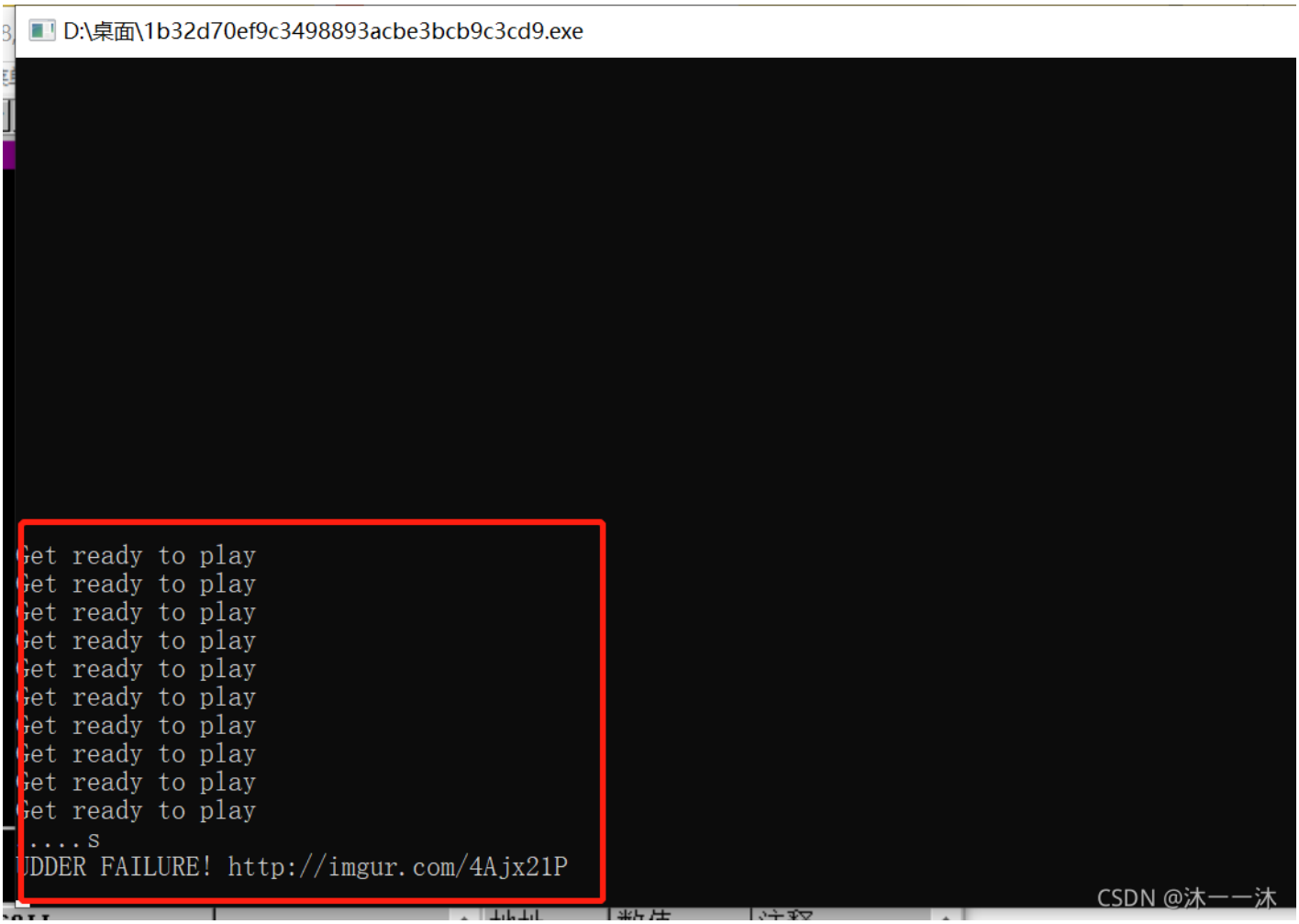
看判断函数的反汇编代码：

```
.text:00B814CA
.text:00B814CA loc_B814CA:
.text:00B814CA mov     edx, 186A0h
.text:00B814CF mov     ecx, esi
.text:00B814D1 call    panduan
.text:00B814D6 pop     edi
.text:00B814D7 pop     esi
.text:00B814D8 pop     ebx
.text:00B814D9 test    al, al
.text:00B814DB jz      short loc_B81503
```


在OD中修改对应内存地址的反汇编代码，你也可以直接用IDA修补反汇编代码调试：

```
00B814B9  .  59              pop     ecx
00B814BA  .  68 C8000000     push    0xC8                    ┌Timeout = 200. ms
00B814BF  .  FF15 2021B900   call    dword ptr ds:[<&KERNEL32.Sleep>  └Sleep
00B814C5  .  83EF 01         sub     edi, 0x1
00B814C8  .^ 75 E5           jnz     X1b32d70e.00B814AF
00B814CA  >  BA A0860100     mov     edx, 0x186A0
00B814CF  .  8BCE            mov     ecx, esi
00B814D1  .  E8 8AFDFFFF     call    1b32d70e.00B81260
00B814D6  .  5F              pop     edi
00B814D7  .  5E              pop     esi
00B814D8  .  5B              pop     ebx
00B814D9  .  8488            test    al, al
00B814DB     74 26           je      X1b32d70e.00B81503
00B814DD  .  FF75 10         push    dword ptr ss:[ebp+0x10]
00B814E0  .  FF75 14         push    dword ptr ss:[ebp+0x14]
00B814E3  .  68 707AB900     push    1b32d70e.00B97A70       ASCII "key is %s (%s)■"
00B814E8  .  E8 86050000     call    1b32d70e.00B81A73
00B814ED  .  E8 31FFFFFF     call    1b32d70e.00B81423
00B814F2  .  68 807AB900     push    1b32d70e.00B97A80       ASCII 0D,"UDDER FAILURE"
00B814F7  .  E8 77050000     call    1b32d70e.00B81A73
00B814FC  .  83C4 10         add     esp, 0x10
00B814FF  .  32C0            xor     al, al
00B81501  .  5D              pop     ebp
00B81502  .  C3              retn
```


前面一切正常，因为但是后面出了问题：



前面正常是因为下面三个都是同一个判断函数：

```
57  srand(v4);                                  // 随机数生成种子
58  sub_B812B2();                               // 休眠10秒
59  sub_B812D5(200u);                           // 10秒输出10个Get ready to play\n
60  if ( !sub_B81435(0x1F4u, 32, 10, v5, (const char *)v25, (const char *)&v23) )
61    return 0;
62  if ( !sub_B81435(0x12Cu, 120, 8, v6, (const char *)v25, (const char *)&v23) )
63    return 0;
64  if ( !sub_B81435(0x12Cu, 109, 5, v7, (const char *)v25, (const char *)&v23) )
65    return 0;
66  sub_B81A73((int)"key is %s (%s)", (const char *)&v23, (const char *)v25);
67  sub_B81A73((int)"\rTRAINING COMPLETE!                                    \n");
68  v8 = Sleep;
69  v9 = 20;
70  do
```


后面出错就去后面找，发现还有三个判断函数：

```
102     --v12;
103   }
104   while ( v12 );
105   sub_B812B2();
106   sub_B812D5(100u);
107   if ( !(unsigned __int8)sub_B81507(0xC8u, (int)v25, (int)&v23) )
108     return 0;
109   if ( !(unsigned __int8)sub_B81507(0xC8u, (int)v25, (int)v25) )
110     return 0;
111   if ( !(unsigned __int8)sub_B81507(0xC8u, (int)v25, (int)v25) )
112     return 0;
113   sub_B81A73((int)"key is %s (%s)", (const char *)&v23, (const char *)v25);
114   sub_B81423();
115   sub_B81A73((int)"\rooooh, you fancy!!!\n");
116   if ( !(unsigned __int8)sub_B81507(0xC8u, (int)v25, (int)&v23)
117     || !(unsigned __int8)sub_B81507(0xC8u, (int)v25, (int)&v23)
```

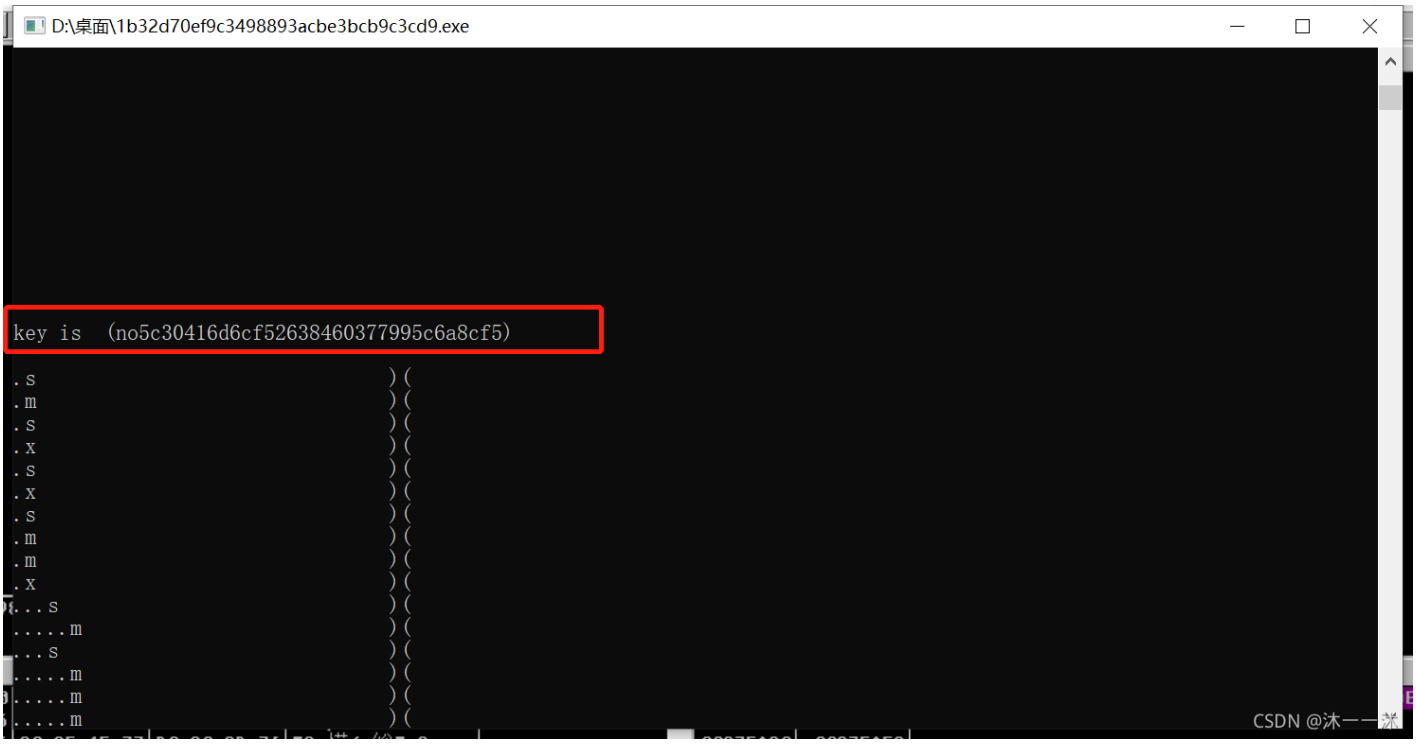

老样子双击跟踪找汇编代码：

```
.text:00B81551
.text:00B81551 loc_B81551:
.text:00B81551 imul    edx, edi, 1F4h
.text:00B81557 mov     ecx, ebx
.text:00B81559 call    panduan
.text:00B8155E pop     edi
.text:00B8155F pop     esi
.text:00B81560 pop     ebx
.text:00B81561 test    al, al
.text:00B81563 jnz     short loc_B81586
```



```
00B8154F  .^ 75 E9          jnz     X1b32d70e.00B8153A
00B81551  > 69D7 F4010000   imul    edx, edi, 0x1F4
00B81557  . 8BCB            mov     ecx, ebx
00B81559  . E8 02FDFFFF     call    1b32d70e.00B81260
00B8155E  . 5F              pop     edi
00B8155F  . 5E              pop     esi
00B81560  . 5B              pop     ebx
00B81561  . 84C0            test    al, al
00B81563  . 74 21           je      X1b32d70e.00B81586
00B81565  . FF75 0C         push    dword ptr ss:[ebp+0xC]
00B81568  . FF75 10         push    dword ptr ss:[ebp+0x10]
00B8156B  . 68 707AB900     push    1b32d70e.00B97A70          ASCII "key is %s (%s)■"
00B81570  . E8 FE040000     call    1b32d70e.00B81A73
00B81575  . 68 D07AB900     push    1b32d70e.00B97AD0          ASCII "UDDER FAILURE! http://imgur.c
00B8157A  . E8 F4040000     call    1b32d70e.00B81A73
00B8157F  . 83C4 10         add     esp, 0x10
00B81582  . 32C0            xor     al, al
00B81584  . 5D              pop     ebp
00B81585  . C3              retn
00B81586  > B0 01           mov     al, 0x1
00B81588  . 5D              pop     ebp
00B81589  . C3              retn
00B8158A  $ 55              push    ebp
00B8158B  . 8BEC            mov     ebp, esp
```


继续运行程序，成功输出：



main函数与数学算法结合：

攻防世界notsequence：（杨辉三角算法、函数逻辑封装、IDA对char型(byte)的4*计数）

无壳，32位ELF文件，照例扔入32位IDA中查看伪代码，有Main函数看main函数：

题型是与用户输入有关的生成型flag，逻辑是经过两个check，分析第一个check函数v2 = sub_80486CD(&unk_8049B

E0)：

```
int __cdecl sub_80486CD(int input_flag)
{
  int j; // [esp+0h] [ebp-14h]
  int v3; // [esp+4h] [ebp-10h]
  int i; // [esp+8h] [ebp-Ch]
  int v5; // [esp+Ch] [ebp-8h]

  v5 = 0;
  for ( i = 0; i <= 1024 && *( DWORD *)(4 * i + input_flag); i = v5 * (v5 + 1) / 2 )
  {
    v3 = 0;
    for ( j = 0; j <= v5; ++j )
      v3 += *( DWORD *)(4 * (j + i) + input_flag);
    if ( 1 << v5 != v3 )
      return -1;
    ++v5;
  }
  return v5;
}
```

v3和v5双重循环

v3在input_flag中跳着取。

// v3和v5有关，由v3逆向出flag

这里一开始我没看懂，按照反向逆向逻辑来看双重循环的话我得知道v5的值，但是这里并没有，所以我在主函数处发现了v2=20，但是又不确定v2在第二个check函数里有没有改变过，结果是没有。

所以v5=v2=20，有了v5的值就可以进行反向第一个check中双层循环中的 for (j = 0; j <= v5; ++j)循环了。

```
  }
  if ( v2 == 20 )
  {
    puts("Congratulations! fl4g is :\nRCTF{md5(/*what you input without space or \\n~*/)}");
    exit(0);
  }
  return 0;
}
```

有了最终v2=v5值就可以反向逻辑了

可是之后我还是逆向不了，我知道v3的个数和v5一样，可是v3 += *(_DWORD *)(4 * (j + i) + input_flag)这句代码标识v3是在input_flag中跳着取的啊，那这个逻辑逆向起来就相当麻烦了，我不会。(哭~)

查了资料才发现这是杨辉三角，算法逆向题，没办法，只能跟着wp走了，并附上我自己的见解：

首先这里积累第一个经验，附上杨辉三角解析：

[1] #0 /1 |2^0=1

[1, 1] #1 /2 |2^1=2

[1, 2, 1] #3 /3 |2^2=4

[1, 3, 3, 1] #6 /4 |2^3=8

[1, 4, 6, 4, 1] #10 /5 |2^4=16

[1, 5, 10, 10, 5, 1]

[1, 6, 15, 20, 15, 6, 1]

[1, 7, 21, 35, 35, 21, 7, 1]

[1, 8, 28, 56, 70, 56, 28, 8, 1]

[1, 9, 36, 84, 126, 126, 84, 36, 9, 1]

[1, 10, 45, 120, 210, 252, 210, 120, 45, 10, 1]

.

.

这样来看杨辉三角第一批特征：(n是行号且从0开始)

(1)最左边代表行号，0就是第0行。（行号从0开始）

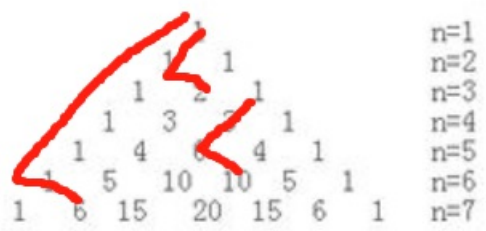
(2)第1个字符在数组（第一行到当前行组成的数组）中的位置，#后的数字。（n*(n+1)/2，n是行号且从0开始）

(3)一行的有几个数字 /后的内容。（n+1，n是行号从0开始）

(4)整行的和。|后的内容，是2的行号次方。(2^n，n是行号且从0开始)

.

.



.

.

这样来看杨辉三角第二批特征：

第n行数字的和为2^（n），行号从0开始

1=2^（0-0），1+1=2^（1-0），1+2+1=2^（2-0），1+3+3+1=2^（3-0），1+4+6+4+1=2^（4-0），1+5+10+10+5+1=2^（5-0）。

.

.

斜线上数字的和等于其向左（从左上方到右下方的斜线），拐角上的数字。(在图中以用红线标好)

1+1=2，1+1+1=3，1+1+1+1=4，1+2=3，1+2+3=6，1+2+3+4=10，1+3=4，1+3+6=10，1+4=5

接下来重新分析check1函数代码：（考察杨辉三角第二批特性中的第n行数字和）

这段代码check1函数的作用是检测每一行求和结果是否为2^k（k从0开始），可以抽象成一个二维结构，有[k]行（第一行k=0），每行开头为第k*(k+1)/2个数。

```
int __cdecl sub_80486CD(int input_flag)

{

int j; // [esp+0h] [ebp-14h]

int v3; // [esp+4h] [ebp-10h]
```

int i; // [esp+8h] [ebp-Ch]

int v5; // [esp+Ch] [ebp-8h]

v5 = 0; //这里积累第二个经验：通过v5的0、1、2、3……然后退出循环中 i 表达式的前几个值0、1、3、6、10……可以发现问题，因为这不是遍历或者有规律的遍历（每次检查第四个），而且v5 * (v5 + 1) / 2 是等差数列的公式，结合前面的逻辑逆向麻烦性，由此要知道考的是算法。

for (i = 0; i <= 1024 && *(_DWORD *)(4 * i + input_flag); i = v5 * (v5 + 1) / 2) //等差数列公式 i，这里4*i应该只是为了迎合int类型，IDA可能默认i是char的byte类型了。

{

v3 = 0;

for (j = 0; j <= v5; ++j) //这里积累第三个经验：在杨辉三角那里，每一行的数的总和等于2的以该行号的次方，行号从0开始算起。

v3 += *(_DWORD *)(4 * (j + i) + input_flag);

if (1 << v5 != v3) // 所以这里v5是行上数的个数，这里1 << v5就表示2的v5次方，就是2的行号次方(从0开始)。v3 += *(_DWORD *)(4 * (j + i) + input_flag)中i是v5行前的杨辉三角的个数，因为我们是一维排列杨辉三角的，所以只能用(4 * (j + i)这种表达式来遍历第v5行上的v5+1个数(杨辉三角行从0开始！)，这里4*i应该只是为了迎合int类型，IDA可能默认i是char的byte类型了。

return -1;

++v5; //v5的0、1、2、3，是杨辉三角对应行上的个数，递增数列。

}

return v5;

}

接着分析check2函数的代码：（考察杨辉三角第二批特性中的斜线上的数字和）

int __cdecl sub_8048783(int input_flag, int k_20)

{

int v3; // [esp+10h] [ebp-10h]

int v4; // [esp+14h] [ebp-Ch]

int i; // [esp+18h] [ebp-8h]

int v6; // [esp+1Ch] [ebp-4h]

v6 = 0;

for (i = 1; i < k_20; ++i) //这里i总0、1、2……这样连续递增

{

v4 = 0;

v3 = i - 1; //这里v3从0、1、2、这样连续递增，

if (!*(_DWORD *)(4 * i + input_flag)) //这里4*i应该只是为了迎合int类型，IDA可能默认i是char的byte类型了。

return 0;

while (k_20 - 1 > v3)

{

v4 += *(_DWORD *)(4 * (v3 * (v3 + 1) / 2 + v6) + input_flag); //这里积累第四个经验：这里等差数列表达式v3 * (v3 + 1) / 2前面说过了，是杨辉三角的第一批特征中第N行前面的个数，v6从0开始递增，表示取杨辉三角v3行的v6列的值，而在这个循环中v3是变换的，也就是取得杨辉三角的行是变化的，而v6在此一个该循环中是固定的，所以可以看成是取每一行(v3)的同一个列(v6)

++v3;

}

if (*(_DWORD *)(4 * (v3 * (v3 + 1) / 2 + i) + input_flag) != v4) //这里由于前面循环++v3后表明行号向下了一行，而 i 从1开始，v6从0开始，所以 i 永远比 v6大1，v6 比 i 多一列。所以这里可以看作[0]-[k-1]行的 [v6] 列求和等于[k]行的 [i]

return 0;

++v6;

}

return 1;

}

所以答案很明显了，是杨辉三角的前20行就是答案，这里积累第5个经验，写杨辉三角生成脚本：(代码标注很详细了，希望对自己日后有帮助！)

def triangles():

s=[1] #这里s[1]作为杨辉三角函数起始值

while True: #无限循环生成杨辉三角

yield s #每次返回一行的杨辉三角列表

s.append(0) #给杨辉三角下一列扩充一个数的空间，因为每一行比上一行多1个

s=[s[i-1]+s[i] for i in range(len(s))] #覆盖生成杨辉三角行列表，满足杨辉三角的下一行的第n个数等于上一行的第n和n-1的和

n=0 #设置计数器，因为只打印前20行

flag=''

for i in triangles(): #每次获取从triangels函数的yield返回的一行列表

#print(i) #打印每一行杨辉三角

flag+=''.join(map(str,i)) #返回通过指定字符连接序列中元素后生成的新字符串，以str为间隔，默认为逗号。而列表就是逗号间隔的

n+=1

if n==20:

break

import hashlib

flag=hashlib.md5(flag.encode()).hexdigest() #这里把flag的列表流变成了字节流，就去掉了列表保留了每个元素了，然后直接加密

print("RCTF{"+flag+"}")

结果：

```
└─$ python 2.py
RCTF{37894beff1c632010dd6d524aa9604db}
```

攻防世界SignIn：（RSA加密/解密算法、函数积累、字符ASCII码做索引、ASCII码表相关、RSA的ASCII字符整数16进制拆分转换算法）

64位ELF文件无壳，照例扔入IDA64中查看伪代码信息，有Main函数看main函数：

```
1  __int64 __fastcall main(int a1, char **a2, char **a3)
2  {
3    char v4[16]; // [rsp+0h] [rbp-4A0h] BYREF
4    char v5[16]; // [rsp+10h] [rbp-490h] BYREF
5    char v6[16]; // [rsp+20h] [rbp-480h] BYREF
6    char v7[16]; // [rsp+30h] [rbp-470h] BYREF
7    char v8[112]; // [rsp+40h] [rbp-460h] BYREF
8    char v9[1000]; // [rsp+B0h] [rbp-3F0h] BYREF
9    unsigned __int64 v10; // [rsp+498h] [rbp-8h]
10
11   v10 = __readfsqword(0x28u);
12   puts("[sign in]");
13   printf("[input your flag]: ");
14   __isoc99_scanf("%99s", v8);
15   sub_96A(v8, v9);
16   __gmpz_init_set_str(v7, "ad939ff59f6e70bcbfad406f2494993757eee98b91bc244184a377520d06fc35", 16LL);
17   __gmpz_init_set_str(v6, v9, 16LL);
18   __gmpz_init_set_str(v4, "10346103590081691412139010129904904441395040517371217043416168653987816098454 9", 10LL);
19   __gmpz_init_set_str(v5, "65537", 10LL);
20   __gmpz_powm(v6, v6, v5, v4);
21   if ( (unsigned int)__gmpz_cmp(v6, v7) )
22     puts("GG!");
23   else
24     puts("TTTTTTTTTql!");
25   return 0LL;
26 }
```

RSA算法相关的系统函数

代码一目了然，就是中间这几个系统函数有点意思，以前说系统函数通常不是考点，但这么多系统函数就有点问题了。上网查一下这些系统函数用法：

__gmpz_init_set_str 其实就是 mpz_init_set_str int mpz_init_set_str (mpz_t rop, const char *str, int base) 函数：

这三个参数分别是多精度整数变量，字符串，进制。 这个函数的作用就是将 str 字符数组以 base 指定的进制解读成数值并写入 rop 所指向的内存。

void mpz_powm (mpz_t rop, const mpz_t base, const mpz_t exp, const mpz_t mod) 函数：

其实就是计算 base 的 exp 次方，并对 mod 取模，最后将结果写入 rop 中， 这个运算的过程和RSA的加密过程一样。

接下来就是__gmpz_cmp函数，看这个函数名就知道这是比较函数。

mpz_cmp(b, c)； //b 大于 c，返回 1；b

等于 c，返回 0；b 小于 c，返回-1*/

重述一下就是：

mpz_powm(op1,op2,op3,op4)； //求幂模函数 即 op1=op2^op3 mod op4;

mpz_init_set_str(b, “200000”, 10)； //即 b=200000，十进制

mpz_cmp(b, c)； //b 大于 c，返回 1；b 等于 c，返回 0；b 小于 c，返回-1*/

真的，整个运算过程和RSA计算一样，但是在此之前先看一下最前面的sub_96A(v8, v9)函数的内容：

```
11  v10 = __readfsqword(0x28u);
12  puts("[sign in]");
13  printf("[input your flag]: ");
14    isoc99_scanf("%99s", v8);
15  sub_96A(v8, v9);
16  __gmpz_init_set_str(v7, "ad939ff59f8e70bcbfad406f2494993757eee98b91bc244184a377520d06fc35", 16LL);
17  __gmpz_init_set_str(v6, v9, 16LL);
18  __gmpz_init_set_str(v4, "10346103590081691412139010129904904441395040517371217043416168653987816090
19  __gmpz_init_set_str(v5, "65537", 10LL);
20  __gmpz_powm(v6, v6, v5, v4);
```

双击跟踪一下该自定义函数作用

CSDN @沐一一沐，一沐沐一

双击跟踪看一下：

```
1  size_t __fastcall sub_96A(const char *a1, __int64 a2)
2  {
3    size_t result; // rax
4    int v3; // [rsp+18h] [rbp-18h]
5    int i; // [rsp+1Ch] [rbp-14h]
6
7    v3 = 0;
8    for ( i = 0; ; i += 2 )
9    {
10     result = strlen(a1);
11     if ( v3 >= result )
12       break;
13     *(_BYTE *)(a2 + i) = byte_202010[a1[v3] >> 4];
14     *(_BYTE *)(a2 + i + 1LL) = byte_202010[a1[v3++] & 0xF];
15   }
16   return result;
17 }
```

字符ASCII码作数组下标

取整数部分

取余数部分

CSDN @沐一一沐，一沐沐一

这里积累第一个经验：

这里发现取两个数组的下标，第一个a1+i 取的是以输入字符除以16后的整数部分为byte_202010下标。

第二个a1+i+1 取的是以输入字符除以16后的余数部分为byte_202010下标。

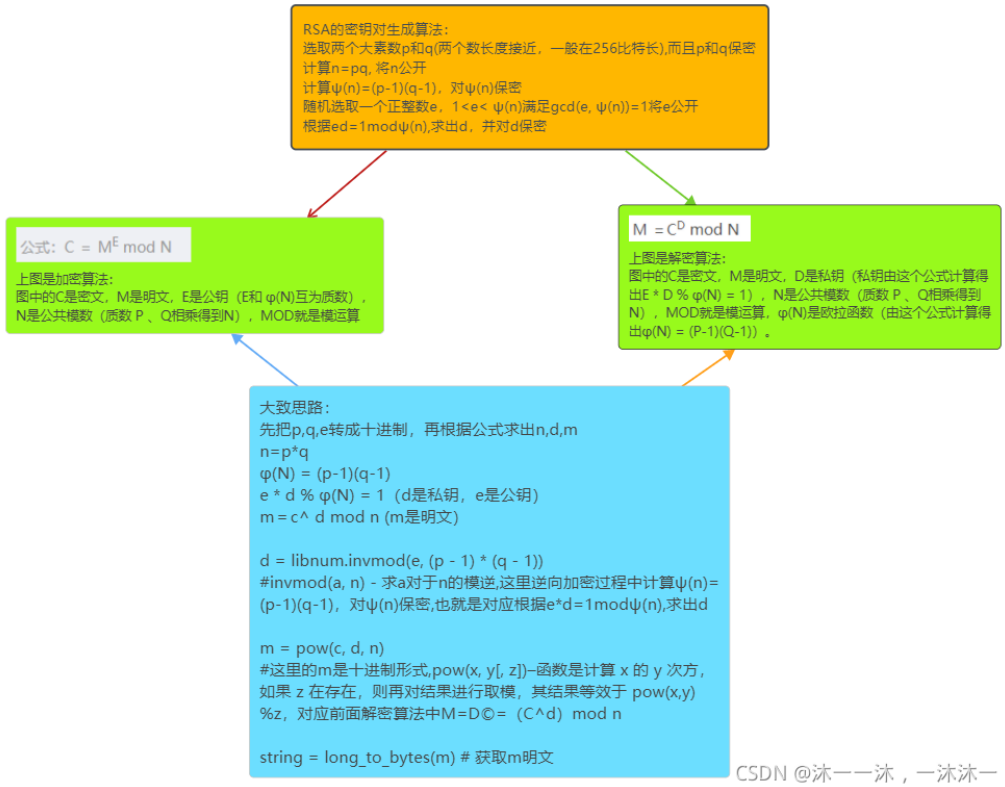
再看一下byte_202010数组内容，是0到f的字符，这是十六进制的基数：(一开始我并没有看出是十六进制的基数~哭)

```
.data:0000000000202010 ; _BYTE a0123456789abcd[16]
.data:0000000000202010 a0123456789abcd db '0123456789abcdef'   ; DATA XREF: sub_96A+47↑o
.data:0000000000202010                                         ; sub_96A+7F↑o
.data:0000000000202010 _data           ends
.data:0000000000202010
.bss:0000000000202020 ; ===========================================================================
```

16进制字符

一个整数一个余数你会发现这是把输入字符变成两个分开的十六进制存储起来，比如输入字符 '1' ，它的整数是49，49除16的整数是3，余数是1，在byte_202010下标中分别对应3和1，构成的31就是字符 '1'的ASCII的十六进制形式，只不过是分开的十六进制，3 1 共两个字节。

然后后面又回顾了RSA的算法，这次分析细致一点，积累起来日后用：



再看我们的原伪代码，有哪些信息：

```
13  printf("[input your flag]: ");
14  __isoc99_scanf("%99s", input_flag);
15  sub_96A(input_flag, (__int64)input_flag2);
16    gmpz_init_set_str(v7, "ad939ff59f6e70bcbfad406f2494993757eee98b91bc244184a377520d06fc35", 16LL);
17    gmpz_init_set_str(v6, input_flag2, 16LL);
18    gmpz_init_set_str(v4, "10346103590081691412139010129904904441395040517371217043416168653987816098454 9", 10LL);
19    gmpz_init_set_str(v5, "65537", 10LL);
20    gmpz_powm(v6, v6, v5, v4);
21  if ( (unsigned int)__gmpz_cmp(v6, v7) )
22    puts("GG!");
23  else
```

16进制转换，明文M

十进制转换，模N

RSA算法

十进制转换，指数E

由 __gmpz_powm(v6, v6, v5, v4)函数可以看出v6是密文，也是我们用于加密的明文，而且这个明文是我们输入flag取前面说的分开的十六进制后的数。

v5是指数=65537

v4是模N=10346103590081691412139010129904904441395040517371217043416168653987816098454 9

C可以由判断语句 if ((unsigned int)__gmpz_cmp(v6, v7))得来，密文C就是 v7=0xad939ff59f6e70bcbfad406f2494993757eee98b91bc244184a377520d06fc35（后面取十六进制）

这里这里考的是我们输入明文后加密出来的密文与v7一样，是常规的RSA解密：

第一种方法，跑以前积累的RSA脚本：

```
e = 65537
c = 0xad939ff59f6e70bcbfad406f2494993757eee98b91bc244184a377520d06fc35
n = 10346103590081691412139010129904904441395040517371217043416168653987816098454 9
```

```
└─$ python2 solve.py --verbose  -i /home/wdnmd/桌面/1.txt
DEBUG: factor N: try past ctf primes
DEBUG: factor N: try Gimmicky Primes method
DEBUG: factor N: try Wiener's attack
DEBUG: Starting new HTTP connection (1): www.factordb.com:80
DEBUG: http://www.factordb.com:80 "GET /index.php?query=10346103590081691412139010129904904
44413950405173712170434161686539878160984549 HTTP/1.1" 200 1000
DEBUG: http://www.factordb.com:80 "GET /index.php?id=1100000001345025700 HTTP/1.1" 200 876
DEBUG: http://www.factordb.com:80 "GET /index.php?id=1100000001345025699 HTTP/1.1" 200 872
DEBUG: d = 0xca9df79a789c4f44873d36b5f28deef25bccbabdfa0c187d37c570034d3f2cbdL
INFO: suctf{Pwn_@_hundred_years}
```


第二种方法，自己写RSA解密脚本：(用我之前注释的脚本算了，因为有一些密码学的库不了解)

这里积累第二个经验：

首先根据解密算法我们现在手上有C密文和N模数，但是没有D。而D的算法是e*d=1modψ(n)的逆运算，e我们有，ψ(n)=(p-1)(q-1)也可以通过大数分解由N分出p、q。最后逆运算函数就是libnum.invmod(e, (p - 1) * (q - 1))即可算出D。

D算出来后明文M就可以算出来了，用函数m = pow(c, d, n) 算出M，但是我们的M是前面经过转换拆分的两个单字符十六进制数，这其实是刚好满足RSA最后的明文长字节转换，运行常规RSA解密脚本时才不会出错。

大数分解网址：

http://www.factordb.com/index.php

最终脚本：(这里积累第四个经验)

```
import libnum

from Crypto.Util.number import long_to_bytes

q = 282164587459512124844245113950593348271

p = 366669102002966856876605669837014229419

e = 65537
```

c = 0xad939ff59f6e70bcbfad406f2494993757eee98b91bc244184a377520d06fc35

n = 10346103590081691412139010129904904441395040517371217043416168653987816098454 9

d = libnum.invmod(e, (p - 1) * (q - 1)) #invmod(a, n) - 求a对于n的模逆,这里逆向加密过程中计算ψ(n)=(p-1)(q-1)，对ψ(n)保密,也就是对应根据e*d=1modψ(n),求出d

m = pow(c, d, n) # 这里的m是十进制形式,pow(x, y[, z])--函数是计算 x 的 y 次方，如果 z 在存在，则再对结果进行取模，其结果等效于 pow(x,y) %z，对应前面解密算法中M=D(C)=（C^d）mod n

string = long_to_bytes(m) # 获取m明文字符串。

print(string)

结果：

攻防世界ReverseMe-120：（base64加密/解密算法、可变参数混淆、寄存器传参、函数名称暗示、冗余中锁定关键代码、函数积累、数组首地址变化遍历字符串算法积累）

32位无壳，还是exe文件，按照流程，先简单运行一下看一下主要显示的字符串：

然后照例扔入IDA32中查看伪代码，有Main函数看main函数：

如下所示，第一个框判断出输入位置并改名input_flag，然后经过第二个框的加密函数处理。继续往下走是其它的操作，其中第三个框的函数双击跟踪不了，最后是明文比较，所以要逆向逻辑。

```
{
  if ( v15 >= 16 )
  {
    v5 = _mm_load_si128((const __m128i *)&byte_414F20);
    v6 = v15 - (v15 & 0xF);
    v7 = (const __m128i *)&v13;
    do
    {
      v8 = _mm_loadu_si128(v7);
      v4 += 16;
      ++v7;
      v7[-1] = _mm_xor_si128(v8, v5);
    }
    while ( v4 < v6 );
  }
  for ( ; v4 < v3; ++v4 )
    *(&v13 + v4) ^= 0x25u;
}
v9 = strcmp(&v13, "you_know_how_to_remove_junk_code");
if ( v9 )
  v9 = v9 < 0 ? -1 : 1;
if ( v9 )
  printf("wrong\n");
else
  printf("correct\n");
system("pause");
return 0;
}
00000655 _main:41 (401255)
```

上面的输入经过一遍操作之后，最后是明文比较

上面第二个框的函数((void (__cdecl *)(char *, unsigned int))sub_401000)(&input_flag, strlen(&input_flag))摆明是对输入字符串进行操作，结果双击跟踪后参数列表却有四个，后来才发现这种参数前后不对等是IDA对寄存器传参的识别，最左边的@eax不明白，也不是寄存器返回值。

```
int __usercall sub_401000@<eax>(unsigned int *a1@<edx>, _BYTE *a2@<ecx>, unsigned __int8 *input_flag, unsigned int _strlen)
{
  int v4; // ebx
  unsigned int v5; // eax
  int v6; // ecx
  unsigned __int8 *v7; // edi
  int v8; // edx
  bool v9; // zf
  unsigned __int8 v10; // cl
  char v11; // cl
  _BYTE *v12; // esi
  unsigned int v13; // ecx
  int v14; // ebx
  unsigned __int8 v15; // cl
  char v16; // dl
  int v20; // [esp+14h] [ebp-4h]
  unsigned int v21; // [esp+14h] [ebp-4h]
  int _strlena; // [esp+24h] [ebp+Ch]

  v4 = 0;
  v5 = 0;
  v6 = 0;
  v20 = 0;
  if ( !_strlen )
    return 0;
  v7 = input_flag;
  do
  {
```

ecx后来发现是寄存器传参，传入的是最后要明文比较的v13。要通过汇编语言查看。

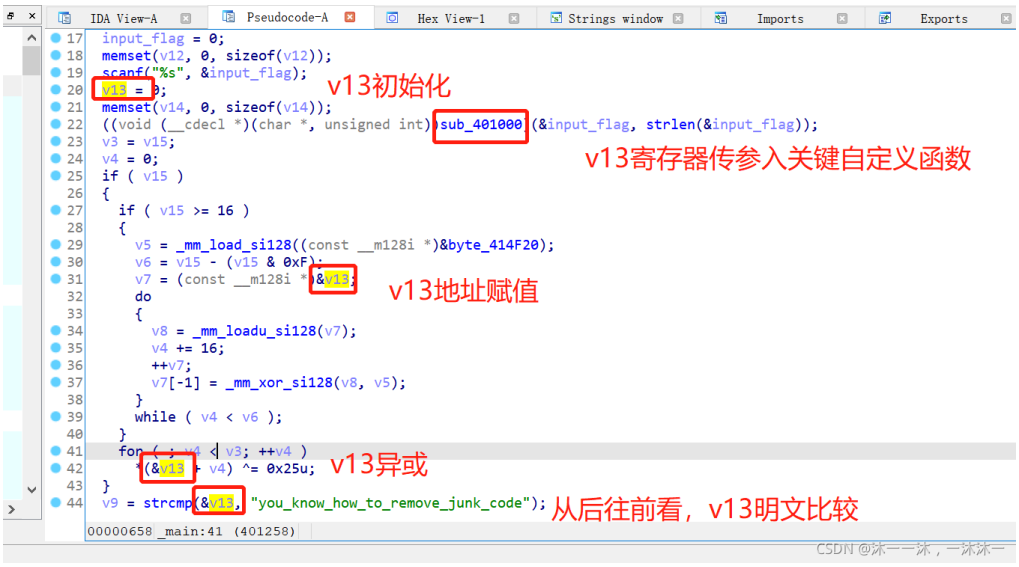
显式传入的两个参数

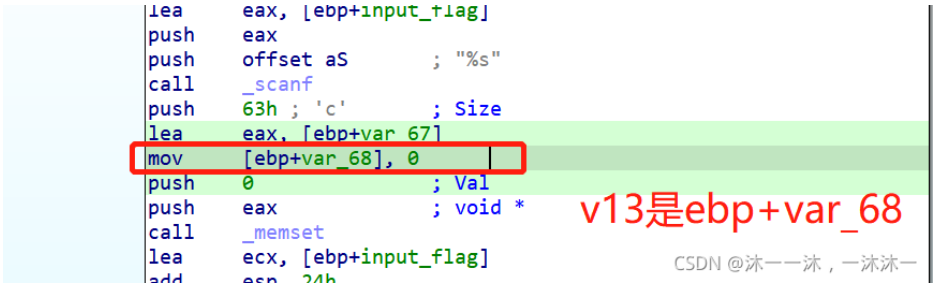
sub_401000函数里代码是很多的，而且最后比较中暗示you_know_how_to_remove_junk_code有垃圾代码，可是垃圾代码一开始真的不知道怎么区分，就算OD动态垃圾代码也会执行。突然回想起前面做题中从后面看起的方法，最后比较的内容一个个找相关，不要考虑地址间接访问修改这种高级技巧先，现在还用不上。

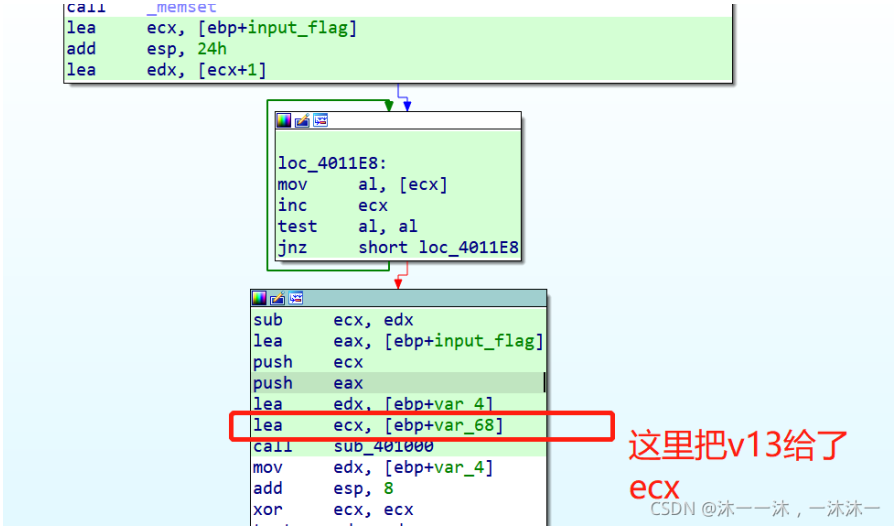
以前的知识回顾：

> 8：
> (这里积累第8个经验)
> 上面是从前往后看的，常规做法是从后往前看，就是确定比较的是s，从s开始排除其他无关变量，确定s由v11赋值而来，找到给v11赋值的代码排除其它干扰代码，从模58和除58中得出是base58加密。

所以从后往前看就是主函数比较的是v13，但v13并没有作为参数传入sub_401000，所以应该是v13以寄存器传参了。查看伪代码也发现的确如此：







而在sub_401000函数内，ecx赋给的局部变量也叫v13。函数中涉及v13的操作集中在第一个框中if语句的下面，所以上面的代码就是垃圾代码，冗余且不用分析。

```
v13 = ((unsigned int)(6 * v6 + 7) >> 3) - v4;
if ( a2 && *a1 >= v13 )
{
  v21 = 3;
  v14 = 0;
  for ( _strlena = 0; v5; --v5 )
  {
    v15 = *v7;
    if ( *v7 != 13 && v15 != 10 && v15 != 32 )
    {
      v16 = byte_414E40[v15];
      v21 -= v16 == 64;
      v14 = v16 & 0x3F | (v14 << 6);
      if ( ++_strlena == 4 )
      {
        _strlena = 0;
        if ( v21 )
          *v12++ = BYTE2(v14);
        if ( v21 > 1 )
          *v12++ = BYTE1(v14);
        if ( v21 > 2 )
          *v12++ = v14;
      }
    }
    ++v7;
  }
```

函数中对v13的操作从这里开始，移位操作看起来像base64

这里v21有三项操作，且判断语句是==4，所以是4变3的base64解密操作。另一个判断一句是这里没有赋值base64加密的等于号


```
    {
      v16 = byte_414E40[v15];
      v21 -= v16 == 64;
      v14 = v16 & 0x3F | (v14 << 6);
      if ( ++_strlena == 4 )
      {
        _strlena = 0;
        if ( v21 )
          *v12++ = BYTE2(v14);
        if ( v21 > 1 )
          *v12++ = BYTE1(v14);
        if ( v21 > 2 )
          *v12++ = v14;
      }
    }
    ++v7;
  }
  *a1 = v12 - a2;
  return 0;
}
*a1 = v13;
return -42;
}
```

最后返回的是v13

所以寄存器传参后也会返回

sub_401000函数内对v13的主要操作代码说是base64解密操作，在一篇博客中可以有很好的理解和对比：

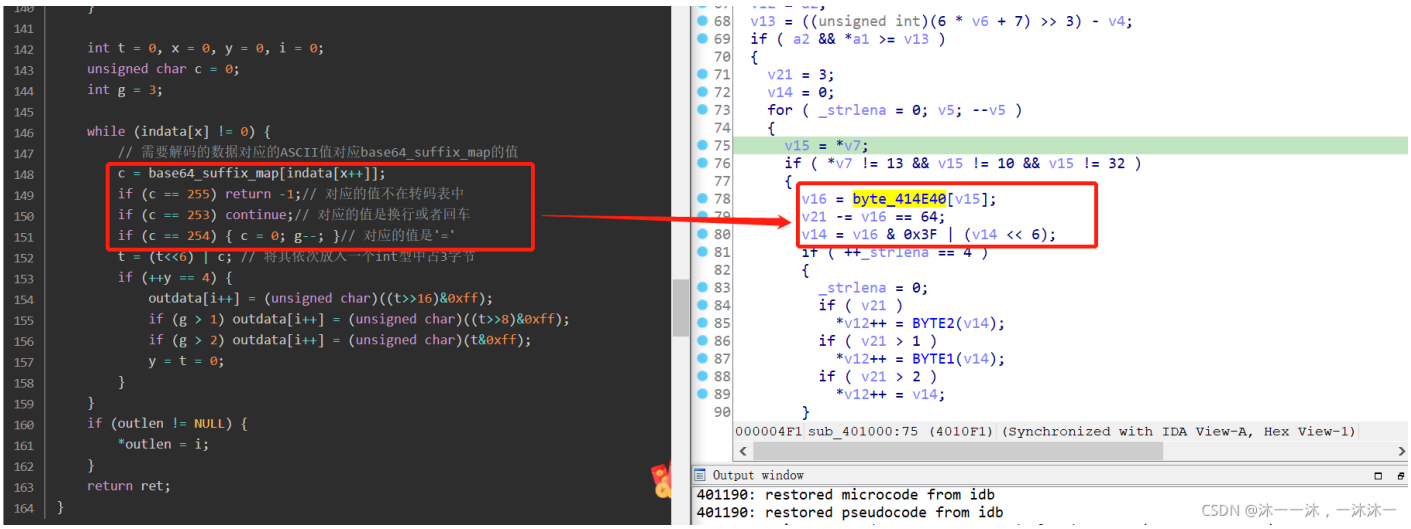

```
    return ret = -2;
}

int t = 0, x = 0, y = 0, i = 0;
unsigned char c = 0;
int g = 3;

while (indata[x] != 0) {
    // 需要解码的数据对应的ASCII值对应base64_suffix_map的值
    c = base64_suffix_map[indata[x++]];
    if (c == 255) return -1;// 对应的值不在转码表中
    if (c == 253) continue;// 对应的值是换行或者回车
    if (c == 254) { c = 0; g--; }// 对应的值是'='
    t = (t<<6) | c; // 将其依次放入一个int型中占3字节
    if (++y == 4) {
        outdata[i++] = (unsigned char)((t>>16)&0xff);
        if (g > 1) outdata[i++] = (unsigned char)((t>>8)&0xff);
        if (g > 2) outdata[i++] = (unsigned char)(t&0xff);
        y = t = 0;
    }
}
if (outlen != NULL) {
    *outlen = i;
}
return ret;
}
```

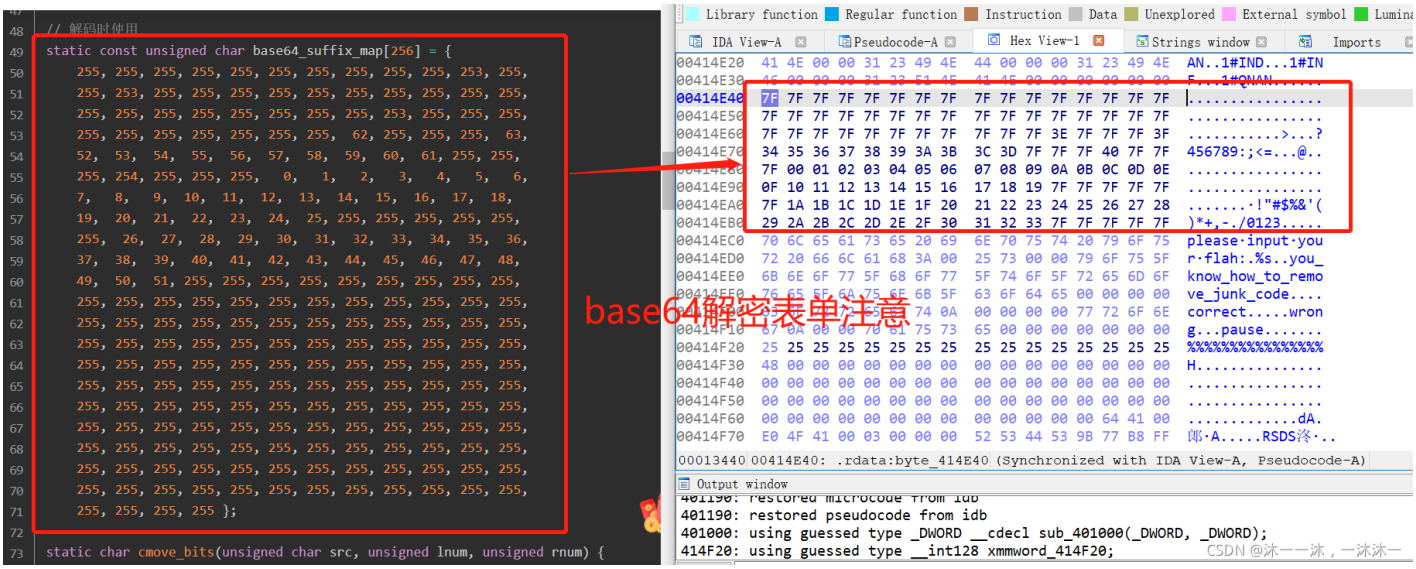
```
IDA Vi… | Pseudoco… | Hex Vi… | Strings wi… | Imp… | Exp…
67  v12 = a2;
68  v13 = ((unsigned int)(6 * v6 + 7) >> 3) - v4;
69  if ( a2 && *a1 >= v13 )
70  {
71    v21 = 3;
72    v14 = 0;
73    for ( _strlena = 0; v5; --v5 )
74    {
75      v15 = *v7;
76      if ( *v7 != 13 && v15 != 10 && v15 != 32 )
77      {
78        v16 = byte_414E40[v15];
79        v21 -= v16 == 64;
80        v14 = v16 & 0x3F | (v14 << 6);
81        if ( ++_strlena == 4 )
82        {
83          _strlena = 0;
84          if ( v21 )
85            *v12++ = BYTE2(v14);
86          if ( v21 > 1 )
87            *v12++ = BYTE1(v14);
88          if ( v21 > 2 )
89            *v12++ = v14;
90        }
91      }
92      ++v7;
93    }
00000532 sub_401000:83 (401132) (Synchronized with IDA View-A)
Output window
401190: restored microcode from idb
401190: restored pseudocode from idb
401000: using guessed type _DWORD __cdecl sub_401000(_DWORD, _DWORD);
```

他们涉及的解密数组base64_suffix_map和byte_414E40有着一样的内容：

附上别人的话：(这里积累第三个经验)

base64的特征：base64正向加密，每三个字节处理变成四个字节，生成一个长字节，再从这个长字节中查四次表生成对应的四个字符。反过来就是先将4个字符进行查表转化成四个字节，然后再四变三。具体的细节就不讨论了，特征应该是如此。对于base系列加密解密，查表与字节变换是核心，非常简单，以后应该留心不能再识别不出来了。





所以sub_401000函数就是base64解码了。

最后就是比较前的一点操作了，这里最外层的v15我也跟踪不出来哪里赋值给了它。看了很多其它资料都说是直接执行第二个蓝框的for循环异或，但是其实是没有进入for循环的，而是在第一个蓝框中执行，但是第一个蓝框和第二个蓝框执行的操作是一样的，说明for循环是出题者给的暗示：

(这当然也是从别人博客中提炼出来的，判断没有进入for循环的办法就是输入正确的flag后OD动态跟踪)

```
memset(v14, 0, sizeof(v14));
((void (__cdecl *)(char *, unsigned int))sub_401000)(&input_flag, strlen(&input_flag));
v3 = v15;
v4 = 0;
if ( v15 )
{
  if ( v15 >= 16 )
  {
    v5 = _mm_load_si128((const __m128i *)&byte_414F20);
    v6 = v15 - (v15 & 0xF);
    v7 = (const __m128i *)&v13;
    do
    {
      v8 = _mm_loadu_si128(v7);
      v4 += 16;
      ++v7;
      v7[-1] = _mm_xor_si128(v8, v5);
    }
    while ( v4 < v6 );
  }
  for ( ; v4 < v3; ++v4 )
    *(&v13 + v4) ^= 0x25u;
}
v9 = strcmp(&v13, "you_know_how_to_remove_junk_code");
if ( v9 )
  v9 = v9 < 0 ? -1 : 1;
```

输入正确flag后进入第一个蓝框，并没有进入第二个蓝框。但是第一个蓝框和第二个蓝框的加密操作是一样的。所以第二个蓝框是出题者给的提示。

出题者给的提示，并没有进入。

这里xmmword有16字节个0x25，符合__m128i _mm_load_si128函数：

```
.rdata:00414F20 xmmword_414F20  xmmword 25252525252525252525252525252525h
```

这里v7=&v13，++v7把地址往前加了一个，配合后面赋值给v7[-1]就是变形的对v7数组赋值。

```
if ( v15 )
{
  if ( v15 >= 0x10 )
  {
    v5 = _mm_load_si128((const __m128i *)&xmmword_414F20);// xmmword_414F20是16字节的（0x25）数据
    v6 = v15 - (v15 & 0xF);
    v7 = (const __m128i *)&v13;                 // v7指向v13
    do
    {
      v8 = _mm_loadu_si128(v7);                 // 这条命令和上面的_mm_load_si128是Intel SSE指令集中load系列，用于加载数据，从内存到暂存器
      v4 += 16;
      ++v7;                                     // v7++，这也是下面将异或后的结果放在v7[-1]的原因
      _mm_storeu_si128((__m128i *)&v7[-1], _mm_xor_si128(v8, v5));// store系列，用于将计算结果等SSE暂存器的数据保存到内存中；_mm_xor_si128计算128位的按位异或；
                                                // 也就是将v8和v5按位异或，然后存在v7[-1]中
    }
    while ( v4 < v6 );
  }
```

这里v4+=16就是为了满足16字节对齐，刚好flag也是32位。

xmmword用于具有MMX和SSE (XMM)指令的128位多媒体操作数（也不知道翻译的对不对，官方解释是“Used for 128-bit multimedia operands with MMX and SSE (XMM) instructions.”）。

SEE指令，参考(https://www.jianshu.com/p/d718c1ea5f22)

load(set)系列，用于加载数据，从内存到暂存器。

__m128i _mm_load_si128(__m128i *p);

__m128i _mm_loadu_si128(__m128i *p);

store系列，用于将计算结果等SSE暂存器的数据保存到内存中。

void _mm_store_si128 (__m128i *p, __m128i a);

void _mm_storeu_si128 (__m128i *p, __m128i a);

_mm_load_si128函数表示从内存中加载一个128bits值到暂存器，也就是16字节，**注意：**p必须是一个16字节对齐的一个变量的地址。返回可以存放在代表寄存器的变量中的值。

_mm_loadu_si128函数和_mm_load_si128一样的，但是不要求地址p是16字节对齐。

store系列的_mm_store_si128和_mm_storeu_si128函数，与上面的load系列的函数是对应的。 表示将__m128i变量a的值存储到p所指定的地址中去。

_mm_xor_si128用于计算128位（16字节）的按位异或，然后通过v14控制循环结束的条件，可以看到v14增长的步长为16，是为了满足前面函数的16字节对齐操作。而且通过上面得到的flag值解码得到的字符串为32个字节大小，正好是16的整数倍。

总算把全部梳理完了，所以流程就是:用户输入---->sub_401000函数base64解码---->每个字符异或处理---->明文比较。

解题脚本：

```
import base64

key1="you_know_how_to_remove_junk_code"

key2=""

for i in range(len(key1)):

key2+=chr(ord(key1[i])^0x25)

print(base64.b64encode(bytes(key2,encoding='UTF-8')))
```

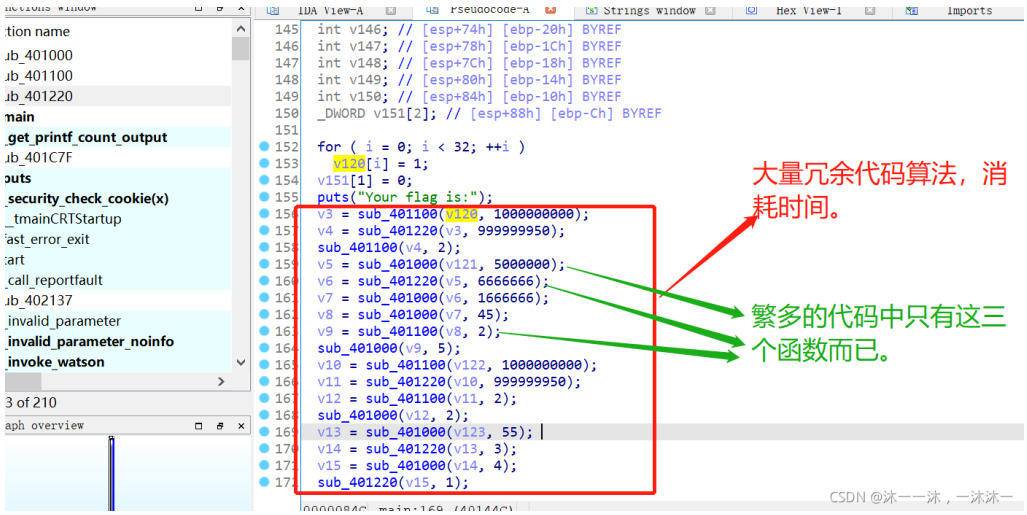
结果：

```
└─$ python 1.py
b'XEpQek5LSlJ6TUpSelFKeldASEpTQHpPUEtOekZKQUA='
```

main函数中嵌入大量冗余代码，拆分代码混淆：

攻防世界Newbie_calculations：（非预期行为、不能直接运行、题目描述暗示、栈地址连续小数组、c语言写脚本、不同系统的特殊数、负数作循环条件）

32位无壳，照例扔入IDA32中查看信息：



浮上眼前的是一堆自定义函数，而且数量很多，吓傻了，快速浏览并随便点进函数看来一下，函数代码还多，以为是混淆，但是又想不出是什么混淆。

运行程序看一下：

输入也输入不了，还以为是程序的什么限制，更慌了，后来查了资料才决定定下心来好好分析。

首先回顾一下以前积累的经验：

复杂代码本质应该是简洁的，这样才叫出题。

仔细一看，发现繁多的代码结果只有三个函数，sub_401100函数、sub_401000函数、sub_401220函数。

加上运行程序时输入不了不是因为程序有问题，每一个意料之外的事情都有它存在的道理和过程，不要总是怀疑题目本身。繁多的代码和巨大的数字大概率是有很多没用的冗余代码占用了程序运行的时间，才导致没有光标可以输入。

最后一行代码应该就是前面运行完后输出的flag：

该伪代码中没有输入，时间到了就会输出flag，但是要修改前面的无用代码。题目是Newbie_calculations，这种题目暗示要注意，表示往运算方面去想函数。

```
294  v116 = v143;
295  v111 = sub_401220(&v149, 1);
296  v112 = sub_401100(v111, 1000000);
297  sub_401000(v112, v116);
298  v117 = v147;
299  v113 = sub_401000(&v150, 1);
300  sub_401100(v113, v117);
301  sub_401000(v151, v150);
302  sub_401C7F("CTF{", 1);
303  for ( j = 0; j < 32; ++j )
304    sub_401C7F("%c", SLOBYTE(v120[j]));
305  sub_401C7F("}\n");
306  return 0;
307}
```

没有用户输入的地方，程序到了时间就会输出flag，但是前面有大量消耗时间的冗余算法。

CSDN @沐一一沐，一沐沐一

现在开始重新分析，从最开头的显示字符串开始：

```
52  for ( i = 0; i < 32; ++i )
53    v120[i] = 1;
54  v151[1] = 0;
55  puts("Your flag is:");
56  v3 = sub_401100(v120, 1000000000);
57  v4 = sub_401220(v3, 999999950);
58  sub_401100(v4, 2);
```

32位数组的v120临时赋值为1，后面会覆盖成flag的一部分

分析第一个sub_401100函数：（这里传入参数被我改名了，简单的传入参数相乘）

_DWORD *__cdecl sub_401100(_DWORD *v120, int _1000000000) //返回v120，故只看对v120的操作

{

int v3; // [esp+Ch] [ebp-1Ch]

int v4; // [esp+14h] [ebp-14h]

int v5; // [esp+18h] [ebp-10h]

int v6; // [esp+18h] [ebp-10h]

int v7; // [esp+1Ch] [ebp-Ch]

int v8; // [esp+20h] [ebp-8h] BYREF

v4 = *v120;

v5 = _1000000000;

v3 = -1;

v8 = 0;

v7 = _1000000000 * v4;

while (_1000000000) //这里积累第一个经验：虽然这里循环1000000000次，但是程序返回的是v120，这里有很多和v120没有关系的其它变量，是用来混淆的，找与v120有关的才是关键。

{

v6 = v7 * v4;

sub_401000(&v8, *v120); //由下面代码知道这是一个相加函数，初始值v8=0，循环1000000000次就是1000000000个v120相加，就是v120 * 1000000000 ，就是传入参数a1*a2结果赋值给第一个参数a1。

++v7;

--_1000000000; //其它的与v120无关的不用看它

v5 = v6 - 1;

}

while (v3) //这里v3=-1，负数循环，这里本来要循环FFFFFFFF，就是100000000 - 1次的，但是后面有*v120=v8的赋值操作，所以这部分也是冗余混淆代码。

{

++v7;

++*v120;

--v3;

--v5;

}

++*v120;

*v120 = v8; //这里最后是赋值v8给v120，所以while(v3)循环根本不用管，前面说过这些和v120没有关系的变量是用来混淆的，不用管。

return v120;

}

分析第二个函数 sub_401000，这也是上面的嵌套函数：（简单的传入参数相加）

_DWORD *__cdecl sub_401000(_DWORD *a1, int a2) //返回a1，故只看对a1的操作

{

int v3; // [esp+Ch] [ebp-18h]

int v4; // [esp+10h] [ebp-14h]

int v5; // [esp+18h] [ebp-Ch]

int v6; // [esp+1Ch] [ebp-8h]

v4 = -1;

v3 = -1 - a2 + 1; //v3=-a2

v6 = 1231;

v5 = a2 + 1231;

while (v3) //这里积累第二个经验：负数做循环条件的知识，v3=-a2，然后在循环体里又--v3，一开始我也以为是死循环，因为0才是false。但是查了资料后说在32位里 -1 就是 FFFFFFFF，就是100000000 - 1。所以这一下子就转正了！所以如果while(-1)就循环100000000 - 1次，这里while(-a2)，所以就循环100000000 - a2次。

{

++v6;

--*a1; //同样的返回a1我们只关注a1即可，这个循环100000000 - a2次，每次a1-1，所以a1变成a1=a1-(100000000 - a2)

--v3;

--v5;

}

while (v4) //这里while(-1)循环100000000 - 1次

{

--v5;

++*a1; //这里加上上面的循环变成a1=a1-(100000000 - a2) + (100000000 - 1) = a1+a2-1

--v4;

}

++*a1; //这里+1，最后结果就变成a1=a1+a2-1+1=a1+a2

return a1; //所以这个函数的作用就是a1=a1+a2，就是把传入的两个参数相加，结果赋值给第一个参数

}

分析最后一个函数sub_401220函数：（简单的传入参数相加）

```
_DWORD *__cdecl sub_401220(_DWORD *a1, int a2) //返回a1，故只看对a1的操作
{
int v3; // [esp+8h] [ebp-10h]
int v4; // [esp+Ch] [ebp-Ch]
int v5; // [esp+14h] [ebp-4h]
int v6; // [esp+14h] [ebp-4h]
v4 = -1;
v3 = -1 - a2 + 1;
v5 = -1;
while ( v3 ) //前面说过这里负数循环100000000 - a2次
{
++*a1; //所以这里a1=a1+100000000 - a2
--v3;
--v5;
}
v6 = v5 * v5;
while ( v4 ) //这里负数循环100000000 - 1次
{
v6 *= 123;
++*a1; //这里a1=a1+100000000 - a2+100000000 - 1
--v4;
}
++*a1; //这里a1=a1+100000000 - a2+100000000 - 1+1，这里积累第三个经验，在32位系统中100000000就是0了，所以上面要写成a1=a1-a2，所以在运算题型中，程序的系统位数也是关键内容
return a1; //所以这个函数就是简单的参数相减操作a1-a2，结果赋值给第一个参数
}
```

那么到这里已经理清程序了，三个函数都可以提取成简单的相乘、相减、相加操作，然后就修改程序了：

第一种手动计算写python脚本：

这里积累第4个经验，IDA反汇编代码中可能把一个连续的数组拆成好多个变量，这些变量在函数栈中是连续的。但是后面整理数组时你很难发现和很难梳理他们是不是同一个数组的内容。此时应该在IDA函数栈中修改变量数组大小为它真正的数组大小。

举个例子，下面明明是打印v120[32]数组的：

```
287  v117 = v136;
288  v113 = sub_401000(&v139, 1);
289  sub_401100(v113, v117);
290  sub_401000(v140, v139);
291  sub_401C7F("CTF{");
292  for ( j = 0; j < 32; ++j )
293    sub_401C7F("%c", SLOBYTE(v120[j]));
294  sub_401C7F("}\n");
295  return 0;
296}
```

最后只对v120进行读取操作

可是IDA变量却只给了V120[12]数组和一堆其它变量，就是它把数组拆分了：

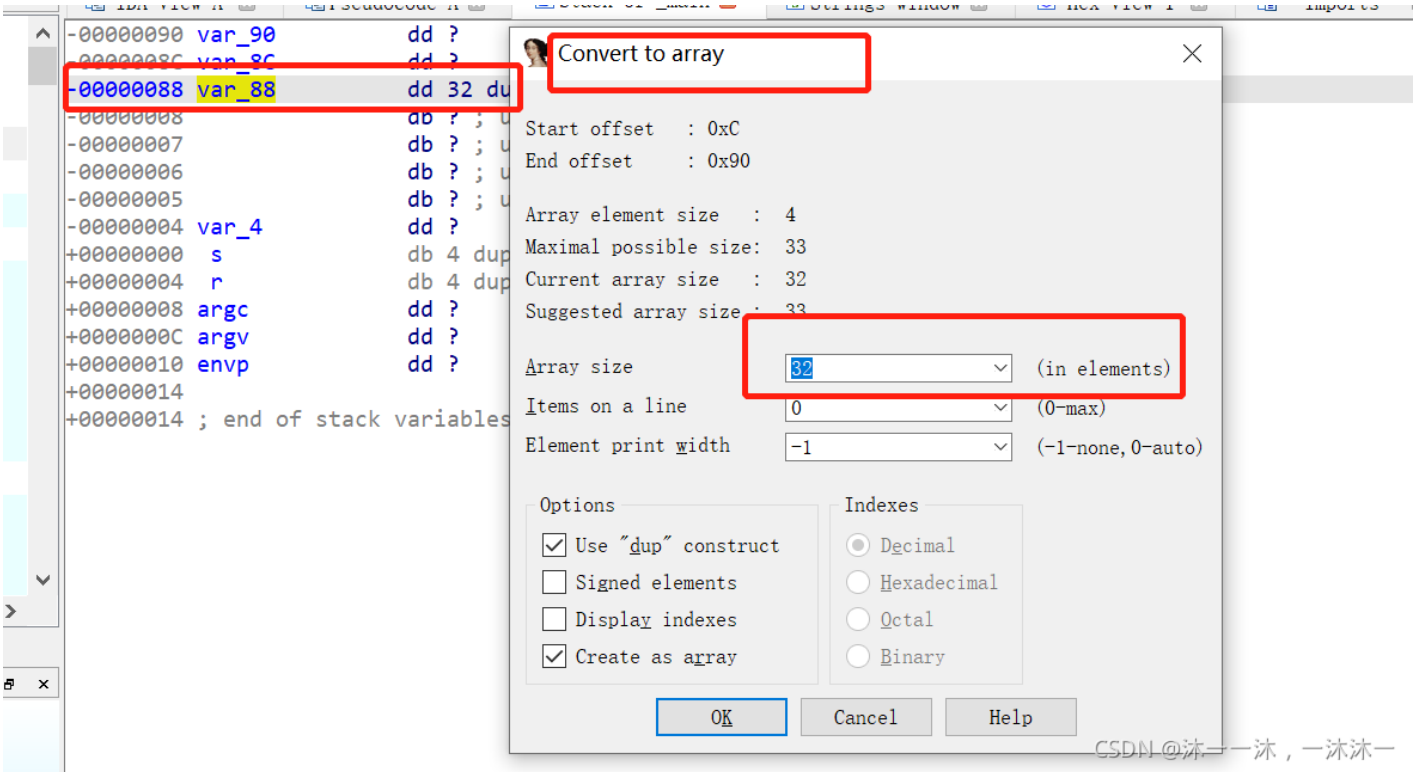
```
116  int v117; // [esp-4h] [ebp-98h]
117  int i; // [esp+4h] [ebp-90h]
118  int j; // [esp+8h] [ebp-8Ch]
119  int v120[12]; // [esp+Ch] [ebp-88h] BYREF
120  int v121; // [esp+3Ch] [ebp-58h] BYREF
121  int v122; // [esp+40h] [ebp-54h] BYREF
122  int v123; // [esp+44h] [ebp-50h] BYREF
123  int v124; // [esp+48h] [ebp-4Ch] BYREF
124  int v125; // [esp+4Ch] [ebp-48h] BYREF
125  int v126; // [esp+50h] [ebp-44h] BYREF
126  int v127; // [esp+54h] [ebp-40h] BYREF
127  int v128; // [esp+58h] [ebp-3Ch] BYREF
128  int v129; // [esp+5Ch] [ebp-38h] BYREF
129  int v130; // [esp+60h] [ebp-34h] BYREF
130  int v131; // [esp+64h] [ebp-30h] BYREF
131  int v132; // [esp+68h] [ebp-2Ch] BYREF
132  int v133; // [esp+6Ch] [ebp-28h] BYREF
133  int v134; // [esp+70h] [ebp-24h] BYREF
134  int v135; // [esp+74h] [ebp-20h] BYREF
135  int v136; // [esp+78h] [ebp-1Ch] BYREF
136  int v137; // [esp+7Ch] [ebp-18h] BYREF
137  int v138; // [esp+80h] [ebp-14h] BYREF
138  int v139; // [esp+84h] [ebp-10h] BYREF
139  _DWORD v140[2]; // [esp+88h] [ebp-Ch] BYREF
140
141  for ( i = 0; i < 32; ++i )
```

可是前面v120却只有12位的数组，后面还对v3等其它数组或变量进行操作。这些都是从v120数组地址中拆分出来的。

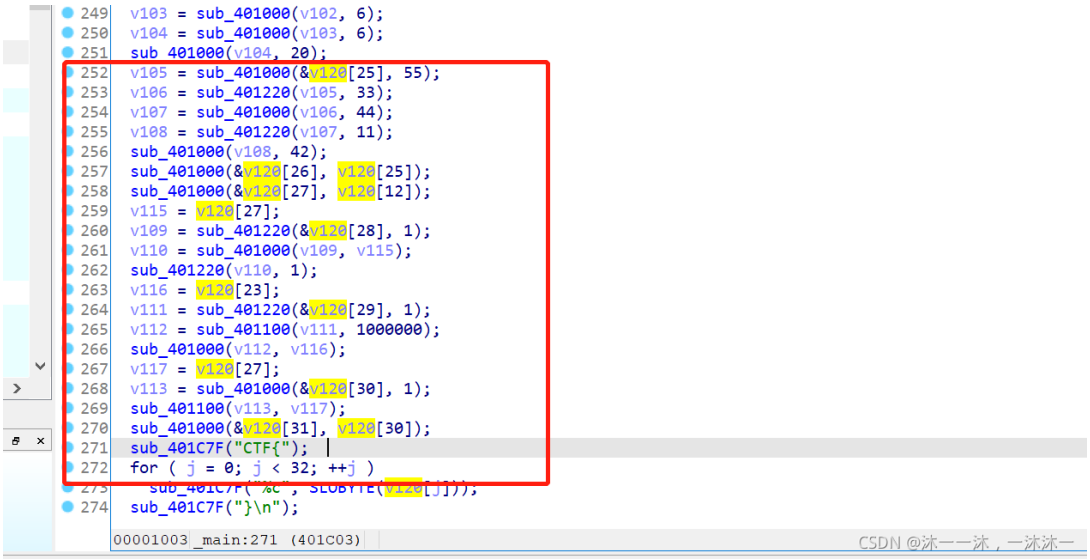
导致的后果就是后面的代码因为用的是连续变量代替数组下标，所以很难理解哪个变量对应哪个下标：(变量的间隔还不同！)

```
218  sub_401000(v60, 20);
219  v61 = sub_401100(&v125, 10);
220  v62 = (_DWORD *)sub_401220(v61, 5);
221  v63 = sub_401100(v62, 8);
222  v64 = sub_401000(v63, 9);
223  sub_401000(v64, 48);
224  v65 = sub_401000(&v126, 7);
225  v66 = sub_401000(v65, 6);
226  v67 = sub_401000(v66, 5);
227  v68 = sub_401000(v67, 4);
228  v69 = sub_401000(v68, 3);
229  v70 = sub_401000(v69, 2);
230  v71 = sub_401000(v70, 1);
231  sub_401000(v71, 20);
232  v72 = sub_401000(&v127, 7);
233  v73 = sub_401000(v72, 2);
234  v74 = sub_401000(v73, 4);
235  v75 = sub_401000(v74, 3);
236  v76 = sub_401000(v75, 6);
237  v77 = sub_401000(v76, 5);
238  v78 = sub_401000(v77, 1);
239  sub_401000(v78, 20);
240  v79 = sub_401100(&v128, 1000000);
241  v80 = (_DWORD *)sub_401220(v79, 999999);
242  v81 = sub_401100(v80, 4);
243  v82 = sub_401000(v81, 50);
244  sub_401220(v82, 1);
```


所以我们要在IDA栈中修改v120[12]为v120[32]：



这样修改之后就好看多了，不过手动计算好像还是很麻烦，算了，不手动计算了。(笑~)



第二种方法：直接复制到C语言中修改代码

直接复制到C语言中修改代码吧，很简单的，首先修改_Dword为int，然后把三个函数都改个函数名，打印函数换成Printf就好啦！

(注意！这里如果没有像前面那样修改栈v120[32]数字的话，多个拆分变量在dev中就会造成变量之间的空间不连续，不连续就没法作为一个连续数组输出了，就会输出个四不像出来。)：

```
#include<iostream>

using namespace std;

int *first(int *a1,int a2) //函数题要在Main函数外声明，返回类型是指针，所以int *做返回类型。

{

*a1=*a1*a2;
```

```
return a1;
}
int *second(int *a1,int a2)
{
*a1=*a1-a2;
return a1;
}
int *third(int *a1,int a2)
{
*a1=*a1+a2;
return a1;
}
int main(int argc, const char **argv, const char **envp)
{
int *v3; // eax
int *v4; // eax
int *v5; // eax
int *v6; // eax
int *v7; // eax
int *v8; // eax
int *v9; // eax
int *v10; // eax
int *v11; // eax
int *v12; // eax
int *v13; // eax
int *v14; // eax
int *v15; // eax
int *v16; // eax
int *v17; // eax
int *v18; // eax
int *v19; // eax
```

```
int *v20; // eax
int *v21; // eax
int *v22; // eax
int *v23; // eax
int *v24; // eax
int *v25; // eax
int *v26; // eax
int *v27; // eax
int *v28; // eax
int *v29; // eax
int *v30; // eax
int *v31; // eax
int *v32; // eax
int *v33; // eax
int *v34; // eax
int *v35; // eax
int *v36; // eax
int *v37; // eax
int *v38; // eax
int *v39; // eax
int *v40; // eax
int *v41; // eax
int *v42; // eax
int *v43; // eax
int *v44; // eax
int *v45; // eax
int *v46; // eax
int *v47; // eax
int *v48; // eax
int *v49; // eax
int *v50; // eax
```

```
int *v51; // eax
int *v52; // eax
int *v53; // eax
int *v54; // eax
int *v55; // eax
int *v56; // eax
int *v57; // eax
int *v58; // eax
int *v59; // eax
int *v60; // eax
int *v61; // eax
int *v62; // eax
int *v63; // eax
int *v64; // eax
int *v65; // eax
int *v66; // eax
int *v67; // eax
int *v68; // eax
int *v69; // eax
int *v70; // eax
int *v71; // eax
int *v72; // eax
int *v73; // eax
int *v74; // eax
int *v75; // eax
int *v76; // eax
int *v77; // eax
int *v78; // eax
int *v79; // eax
int *v80; // eax
int *v81; // eax
```

```
int *v82; // eax
int *v83; // eax
int *v84; // eax
int *v85; // eax
int *v86; // eax
int *v87; // eax
int *v88; // eax
int *v89; // eax
int *v90; // eax
int *v91; // eax
int *v92; // eax
int *v93; // eax
int *v94; // eax
int *v95; // eax
int *v96; // eax
int *v97; // eax
int *v98; // eax
int *v99; // eax
int *v100; // eax
int *v101; // eax
int *v102; // eax
int *v103; // eax
int *v104; // eax
int *v105; // eax
int *v106; // eax
int *v107; // eax
int *v108; // eax
int *v109; // eax
int *v110; // eax
int *v111; // eax
int *v112; // eax
```

```
int *v113; // eax
int v115; // [esp-8h] [ebp-9Ch]
int v116; // [esp-4h] [ebp-98h]
int v117; // [esp-4h] [ebp-98h]
int i; // [esp+4h] [ebp-90h]
int j; // [esp+8h] [ebp-8Ch]
int v120[33]; // [esp+Ch] [ebp-88h] BYREF
for ( i = 0; i < 32; ++i )
v120[i] = 1; // 最后操作的是v120，直接跟踪v120即可，这里赋值v120[32]都为1
v120[32] = 0;
puts("Your flag is:");
v3 = first(v120, 1000000000);
v4 = second(v3, 999999950);
first(v4, 2); // v120=100
v5 = third(&v120[1], 5000000);
v6 = second(v5, 6666666);
v7 = third(v6, 1666666);
v8 = third(v7, 45);
v9 = first(v8, 2);
third(v9, 5); // 97
v10 = first(&v120[2], 1000000000);
v11 = second(v10, 999999950);
v12 = first(v11, 2);
third(v12, 2); // 104
v13 = third(&v120[3], 55);
v14 = second(v13, 3);
v15 = third(v14, 4);
second(v15, 1); // 56
v16 = first(&v120[4], 100000000);
v17 = second(v16, 99999950);
v18 = first(v17, 2);
```

```
third(v18, 2); // 102
v19 = second(&v120[5], 1);
v20 = first(v19, 1000000000);
v21 = third(v20, 55);
second(v21, 3); // 58
v22 = first(&v120[6], 1000000);
v23 = second(v22, 999975);
first(v23, 4); // 100
v24 = third(&v120[7], 55);
v25 = second(v24, 33);
v26 = third(v25, 44);
second(v26, 11); // 56
v27 = first(&v120[8], 10);
v28 = second(v27, 5);
v29 = first(v28, 8);
third(v29, 9); // 49
v30 = third(&v120[9], 0);
v31 = second(v30, 0);
v32 = third(v31, 11);
v33 = second(v32, 11);
third(v33, 53); // 54
v34 = third(&v120[10], 49);
v35 = second(v34, 2);
v36 = third(v35, 4);
second(v36, 2); // 50
v37 = first(&v120[11], 1000000);
v38 = second(v37, 999999);
v39 = first(v38, 4);
third(v39, 50); // 54
v40 = third(&v120[12], 1);
v41 = third(v40, 1);
```

```
v42 = third(v41, 1);
v43 = third(v42, 1);
v44 = third(v43, 1);
v45 = third(v44, 1);
v46 = third(v45, 10);
third(v46, 32); // 49
v47 = first(&v120[13], 10);
v48 = second(v47, 5);
v49 = first(v48, 8);
v50 = third(v49, 9);
third(v50, 48); // 97
v51 = second(&v120[14], 1);
v52 = first(v51, -294967296);
v53 = third(v52, 55);
second(v53, 3); // 52
v54 = third(&v120[15], 1);
v55 = third(v54, 2);
v56 = third(v55, 3);
v57 = third(v56, 4);
v58 = third(v57, 5);
v59 = third(v58, 6);
v60 = third(v59, 7);
third(v60, 20); // 48
v61 = first(&v120[16], 10);
v62 = second(v61, 5);
v63 = first(v62, 8);
v64 = third(v63, 9);
third(v64, 48); // 97
v65 = third(&v120[17], 7);
v66 = third(v65, 6);
v67 = third(v66, 5);
```

```
v68 = third(v67, 4);
v69 = third(v68, 3);
v70 = third(v69, 2);
v71 = third(v70, 1);
third(v71, 20); // 49
v72 = third(&v120[18], 7);
v73 = third(v72, 2);
v74 = third(v73, 4);
v75 = third(v74, 3);
v76 = third(v75, 6);
v77 = third(v76, 5);
v78 = third(v77, 1);
third(v78, 20); // 49
v79 = first(&v120[19], 1000000);
v80 = second(v79, 999999);
v81 = first(v80, 4);
v82 = third(v81, 50);
second(v82, 1); // 53
v83 = second(&v120[20], 1);
v84 = first(v83, -294967296);
v85 = third(v84, 49);
second(v85, 1);
v86 = second(&v120[21], 1); // 48
v87 = first(v86, 1000000000);
v88 = third(v87, 54);
v89 = second(v88, 1);
v90 = third(v89, 1000000000);
second(v90, 1000000000); // 53
v91 = third(&v120[22], 49);
v92 = second(v91, 1);
v93 = third(v92, 2);
```

```
second(v93, 1); // 50
v94 = first(&v120[23], 10);
v95 = second(v94, 5);
v96 = first(v95, 8);
v97 = third(v96, 9);
third(v97, 48); // 97
v98 = third(&v120[24], 1);
v99 = third(v98, 3);
v100 = third(v99, 3);
v101 = third(v100, 3);
v102 = third(v101, 6);
v103 = third(v102, 6);
v104 = third(v103, 6);
third(v104, 20); // 49
v105 = third(&v120[25], 55);
v106 = second(v105, 33);
v107 = third(v106, 44);
v108 = second(v107, 11);
third(v108, 42); // 97
third(&v120[26], v120[25]); // 56
third(&v120[27], v120[12]);
v115 = v120[27];
v109 = second(&v120[28], 1);
v110 = third(v109, v115);
second(v110, 1);
v116 = v120[23];
v111 = second(&v120[29], 1);
v112 = first(v111, 1000000);
third(v112, v116);
v117 = v120[27];
v113 = third(&v120[30], 1);
```

first(v113, v117);

third(&v120[31], v120[30]);

printf("CTF{");

for (j = 0; j < 32; ++j)

printf("%c", (v120[j]));

printf("}");

return 0;

}

结果：

```
{
    *a1=*a1*a2;
    return a1;
}
int *second(int *a1,int a2)
{
    *a1=*a1-a2;
    return a1;
}
int *third(int *a1,int a2)
{
    *a1=*a1+a2;
    return a1;
}

int main(int argc, const char **argv, const char *
{
```

```
D:\桌面\Untitled2.exe
Your flag is:
CTF{daf8f4d816261a41a115052a1bc21ade}
--------------------------------
Process exited after 0.2494 seconds with return value 0
请按任意键继续. . .
```

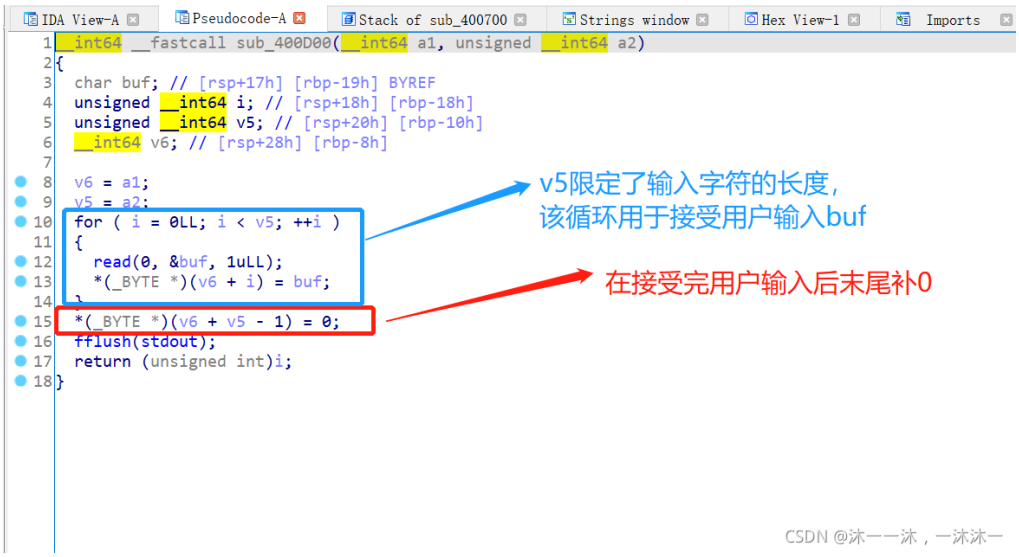

攻防世界testre：（函数逻辑封装、冗余中锁定关键代码、base58加密算法、）

64位ELF文件，无壳，照例扔入IDA64中查看伪代码，有main函数看main函数：

主函数就两个自定义函数，按顺序跟踪第一个 sub_400D00函数。

（这里积累第一个经验）

v6是字符数组，v5是17uLL，限定了v6接受输入字符的长度，就是flag的长度。v6 + v5 - 1就是在接受用户输入的字符后在结尾附0结束符。



继续跟踪下一个sub_400700自定义函数：

__int64 __fastcall sub_400700(void *a1, _QWORD *a2, __int64 a3, size_t a4)

{

unsigned __int8 *v4; // rcx

_DWORD v6[2]; // [rsp+0h] [rbp-C0h] BYREF

int c; // [rsp+8h] [rbp-B8h]

char v8; // [rsp+Fh] [rbp-B1h]

int v9; // [rsp+10h] [rbp-B0h]

bool v10; // [rsp+17h] [rbp-A9h]

unsigned __int8 *v11; // [rsp+18h] [rbp-A8h]

char v12; // [rsp+27h] [rbp-99h]

int v13; // [rsp+28h] [rbp-98h]

int v14; // [rsp+2Ch] [rbp-94h]

unsigned __int64 i; // [rsp+30h] [rbp-90h]

size_t n; // [rsp+38h] [rbp-88h]

size_t v17; // [rsp+40h] [rbp-80h]

size_t v18; // [rsp+48h] [rbp-78h]

size_t j; // [rsp+50h] [rbp-70h]

size_t v20; // [rsp+58h] [rbp-68h]

int v21; // [rsp+64h] [rbp-5Ch]

unsigned __int64 v22; // [rsp+68h] [rbp-58h]

int v23; // [rsp+74h] [rbp-4Ch]

```
_DWORD *v24; // [rsp+78h] [rbp-48h]
__int64 v25; // [rsp+80h] [rbp-40h]
void *v26; // [rsp+88h] [rbp-38h]
int v27; // [rsp+94h] [rbp-2Ch]
size_t v28; // [rsp+98h] [rbp-28h]
__int64 v29; // [rsp+A0h] [rbp-20h]
_QWORD *v30; // [rsp+A8h] [rbp-18h]
void *s; // [rsp+B0h] [rbp-10h]
char v32; // [rsp+BFh] [rbp-1h]
s = a1;
v30 = a2;
v29 = a3;
v28 = a4;
v27 = -559038737;
v26 = malloc(0x100uLL);
v25 = v29;
v24 = v6;
v22 = 0LL;
v17 = 0LL;
for ( i = 0LL; i < v28; ++i )
{
v13 = *(unsigned __int8 *)(v25 + i);
*((_BYTE *)v26 + i) = byte_400E90[i % 0x1D] ^ v13;
*((_BYTE *)v26 + i) += *(_BYTE *)(v25 + i);
}
while ( 1 )
{
v12 = 0;
if ( v17 < v28 )
v12 = ~(*(_BYTE *)(v25 + v17) != 0);
if ( (v12 & 1) == 0 )
```

```
break;
++v17;
}
n = 138 * (v28 - v17) / 0x64 + 1;
v23 = ((v17 + v28) << 6) / 0x30 - 1;
v11 = (unsigned __int8 *)v6 - ((138 * (v28 - v17) / 0x64 + 16) & 0xFFFFFFFFFFFFFFF0LL);
memset(v11, 0, n);
v20 = v17;
v18 = n - 1;
while ( v20 < v28 )
{
v21 = *(unsigned __int8 *)(v25 + v20);
for ( j = n - 1; ; --j )
{
v10 = 1;
if ( j <= v18 )
v10 = v21 != 0;
if ( !v10 )
break;
v22 = v11[j] << 6;
v21 += v11[j] << 8;
v9 = 64;
v11[j] = v21 % 58;
*((_BYTE *)v26 + j) = v22 & 0x3F;
v22 >>= 6;
v21 /= 58;
v27 /= v9;
if ( !j )
break;
}
++v20;
```

```
v18 = j;
}
for ( j = 0LL; ; ++j )
{
v8 = 0;
if ( j < n )
v8 = ~(v11[j] != 0);
if ( (v8 & 1) == 0 )
break;
}
if ( *v30 > n + v17 - j )
{
if ( v17 )
{
c = 61;
memset(s, 49, v17);
memset(v26, c, v17);
}
v20 = v17;
while ( j < n )
{
v4 = v11;
*((_BYTE *)s + v20) = byte_400EB0[v11[j]];
*((_BYTE *)v26 + v20++) = byte_400EF0[v4[j++]];
}
*((_BYTE *)s + v20) = 0;
*v30 = v20 + 1;
if ( !strncmp((const char *)s, "D9", 2uLL)
&& !strncmp((const char *)s + 20, "Mp", 2uLL)
&& !strncmp((const char *)s + 18, "MR", 2uLL)
&& !strncmp((const char *)s + 2, "cS9N", 4uLL)
```

&& !strncmp((const char *)s + 6, "9iHjM", 5uLL)

&& !strncmp((const char *)s + 11, "LTdA8YS", 7uLL))

{

v6[1] = puts("correct!");

}

v32 = 1;

v14 = 1;

}

else

{

*v30 = n + v17 - j + 1;

v32 = 0;

v14 = 1;

}

return v32 & 1;

}

代码量多，算法也比较复杂，我也看到了最后的比较函数，也知道s是关键，也感觉有混淆，但是区分不出哪个是混淆代码。(哭~)

查了很多资料(wp)最后发现还是官方wp最详细，现在整理一下自己的思路：

(这里积累第二个经验)

首先看第一个for循环，是对一个字符数组的异或加密操作，字符串fake_secret_makes_you_annoyed中文暗示可以看出是假的flag。这里用到了v26，v26在后面的代码中也有穿插，但是根据别人资料的描述，这里的异或加密后面会发现,并没有用到。

```
36  v27 = -559038737;
37  v26 = malloc(0x100uLL);
38  v25 = v29;
39  v24 = v6;
40  v22 = 0LL;
41  v17 = 0LL;
42  for ( i = 0LL; i < v28; ++i )
43  {
44    v13 = *(unsigned __int8 *)(v25 + i);
45    *((_BYTE *)v26 + i) = byte_400E90[i % 29] ^ v13;
46    *((_BYTE *)v26 + i) += *(_BYTE *)(v25 + i);
47  }
48  while ( 1 )
49  {
50    v12 = 0;
51    if ( v17 < v28 )
52      v12 = ~(*(_BYTE *)(v25 + v17) != 0);
53    if ( (v12 & 1) == 0 )
54      break;
55    ++v17;
56  }
57  n = 138 * (v28 - v17) / 0x64 + 1;
58  v23 = ((v17 + v28) << 6) / 0x30 - 1;
59  v11 = (unsigned __int8 *)v6 - ((138 * (v28 - v17) / 0x64 + 16) & 0xFFFFFFFFFFFFFFF0LL);
60  memset(v11, 0, n);
61  v20 = v17;
```

对v26的操作是正规的，base64加密算法的，但是最后比较发信啊没有对v26进行比较，也就是说一切与v26有关的都不用管。


```
.rodata:0000000000400E8E                db    0
.rodata:0000000000400E8F                db    0
.rodata:0000000000400E90 ; char aFakeSecretMake[]
.rodata:0000000000400E90 aFakeSecretMake db 'fake_secret_makes_you_annoyed',0
.rodata:0000000000400E90                                        ; DATA XREF: sub_400700+9E↑r
.rodata:0000000000400EAE                align 10h
.rodata:0000000000400EB0 ; char byte_400EB0[]
.rodata:0000000000400EB0 byte_400EB0    db 31h                  ; DATA XREF: sub_400700+446↑r
.rodata:0000000000400EB1 a23456789abcdef db '23456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz',0
.rodata:0000000000400EEB                align 10h
.rodata:0000000000400EF0 ; char byte_400EF0[]
.rodata:0000000000400EF0 byte_400EF0    db 41h                  ; DATA XREF: sub_400700+464↑r
.rodata:0000000000400EF1 aBcdefghijklmno db 'BCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/',0
.rodata:0000000000400F31 ; const char s2[3]
.rodata:0000000000400F31 s2             db 'D9',0               ; DATA XREF: sub_400700:loc_400B95↑o
.rodata:0000000000400F34 ; const char aMp[3]
.rodata:0000000000400F34 aMp            db 'Mp',0               ; DATA XREF: sub_400700+4D3↑o
.rodata:0000000000400F37 ; const char aMr[3]
.rodata:0000000000400F37 aMr            db 'MR',0               ; DATA XREF: sub_400700+4FD↑o
.rodata:0000000000400F3A ; const char aCs9n[]
.rodata:0000000000400F3A aCs9n          db 'cS9N',0             ; DATA XREF: sub_400700+527↑o
```

数组内的内容


```
61  v20 = v17;
62  v18 = n - 1;
63  while ( v20 < v28 )
64  {
65    v21 = *(unsigned __int8 *)(v25 + v20);
66    for ( j = n - 1; ; --j )
67    {
68      v10 = 1;
69      if ( j <= v18 )
70        v10 = v21 != 0;
71      if ( !v10 )
72        break;
73      v22 = v11[j] << 6;
74      v21 += v11[j] << 8;
75      v9 = 64;
76      v11[j] = v21 % 58;
77      *((_BYTE *)v26 + j) = v22 & 63;
78      v22 >>= 6;
79      v21 /= 58;
80      v27 /= v9;
81      if ( !j )
82        break;
83    }
84    ++v20;
85    v18 = j;
00000924 sub_400700:66 (400924)
ease change the segment NAME.
```

这里在base58函数中穿插了v26，但是看来v26也是与64位的base64形式

(这里积累第三个经验)

接着看第二个while循环，这里的v12在后面真的是一点都没用上。不要再想着那些地址间接访问了，现在为止除了EASYHOOK有对地址函数的替换之外没有遇到任何一个地址间接访问。所以这里的函数也是与解题无关的混淆代码。

```
39  v24 = v6;
40  v22 = 0LL;
41  v17 = 0LL;
42  for ( i = 0LL; i < v28; ++i )
43  {
44    v13 = *(unsigned __int8 *)(v25 + i);
45    *((_BYTE *)v26 + i) = aFakeSecretMake[i % 29] ^ v13;
46    *((_BYTE *)v26 + i) += *(_BYTE *)(v25 + i);
47  }
48  while ( 1 )
49  {
50    v12 = 0;
51    if ( v17 < v28 )
52      v12 = ~(*(_BYTE *)(v25 + v17) != 0);
53    if ( (v12 & 1) == 0 )
54      break;
55    ++v17;
56  }
57  n = 138 * (v28 - v17) / 0x64 + 1;
58  v23 = ((v17 + v28) << 6) / 0x30 - 1;
59  v11 = (unsigned __int8 *)v6 - ((138 * (v28 - v17) / 0x64 + 16) & 0xFFFFFFFFFFFFFFF
60  memset(v11, 0, n);
61  v20 = v17;
62  v18 = n - 1;
63  while ( v20 < v28 )
64  {
```

对v12的所有操作在后面的比较中都没有涉及，所以v12也是干扰用的。

(这里积累第四个经验)

接着看第三个函数这里v11亮相，根据资料，这里一个模58，一个除58是base58加密的关键，由此判断这是bae58加密,操作对象是v11。(这是别人的图)

```
64  while ( v20 < v28 )
65  {
66    v21 = *(unsigned __int8 *)(v25 + v20);
67    for ( j = n - 1; ; --j )
68    {
69      v10 = 1;
70      if ( j <= v18 )
71        v10 = v21 != 0;
72      if ( !v10 )
73        break;
74      v22 = v11[j] << 6;
75      v21 += v11[j] << 8;
76      v9 = 64;
77      v11[j] = v21 % 58;
78      *((_BYTE *)v26 + j) = v22 & 0x3F;
79      v22 >>= 6;
80      v21 /= 58;
81      v27 /= v9;
82      if ( !j )
83        break;
84    }
85    ++v20;
86    v18 = j;
87  }
```

猜测是base58加密

接着分析第四个循环，这里是两个简单的判断，操作对象是v8。这个V8在后面也没有用上，所以这也是混淆代码：

```
84    ++v20;
85    v18 = j;
86  }
87  for ( j = 0LL; ; ++j )
88  {
89    v8 = 0;
90    if ( j < n )
91      v8 = ~(v11[j] != 0);
92    if ( (v8 & 1) == 0 )
93      break;
94  }
95  if ( *v30 > n + v17 - j )
96  {
97    if ( v17 )
98    {
99      c = 61;
100     memset(s, 49, v17);
101     memset(v26, c, v17);
102   }
103   v20 = v17;
104   while ( j < n )
105   {
106     v4 = v11;
107     *((_BYTE *)s + v20) = byte_400EB0[v11[j]];
108     *((_BYTE *)v26 + v20++) = byte_400EF0[v4[j++]];
109   }
```

对v8的操作在后面也没有用上，
所以v8也是干扰用的。

(这里积累第六个经验)

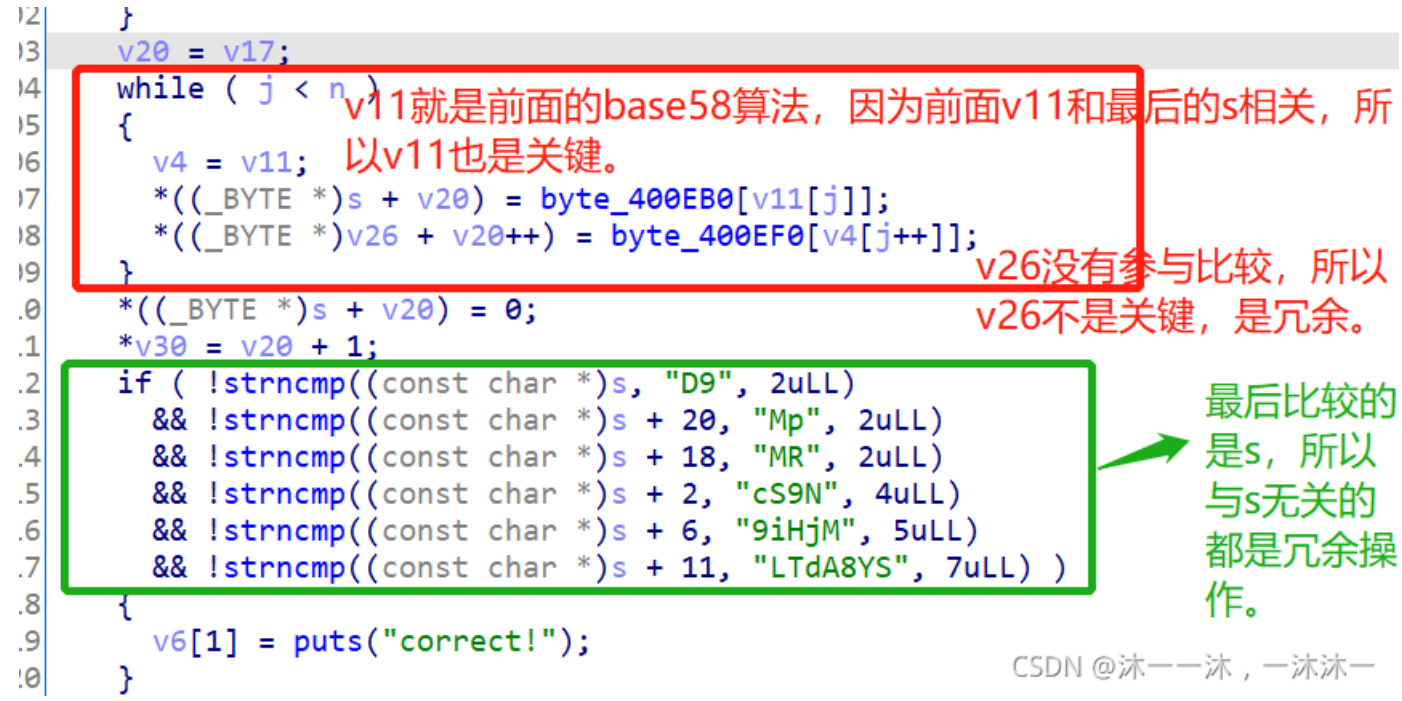
紧接着的if语句只是简单的给s和v26开辟空间，并没有太大作用：

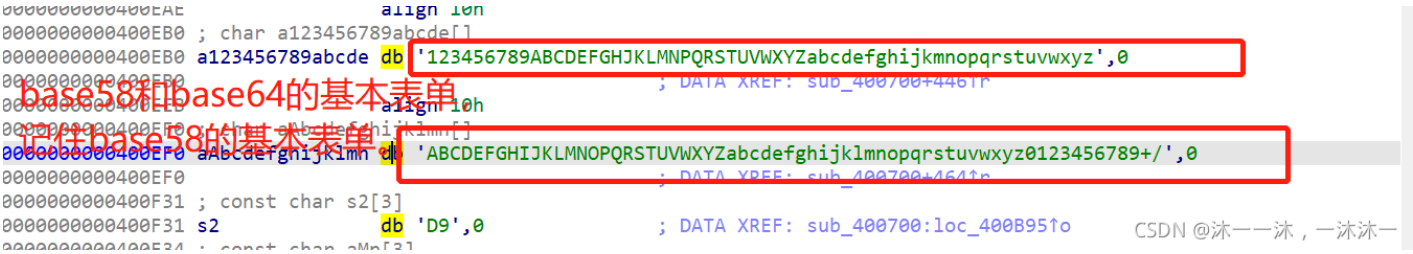
```
91      v8 = ~(v11[j] != 0);
92    if ( (v8 & 1) == 0 )
93      break;
94  }
95  if ( *v30 > n + v17 - j )
96  {
97    if ( v17 )
98    {
99      c = 61;
100     memset(s, 49, v17);
101     memset(v26, c, v17);
102   }
103   v20 = v17;
104   while ( j < n )
105   {
106     v4 = v11;
107     *((_BYTE *)s + v20) = byte_400EB0[v11[j]];
108     *((_BYTE *)v26 + v20++) = byte_400EF0[v4[j++]];
109   }
110   *((_BYTE *)s + v20) = 0;
111   *v30 = v20 + 1;
112   if ( !strncmp((const char *)s, "D9", 2uLL)
113     && !strncmp((const char *)s + 20, "Mp", 2uLL)
114     && !strncmp((const char *)s + 18, "MR", 2uLL)
115     && !strncmp((const char *)s + 2, "cS9N", 4uLL)
116     && !strncmp((const char *)s + 6, "9iHjM", 5uLL)
```

开辟空间用的if语句

(这里积累第7个经验)

最后的while函数就是给s和v26赋值，用到了两个数组，分别是base58和base64的基本元素。根据下面的循环条件可以判断这个v26并没有用于参与判断，也就是说前面关于v26的都是混淆的代码





附上别人的话：

显然一个是base64编码表,一个是base58编码表,最开始把base58编码表错看成了是数字加所有字母,浪费大量时间分析.

仔细观察代码,其实进行base64编码的过程是针对v26,但是v26变量指向的内存完全没有和最后的比较产生关系,所以这都是干扰做题的。

最后观察比较语句,提取出最终串:D9cS9N9iHjMLTdA8YSMRMp

所以最后比较的就是输入后的bae58加密是否相等，那拿D9cS9N9iHjMLTdA8YSMRMp解密即可得到flag：

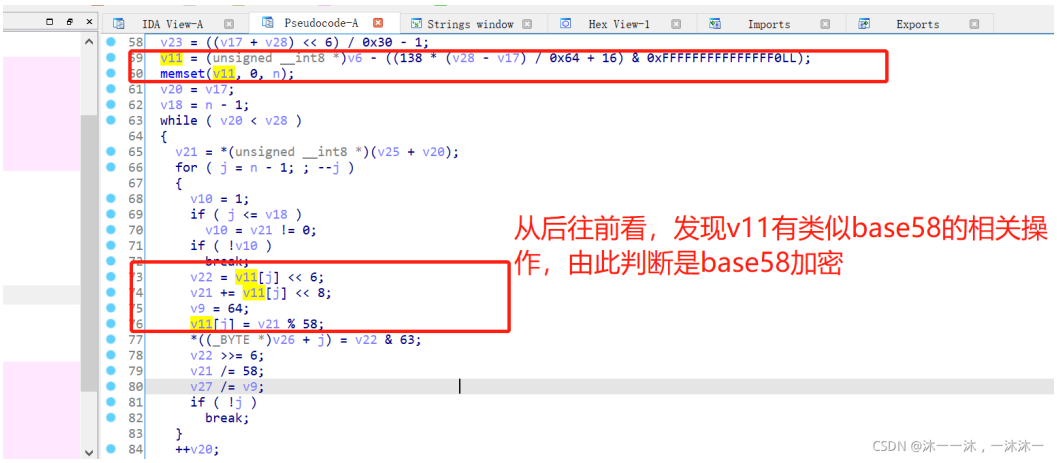
上面是从前往后看的，常规做法是从后往前看，就是确定比较的是s，从s开始排除其他无关变量，确定s由v11赋值而来，找到给v11赋值的代码排除其它干扰代码，从模58和除58中得出是base58加密。

```
101     memset(v26, c, v17);
102   }
103   v20 = v17;
104   while ( j < n )
105   {
106     v4 = v11;
107     *((_BYTE *)s + v20) = byte_400EB0[v11[j]];
108     *((_BYTE *)v26 + v20++) = byte_400EF0[v4[j++]];
109   }
110   *((_BYTE *)s + v20) = 0;
111   *v30 = v20 + 1;
112   if ( !strncmp((const char *)s, "D9", 2uLL)
113     && !strncmp((const char *)s + 20, "Mp", 2uLL)
114     && !strncmp((const char *)s + 18, "MR", 2uLL)
115     && !strncmp((const char *)s + 2, "cS9N", 4uLL)
116     && !strncmp((const char *)s + 6, "9iHjM", 5uLL)
117     && !strncmp((const char *)s + 11, "LTdA8YS", 7uLL) )
118   {
119     v6[1] = puts("correct!");
120   }
121   v32 = 1;
122   v14 = 1;
123 }
124 else
```

从后往前看，锁定关键比较变量s


```
97    if ( v17 )
98    {
99      c = 61;
100     memset(s, 49, v17);
101     memset(v26, c, v17);
102   }
103   v20 = v17;
104   while ( j < n )
105   {
106     v4 = v11;
107     *((_BYTE *)s + v20) = byte_400EB0[v11[j]];
108     *((_BYTE *)v26 + v20++) = byte_400EF0[v4[j++]];
109   }
110   *((_BYTE *)s + v20) = 0;
```

从后往前看，锁定与s相关的v11

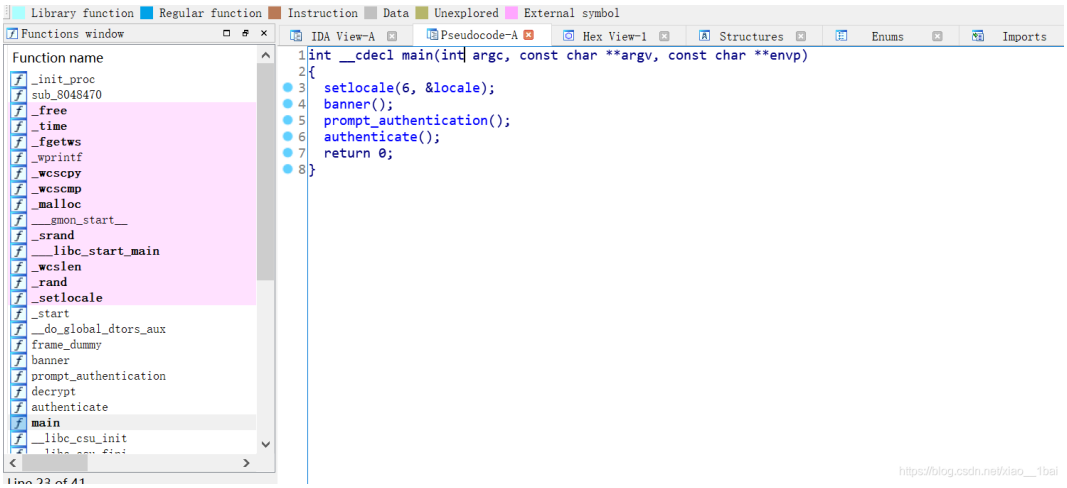


从后往前看，发现v11有类似base58的相关操作，由此判断是base58加密

函数逻辑封装类型：

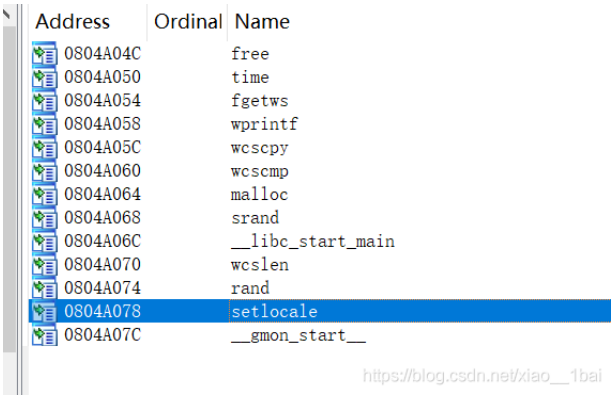
攻防世界的no-strings-attached：（函数名称暗示，GDB动态调试，小端）

32位ELF的linux文件，照例扔如IDA32位中查看代码信息，跟进Main函数：

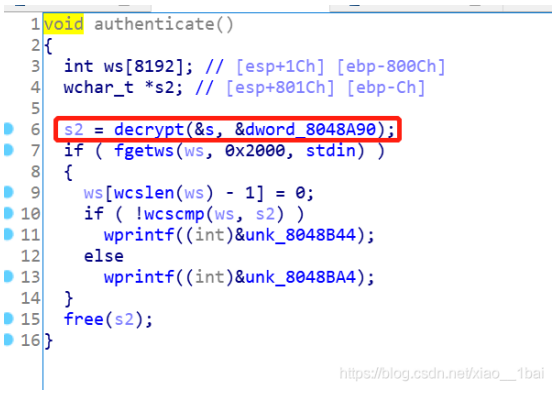


看到四个函数，由于才疏学浅，以为flag不在这里，还去查看了一下strings窗口。也没有flag字眼，有点懵(还是没觉得main的四个函数有问题，还是太菜了啊)。查了查资料，说flag操作就在这四个函数里，于是有回头去看这四个函数。

先看导入表，看那些是自带的函数：



可以看见第一个setlocale是自带的函数，第二第三个双击跟踪进去是打印函数，banner（横幅），猜测应该是打印开头信息的，那么就剩下第四个函数了，双击查看内容：



这里有个decrypt函数，中文名是加密，不在导入表中说明不是系统函数，后面的if判断条件是输入，还有个比较的wcscmp函数，后面两个wprintf分别是success 和access这些成功和拒绝的字符串地址。

fgetws函数是从输入流stdin中获取0x2000个字符给ws，也就是说s2是关键了，s2由decrypt函数得出，decrypt是用户自定义函数，在这里学到了非系统函数的英文名会是题目给的暗示，所以这里是加密操作后与输入的比较，只要输入后与加密后的s2一样就会打印success或access这些字符串，那flag自然也在加密函数中了。

```
wchar_t *__cdecl decrypt(wchar_t *s, wchar_t *a2)
{
  size_t v2; // eax
  signed int v4; // [esp+1Ch] [ebp-1Ch]
  signed int i; // [esp+20h] [ebp-18h]
  signed int v6; // [esp+24h] [ebp-14h]
  signed int v7; // [esp+28h] [ebp-10h]
  wchar_t *dest; // [esp+2Ch] [ebp-Ch]

  v6 = wcslen(s);
  v7 = wcslen(a2);
  v2 = wcslen(s);
  dest = (wchar_t *)malloc(v2 + 1);
  wcscpy(dest, s);
  while ( v4 < v6 )
  {
    for ( i = 0; i < v7 && v4 < v6; ++i )
      dest[v4++] -= a2[i];
  }
  return dest;
}
```


由于这种题是和用户输入的比较的，也就是说flag就在s2里面，我们可以在内存调试中提取s2的值，然后解密即可得到flag。(通常s2就是flag，因为如果s2还是加密的flag的话就不用玩了)

我还尝试print s2指令输出变量s2的值，因为我以为和IDA显示的一样，flag赋值给了s2,后来才想起IDA是根据自己的规则给无法解析变量名赋值的，也就是说在IDA里变量是s2这个名字，但是实际上程序里并没有s2这个变量名,所以只能查看寄存器了，毕竟函数是先返回到eax寄存器中再移动到变量中的。

还有就是admin的wp中给的是n指令然后查看eax寄存器的值，可是n指令执行的是一行高级语言命令，而ni和si才是单步执行一条汇编指令，所以不要调着调着跳过对应指令都不知道。

还有就是这里虽然是decrypt产出flag后赋值给了s2，但是双击s2跟踪显示的是s2初始的地址和值，而s2初始并没有什么东西，decrypt函数是用初始有值的&s进行加密操作后才产出flag赋值给s2的，所以不能用双击跟踪s2初始值的方式得到flag。

&s双击后跟进的字符串值：

```
.rodata:08048AA8 s               dd 143Ah
.rodata:08048AAC                 db  36h ; 6
.rodata:08048AAD                 db  14h
.rodata:08048AAE                 db    0
.rodata:08048AAF                 db    0
.rodata:08048AB0                 db  37h ; 7
.rodata:08048AB1                 db  14h
.rodata:08048AB2                 db    0
.rodata:08048AB3                 db    0
.rodata:08048AB4                 db  3Bh ; ;
.rodata:08048AB5                 db  14h
.rodata:08048AB6                 db    0
.rodata:08048AB7                 db    0
.rodata:08048AB8                 db  80h
.rodata:08048AB9                 db  14h
.rodata:08048ABA                 db    0
.rodata:08048ABB                 db    0
.rodata:08048ABC                 db  7Ah ; z
.rodata:08048ABD                 db  14h
.rodata:08048ABE                 db    0
.rodata:08048ABF                 db    0
.rodata:08048AC0                 db  71h ; q
.rodata:08048AC1                 db  14h
.rodata:08048AC2                 db    0
.rodata:08048AC3                 db    0
.rodata:08048AC4                 db  78h ; x
.rodata:08048AC5                 db  14h
```


GDB动态调试：

```
root@kali:/mnt/hgfs# gdb ./no_strings_attached
GNU gdb (Debian 8.2.1-2) 8.2.1
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./no_strings_attached...(no debugging symbols found)...done
.
```


gdb ./no_strings_attached 将文件加载到GDB中：

```
(gdb) b decrypt
Breakpoint 1 at 0x804865c
(gdb)
```

之前通过IDA，我们知道关键函数是decrypt,所以我们把断点设置在decrypt处，b在GDB中就是下断点的意思，即在decrypt处下断点：

```
(gdb) r
Starting program: /mnt/hgfs/no_strings_attached
Welcome to cyber malware control software.
Currently tracking 1878339819 bots worldwide

Breakpoint 1, 0x0804865c in decrypt ()
```

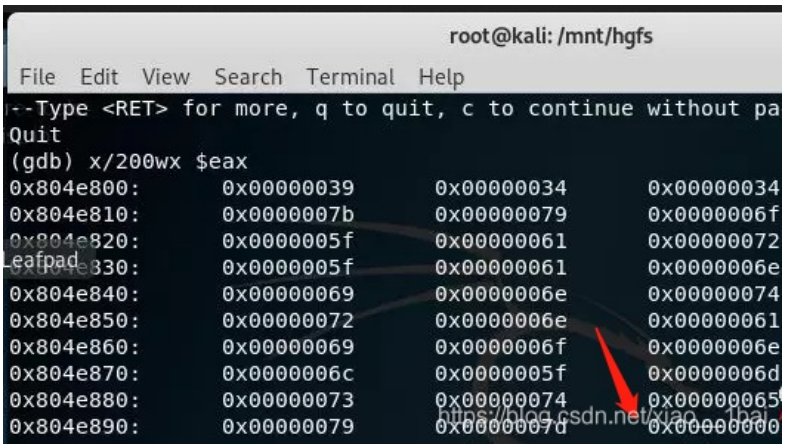
我们要的是经过decrypt函数，生成的字符串，所以我们这里就需要运行一步，GDB中用n来表示运行一步高级语言代码：

```
(gdb) n
Single stepping until exit from function dec
which has no line number information.
0x08048725 in authenticate ()
```

然后我们就需要去查看内存了，去查找最后生成的字符串：

```
.text:08048708 ; __unwind {
.text:08048708                 push    ebp
.text:08048709                 mov     ebp, esp
.text:0804870B                 sub     esp, 8028h
.text:08048711                 mov     dword ptr [esp+4], offset dword_8048A90 ; wchar_t *
.text:08048719                 mov     dword ptr [esp], offset s ; s
.text:08048720                 call    decrypt
.text:08048725                 mov     [ebp+s2], eax
.text:08048728                 mov     eax, ds:stdin@@GLIBC_2_0
.text:0804872D                 mov     [esp+8], eax    ; stream
.text:08048731                 mov     dword ptr [esp+4], 2000h ; n
.text:08048739                 lea     eax, [ebp+ws]
.text:0804873F                 mov     [esp], eax      ; ws
.text:08048742                 call    fgetws
```


通过IDA生成的汇编指令，我们可以看出进过decrypt函数后，生成的字符串保存在EAX寄存器中，所以，我们在GDB就去查看eax寄存器的值：



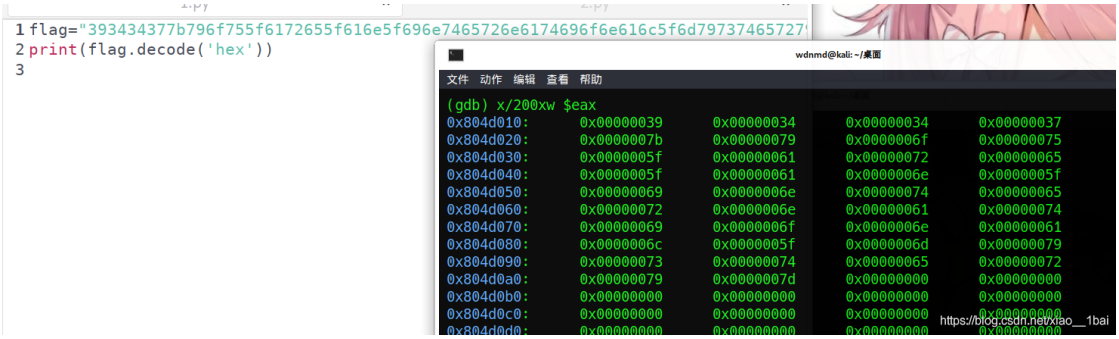
x:就是用来查看内存中数值的，后面的200代表查看多少个

x 代表是以word字节查看看

$ eax代表的eax寄存器中

在这里我们看到0x00000000，这就证明这个字符串结束了，因为，在C中，代表字符串结束的就是"\0",那么前面的就是经过decrypt函数生成的flag。

这里要特别注意一下：操作是面对反汇编低级语言来操作的，所以是对照着内存来操作的！



这里是内存数，所以不用像小端一样反过来(可能只有我才会傻到反过来吧~)，十六进制数解密后就是flag了：（注意，这里请用python2执行，具体原因看我的Python笔记）

flag="393434377b796f755f6172655f616e5f696e7465726e6174696f6e616c5f6d7973746572797d";

print(flag.decode('hex'))

静态仿写加密流程：

首先回顾前面的话：

由于这种题是和用户输入的比较的，也就是说flag就在s2里面，我们可以在内存调试中提取s2的值，然后解密即可得到flag。

flag在s2内，不用gdb查看内存的话s2就无法得知，但是s2是由decrypt这个加密函数得出的，而这里decrypt传入的加密参数&s和&dword_8048A90都可以双击跟踪内存查看初始值，而且decrypt的内部构造也有，那么我们直接提取出&s和&dword_8048A90这两个参数的值，然后仿照decrypt写个一样加密流程的脚本得出的不也是flag吗？

```
s2 = decrypt(&s, &dword_8048A90);
```

```
wchar_t *__cdecl decrypt(wchar_t *s, wchar_t *a2)
{
  size_t v2; // eax
  signed int v4; // [esp+1Ch] [ebp-1Ch]
  signed int i; // [esp+20h] [ebp-18h]
  signed int v6; // [esp+24h] [ebp-14h]
  signed int v7; // [esp+28h] [ebp-10h]
  wchar_t *dest; // [esp+2Ch] [ebp-Ch]

  v6 = wcslen(s);
  v7 = wcslen(a2);
  v2 = wcslen(s);
  dest = (wchar_t *)malloc(v2 + 1);
  wcscpy(dest, s);
  while ( v4 < v6 )
  {
    for ( i = 0; i < v7 && v4 < v6; ++i )
      dest[v4++] -= a2[i];
  }
  return dest;
}
```


所以我们去提取&s和&dword_8048A90的内容：

addr=0x08048AA8 #数组的地址

arr = []

for i in range(39): #数组的个数

arr.append(Dword(addr+4* i))

print(arr)

```
8048643: using guessed type int prompt_authentication(void);
8048708: using guessed type int authenticate(void);
[5178L, 5174L, 5175L, 5179L, 5248L, 5242L, 5233L, 5240L, 5219L, 5222L, 5235L, 5223L, 5218L, 5221L, 5235L, 5216L, 5227L, 5233L, 5240L, 5226L, 5235L, 5232L, 5220L, 5240L,
5230L, 5232L, 5232L, 5220L, 5232L, 5220L, 5230L, 5243L, 5238L, 5240L, 5226L, 5235L, 5243L, 5248L, 0L]
```

提取&dword_8048A90：

addr=0x08048A90 #数组的地址

arr = []

for i in range(6): #数组的个数

arr.append(Dword(addr+4* i))

print(arr)

```
[5121L, 5122L, 5123L, 5124L, 5125L, 0L]
80484B0: using guessed type int __cdecl wprintf(_DWORD);
```

然后就是用python仿照decrypt加密流程写脚本了：（注意：前面c++中v4++是先赋值后再加，所以到了python中v4+=1就放在赋值后面了）

s = [5178, 5174, 5175, 5179, 5248, 5242, 5233, 5240, 5219, 5222, 5235, 5223, 5218, 5221, 5235, 5216, 5227, 5233, 5240, 5226, 5235, 5232, 5220, 5240, 5230, 5232, 5232, 5220, 5232, 5220, 5230, 5243, 5238, 5240, 5226, 5235, 5243, 5248]

a = [5121, 5122, 5123, 5124, 5125]

v6 = len(s)

v7 = len(a)

v2 = len(s)

v4=0

while v4<v6:

for i in range(0,5):

if(i<v7 and v4<v6):

s[v4]-=a[i]

v4 += 1

else:

break

for i in range (38):

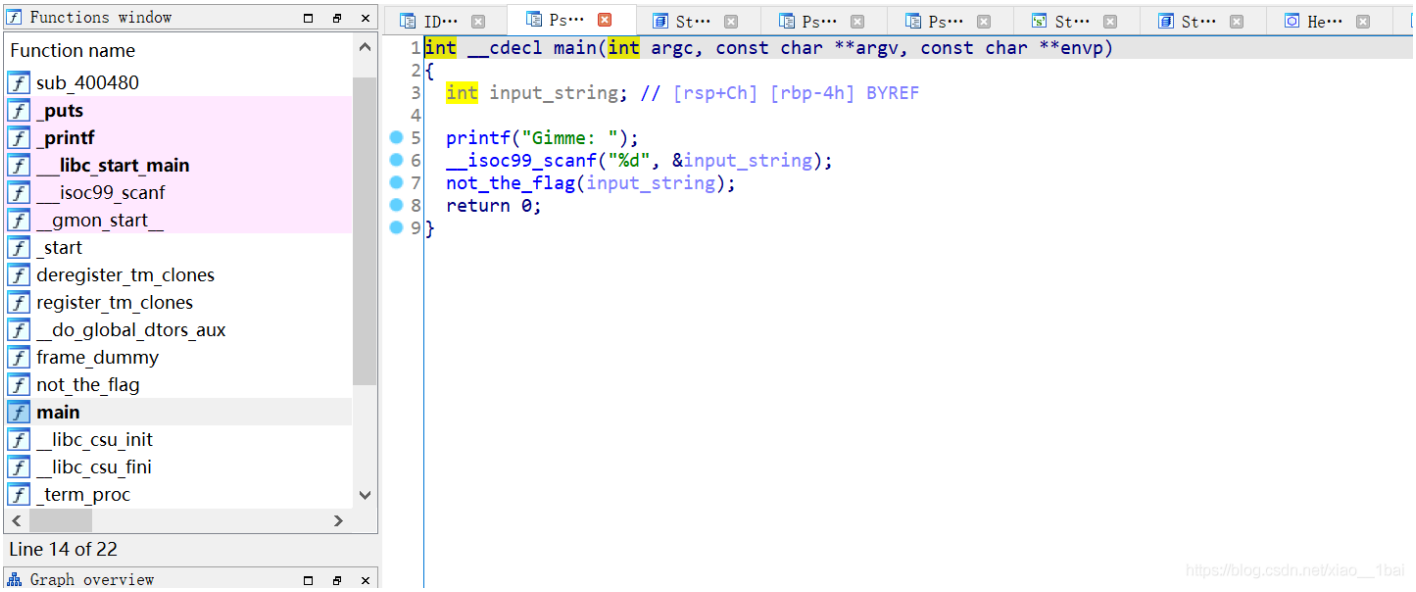
print(chr(s[i]),end="")

攻防世界answer_to_everything：（函数名称暗示、函数逻辑封装、出人意料的flag、题目描述暗示）

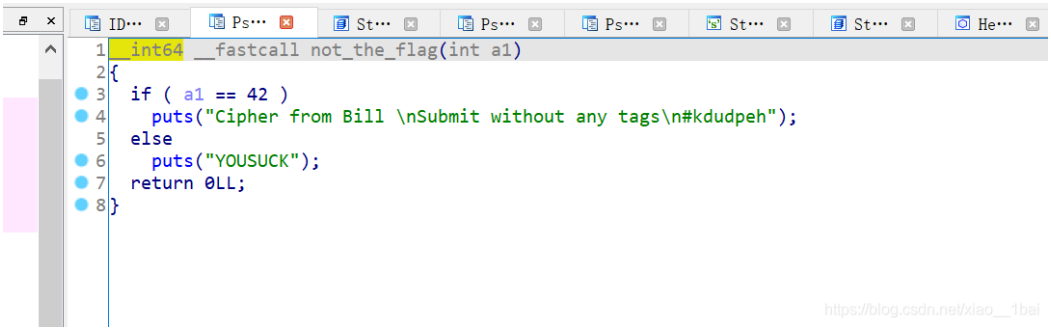
这里看题目犯下第一个错误：

题目中的人名原来可以包含重要信息的，比如这里的sha1就是sha1加密意思，原谅我年长无知。

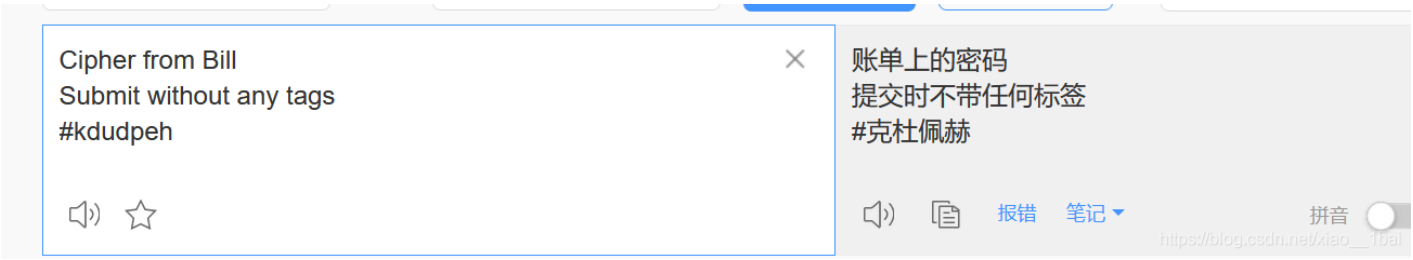
IDA静态分析：



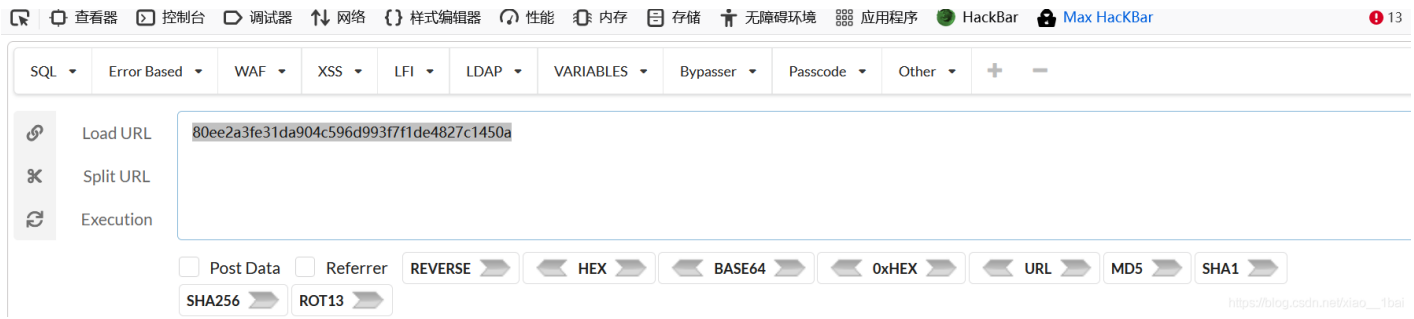
跟踪主函数，看到not_the_flag函数，进去看一下：



这里犯下第二个错误，我看到字符串以为真的是not_the_flag，然后看了其它函数也没发现有用信息，查了资料才发现这里就是flag，因为我没有把他翻译成中文，所以错过了重要提示！！！！



这里已经提示得很透彻了，提交时不要带标签，就是直接提交kdudpeh即可，结合错误1中的sha1人名，flag就是kdudpeh的sha1加密：

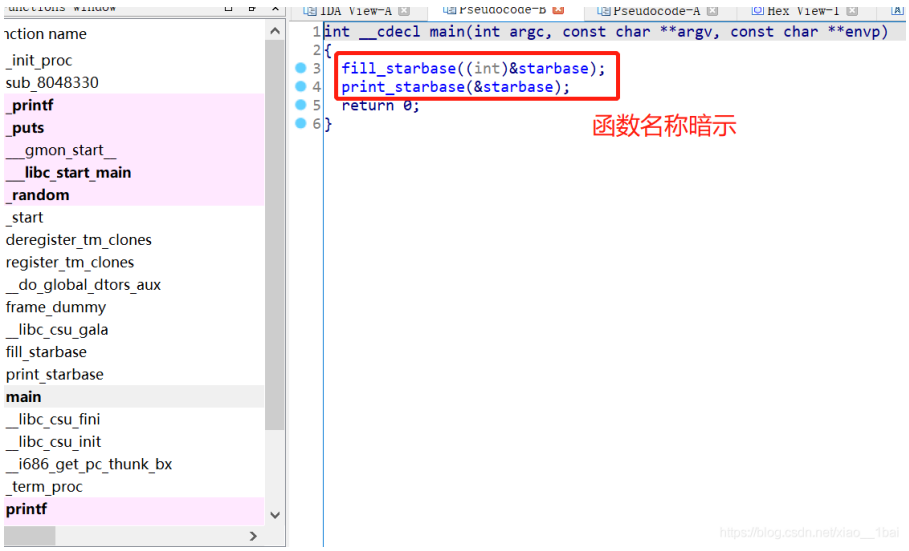


攻防世界secret-galaxy-300：（函数名称暗示、题目描述暗示、字符串拆分算法积累）

下载附件压缩包，解压，得到三个文件：

一开始我很震惊，以为是那种多文件关联的逆向题，结果不是，查看资料后发现这只是三个同一类型文件的三个不同版本而已，一个windows32位exe，另外两个分别是32位和64位的ELF的linux可执行文件，就分析32位的ELF文件吧。

扔入IDA32中查看伪代码，有main函数看main函数：



两个函数，一个填充fill_starbase，一个打印print_starbase，打印的函数跟踪进去没啥，打印一些横幅和其它信息，其中v2跟踪不了，看了一下是作为参数传入的：

```
int __cdecl print_starbase(int a1)
{
  int result; // eax
  const char *v2; // edx
  int i; // [esp+1Ch] [ebp-Ch]

  puts("--------------GALAXY DATABASE-------------");
  printf("%10s | %s | %s\n", "Galaxy name", "Existence of life", "Distance from Earth");
  result = puts("----------------------------------------");
  for ( i = 0; i <= 4; ++i )
  {
    if ( *(_DWORD *)(24 * i + a1 + 8) == 1 )
      v2 = "INHABITED";
    else
      v2 = "IS NOT INHABITED";
    result = printf("%11s | %17s | %d\n", *(const char **)(24 * i + a1), v2, *(_DWORD *)(24 * i + a1 + 4));
  }
  return result;
}
```

这里a1跟踪不了，因为是在外部的&starbase传入的，所以前面fill_starbase猜想是填充该数组的，双击跟踪：

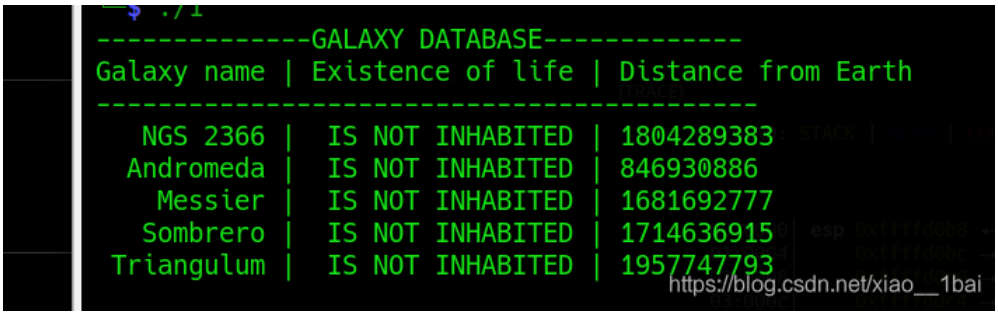
```
void __cdecl fill_starbase(int a1)
{
  int i; // [esp+8h] [ebp-10h]
  int v2; // [esp+Ch] [ebp-Ch]

  v2 = 0;
  for ( i = 0; i <= 4; ++i )
  {
    *(_DWORD *)(a1 + 24 * i) = (&galaxy_name)[i];
    *(_DWORD *)(24 * i + a1 + 4) = random();
    *(_DWORD *)(24 * i + a1 + 8) = 0;
    *(_DWORD *)(24 * i + a1 + 12) = 0;
    *(_DWORD *)(24 * i + a1 + 16) = 24 * (i + 1) + a1;
    *(_DWORD *)(a1 + 24 * i + 20) = v2;
    v2 = 24 * i + a1;
  }
}
```

看到一个数组&galaxy_name，还是取地址。后面是对它的一些运算，双击跟踪数组：

```
.data:08049B78                 public galaxy_name
.data:08049B78 galaxy_name     dd offset aNgs2366      ; DATA XREF: fill_starbase+30↑r
.data:08049B78                                        ; "NGS 2366"
.data:08049B7C off_8049B7C     dd offset aAndromeda    ; DATA XREF: __libc_csu_gala+36↑r
.data:08049B7C                                        ; __libc_csu_gala+60↑r ...
.data:08049B7C                                        ; "Andromeda"
.data:08049B80 off_8049B80     dd offset aMessier      ; DATA XREF: __libc_csu_gala+52↑r
.data:08049B80                                        ; __libc_csu_gala+7C↑r ...
.data:08049B80                                        ; "Messier"
.data:08049B84 off_8049B84     dd offset aSombrero     ; DATA XREF: __libc_csu_gala+AD↑r
.data:08049B84                                        ; "Sombrero"
.data:08049B88 off_8049B88     dd offset aTriangulum   ; DATA XREF: __libc_csu_gala+44↑r
.data:08049B88                                        ; __libc_csu_gala+EC↑r ...
.data:08049B88                                        ; "Triangulum"
.data:08049B8C off_8049B8C     dd offset aDarkSecretGala
.data:08049B8C                                        ; DATA XREF: __libc_csu_gala+D↑r
.data:08049B8C _data           ends                   ; "DARK SECRET GALAXY"
.data:08049B8C
.bss:08049B90 ; ================================================================================
```

看到这里有点不明觉厉，因为至始至终没有flag字眼，想起我还没运行过程序，就去运行一下：

(PS：这里犯下第一个错误：从一开始就运行程序可以帮助我门了解主要显示信息和判断隐藏信息，这里我现在才运行是太后了)



打印的信息在前面分析中都可以看到，这里犯下第二个错误：没有结合题目的暗示，题目是secret-galaxy-300，中文引导型暗示——隐藏的星系，运行结果显示了5个星系，而我前面跟踪的数组有6个星系，少了DARK SECRET GALAXY，那么这个就是关键点！

跟踪DARK SECRET GALAXY的调用，发现一个函数，代码分析如下：（这里是把一个星系字符串拆分成大量的单个字符逐个赋值，可以说是一种算法积累辨识了）

int _libc_csu_gala() //调用DARK SECRET GALAXY的函数

{

int result; // eax

sc[0] = off_8049B8C; // DARK SECRET GALAXY的地址

sc[3] = aAliensAreAroun; //一开始双击跟踪啥也没有，后面是对它的赋值操作

sc[1] = 31337;

sc[2] = 1;

aAliensAreAroun[0] = off_8049B7C[8]; //off_8049B7C处是Andromeda字符串的地址，是第一个星系

aAliensAreAroun[1] = off_8049B88[7]; //off_8049B88处是Triangulum字符串的地址，是第二个星系

aAliensAreAroun[2] = off_8049B80[4]; //off_8049B80是Messier字符串的地址，是第三个星系

aAliensAreAroun[3] = off_8049B7C[6];

aAliensAreAroun[4] = off_8049B7C[1];

aAliensAreAroun[5] = off_8049B80[2];

aAliensAreAroun[6] = 95; //_

aAliensAreAroun[7] = off_8049B7C[8];

aAliensAreAroun[8] = off_8049B7C[3];

aAliensAreAroun[9] = off_8049B84[5]; //off_8049B84是Sombrero的地址，是第四个星系

aAliensAreAroun[10] = 95; //_

aAliensAreAroun[11] = off_8049B7C[8];

aAliensAreAroun[12] = off_8049B7C[3];

aAliensAreAroun[13] = off_8049B7C[4];

aAliensAreAroun[14] = off_8049B88[6];

aAliensAreAroun[15] = off_8049B88[4];

aAliensAreAroun[16] = off_8049B7C[2];

aAliensAreAroun[17] = 95; //_

aAliensAreAroun[18] = off_8049B88[6];

result = (unsigned __int8)off_8049B80[3];

aAliensAreAroun[19] = off_8049B80[3];

aAliensAreAroun[20] = 0;

return result; //这里犯下第三个错误，返回result，可是result是off_8049B80[3]，就是Messier的第三个字符s，我醉了，难怪不显示。因为前面一直在用 aAliensAreAroun，结果这里返回别的东西去了。

}

分析完后可以知道 aAliensAreAroun数组大概就是我们要找的flag了：

(PS：可能是我已经运行且调试过IDA了，所以这里的数组名字和我一开始看到的不一样，IDA应该是自己又修改过了)

第一种方法：

手动调试，就这样不同的字符串一个个截取对应的位拼接即可。

第二种方法：

IDA远程动态调试，下断点在return处，运行：

```
 9  aAliensAreAroun[0] = off_8049B7C[8];
10  aAliensAreAroun[1] = off_8049B88[7];
11  aAliensAreAroun[2] = off_8049B80[4];
12  aAliensAreAroun[3] = off_8049B7C[6];
13  aAliensAreAroun[4] = off_8049B7C[1];
14  aAliensAreAroun[5] = off_8049B80[2];
15  aAliensAreAroun[6] = 95;
16  aAliensAreAroun[7] = off_8049B7C[8];
17  aAliensAreAroun[8] = off_8049B7C[3];
18  aAliensAreAroun[9] = off_8049B84[5];
19  aAliensAreAroun[10] = 95;
20  aAliensAreAroun[11] = off_8049B7C[8];
21  aAliensAreAroun[12] = off_8049B7C[3];
22  aAliensAreAroun[13] = off_8049B7C[4];
23  aAliensAreAroun[14] = off_8049B88[6];
24  aAliensAreAroun[15] = off_8049B88[4];
25  aAliensAreAroun[16] = off_8049B7C[2];
26  aAliensAreAroun[17] = 95;
27  aAliensAreAroun[18] = off_8049B88[6];
28  result = (unsigned __int8)off_8049B80[3];
29  aAliensAreAroun[19] = off_8049B80[3];
30  aAliensAreAroun[20] = 0;
31  return result;
32 }
```

双击跟踪 aAliensAreAroun，按a键生成数组：（a键是IDA生成数组的热键）

```
.bss:08049C0C ; _DWORD sc[10]
.bss:08049C0C sc dd 804886Fh, 7A69h, 1, 8049C34h, 0, 0, 0, 0, 0, 0
.bss:08049C0C                                         ; DATA XREF: __libc_csu_gala+6↑o
.bss:08049C34 aAliensAreAroun db 'aliens_are_around_us',0
.bss:08049C34                                         ; DATA XREF: __libc_csu_gala+1B↑o
.bss:08049C34                                         ; __libc_csu_gala+3F↑w ...
.bss:08049C49 align 4
.bss:08049C49 _bss ends
.bss:08049C49
.prgend:08049C4C ; ===========================================================================
.prgend:08049C4C
```

结果就是aliens_are_around_us

GDB动态调试：

b *0x80485bc //下断点（32位的ELF文件才是这个内存地址啊！其他的不是）

run //运行

x/s 0x8049C34 //查看aAliensAreAroun数组内存

```
0x8049d0d:      ""
0x8049d0e:      ""
0x8049d0f:      ""
pwndbg> x/sb 0x8049C34
0x8049c34 <sc+40>:      "aliens_are_around_us"
pwndbg> x/s 0x8049C34
0x8049c34 <sc+40>:      "aliens_are_around_us"
pwndbg> Quit
pwndbg>
```

攻防世界simple-check-100：（IDA动态调试、GDB动态调试）

下载附件，又是三个同一类型不同版本的附件，还以为终于遇到了那种关联文件的逆向题：用win32文件，照例扔入IDA32中查看伪代码，有main函数看main函数：

```
  printf("Key: ");
  scanf("%s", v8);
  if ( check_key(v8) )
    interesting_function(&v7);
  else
    puts("Wrong");
  return 0;
}
```

关键代码如上，输入和检查判断是v8，而v8=&v6，也就是说我们是在v6地址上操作。

这里犯下第一个错误：

我看v6栈地址的时候发现编译器给v6留了好多空间，但我竟然以为这题没有这么简单，我以为我么输入的v8会覆盖v7~v35这些地址，如果会覆盖的话题型就变成与用户输入有关的生成型flag了，就不能靠简单修改跳转点来做了，毕竟我们的输入会修改数据，可结果就是v6空间大到我们输入的数据不会覆盖其它变量的数据，真是多想了！

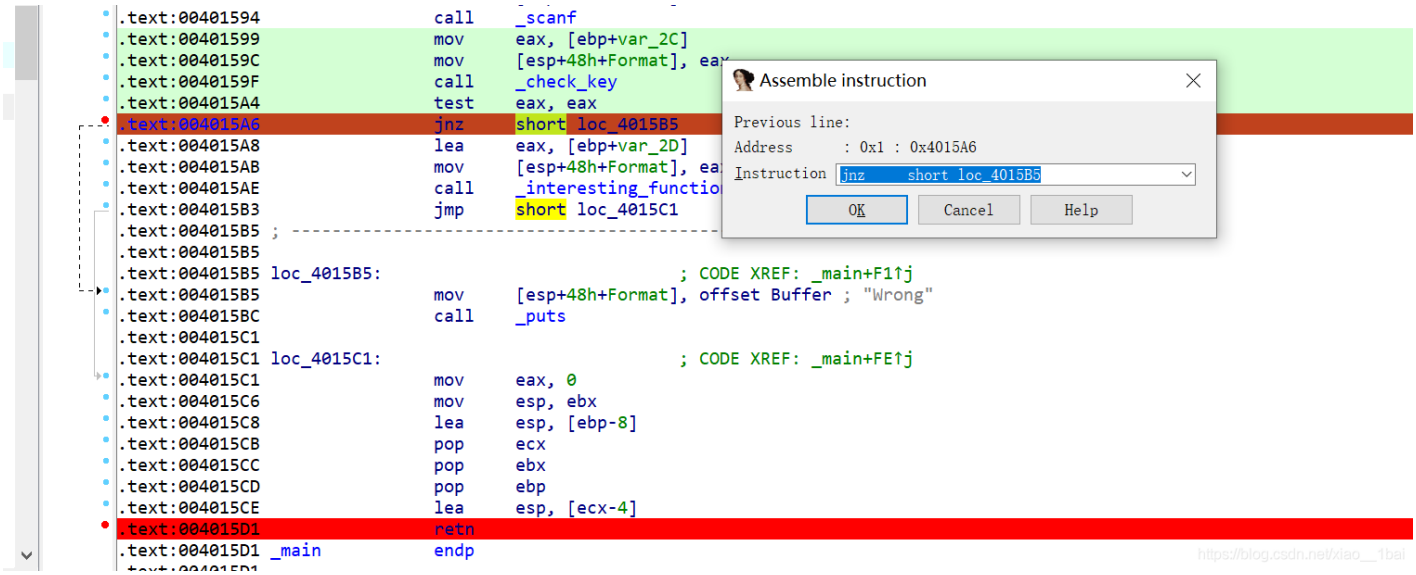
(下图是v6空间，从40~2D,够大了!)

```
-00000040 var_40          db ?
-0000003F                 db ? ; undefined
-0000003E                 db ? ; undefined
-0000003D                 db ? ; undefined
-0000003C                 db ? ; undefined
-0000003B                 db ? ; undefined
-0000003A                 db ? ; undefined
-00000039                 db ? ; undefined
-00000038                 db ? ; undefined
-00000037                 db ? ; undefined
-00000036                 db ? ; undefined
-00000035                 db ? ; undefined
-00000034                 db ? ; undefined
-00000033                 db ? ; undefined
-00000032                 db ? ; undefined
-00000031                 db ? ; undefined
-00000030                 db ? ; undefined
-0000002F                 db ? ; undefined
-0000002E                 db ? ; undefined
-0000002D var_2D          db ?
```

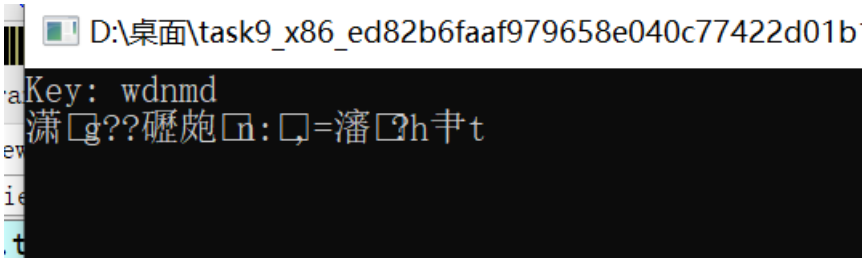

所以我们简单修改跳转条件输出调用后面生成flag的函数即可：

```
if ( check_key(v8) )
  interesting_function(&v7);
else
```

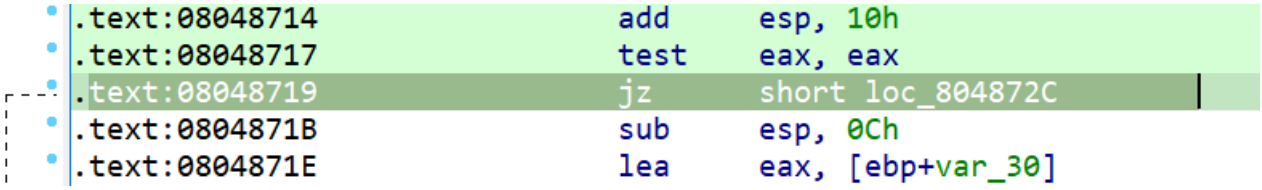
直接用刚学到的IDA本地调试：（修改jz 为 jnz）



额，乱码了:



换linux32位来试，继续扔ELF32位入IDA中查看对应代码在虚拟内存中的位置，好下断点：(在8048719处):



Llnux GDB调试，代码如下：

b *0x8048717 //判断位置test eax eax处下断点

r

set var $eax=1 //这里犯下第二个错误：因为前面位运算语句是test eax eax，我们没法直接修改状态标志位ZF=0或修改jz为jnz， 所以我们直接修改eax让test eax eax使ZF=0

c

结果：

```
► f 0 0x8048719 main+259
  f 1 0xf7ddfe46 __libc_start_main+262
```

```
Key: wdnmd

Breakpoint 2, 0x08048717 in main ()
pwndbg> set var $eax=1
pwndbg> c
Continuing.

Breakpoint 1, 0x08048719 in main ()
pwndbg> c
Continuing.
flag_is_you_know_cracking!!![Inferior 1 (process 2580) exited normally]
pwndbg> Quit
pwndbg>
```


攻防世界re1-100：（函数逻辑封装、出人意料的flag、非预期行为）

64位ELF文件，照例扔入IDA64中查看伪代码信息，有main函数看main函数：

```
1  int __cdecl __noreturn main(int argc, const char **argv, const char **envp)
2  {
3    __pid_t v3; // eax
4    size_t v4; // rax
5    ssize_t v5; // rbx
6    bool v6; // al
7    bool bCheckPtrace; // [rsp+13h] [rbp-1BDh]
8    ssize_t numRead; // [rsp+18h] [rbp-1B8h]
9    ssize_t numReada; // [rsp+18h] [rbp-1B8h]
10   char bufWrite[200]; // [rsp+20h] [rbp-1B0h] BYREF
11   char bufParentRead[200]; // [rsp+F0h] [rbp-E0h] BYREF
12   unsigned __int64 v12; // [rsp+1B8h] [rbp-18h]
13
14   v12 = __readfsqword(0x28u);
15   bCheckPtrace = detectDebugging();
16   if ( pipe(pParentWrite) == -1 )
17     exit(1);
18   if ( pipe(pParentRead) == -1 )
19     exit(1);
20   v3 = fork();
21   if ( v3 != -1 )
22   {
23     if ( v3 )
24     {
25       close(pParentWrite[0]);
26       close(pParentRead[1]);
27       while ( 1 )
28       {
```

进入此自定义函数

000084B4 main:19 (4084B4)

```
printf( "input key : " );
memset(bufWrite, 0, sizeof(bufWrite));
gets(bufWrite);
v4 = strlen(bufWrite);
v5 = write(pParentWrite[1], bufWrite, v4);
if ( v5 != strlen(bufWrite) )
  printf("parent - partial/failed write");
do
{
  memset(bufParentRead, 0, sizeof(bufParentRead));
  numReada = read(pParentRead[0], bufParentRead, 0xC8uLL);
  v6 = bCheckPtrace || checkDebuggerProcessRunning();
  if ( !v6 && checkStringIsNumber(bufParentRead) && atoi(bufParentRead) )
  {
    puts("True");
    if ( close(pParentWrite[1]) == -1 )
      exit(1);
    exit(0);
  }
  puts("Wrong !!!\n");
```

继续进入关键自定义函数

这里犯下第一个错误，前面是一堆系统函数，我知道系统函数通常不是关键，但是它系统函数中又混杂了字符串，加上我之前写的HOOK题，还以为藏了什么重要信息在里面，后来才发现关键逻辑代码在后面。

所以以后遇到这种系统函数多的题目先浏览一下全局，看看系统函数外有没有关键逻辑代码：

```
memset(bufParentRead, 0, sizeof(bufParentRead));
numRead = read(pParentWrite[0], bufParentRead, 0xC8uLL);
if ( numRead == -1 )
  break;
if ( numRead )
{
  if ( !childCheckDebugResult()
    && bufParentRead[0] == '{'
    && strlen(bufParentRead) == 42
    && !strncmp(&bufParentRead[1], "53fc275d81", 10uLL)
    && bufParentRead[strlen(bufParentRead) - 1] == '}'
    && !strncmp(&bufParentRead[31], "4938ae4efd", 10uLL)
    && confuseKey(bufParentRead, 42)
    && !strncmp(bufParentRead, "{daf29f59034938ae4efd53fc275d81053ed5be8c}", 42uLL) )
  {
    responseTrue();
  }
  else
  {
    responseFalse();
```

发现关键代码

前面是对&bufParentRead[1]的开头十个赋值，后面&bufParentRead[31]是对倒数十个赋值，但是后面顺序又乱掉了：strncmp(bufParentRead, “{daf29f59034938ae4efd53fc275d81053ed5be8c}”, 42uLL)

所以中间一定有改变，跟踪一下中间的confuseKey(bufParentRead, 42)函数：

```
22  *(_QWORD *)szPart4 = 0LL;
23  *(_DWORD *)&szPart4[8] = 0;
24  *(_WORD *)&szPart4[12] = 0;
25  szPart4[14] = 0;
26  if ( iKeyLength != 42 )
27    return 0;
28  if ( !szKey )
29    return 0;
30  if ( strlen(szKey) != 42 )
31    return 0;
32  if ( *szKey != '{' )
33    return 0;
34  strncpy(szPart1, szKey + 1, 10uLL);
35  strncpy(szPart2, szKey + 11, 10uLL);
36  strncpy(szPart3, szKey + 21, 10uLL);
37  strncpy(szPart4, szKey + 31, 10uLL);
38  memset(szKey, 0, 0x2AuLL);
39  *szKey = '{';
40  strcat(szKey, szPart3);
41  strcat(szKey, szPart4);
42  strcat(szKey, szPart1);
43  strcat(szKey, szPart2);
44  szKey[41] = '}';
45  return 1;
46}
```

前面比较多东西，但是这次我忽然看到后面的关键了，如截图所示，把字符串分成四份，按3、4、1、2、的顺序重新打乱，而且按照主函数最后混乱代码那里{daf29f59034938ae4efd53fc275d81053ed5be8c}也的确是符合4和1的新顺序。

所以之前的函数顺序就是简单的1、2、3、4、：

{53fc275d81053ed5be8cdaf29f59034938ae4efd}

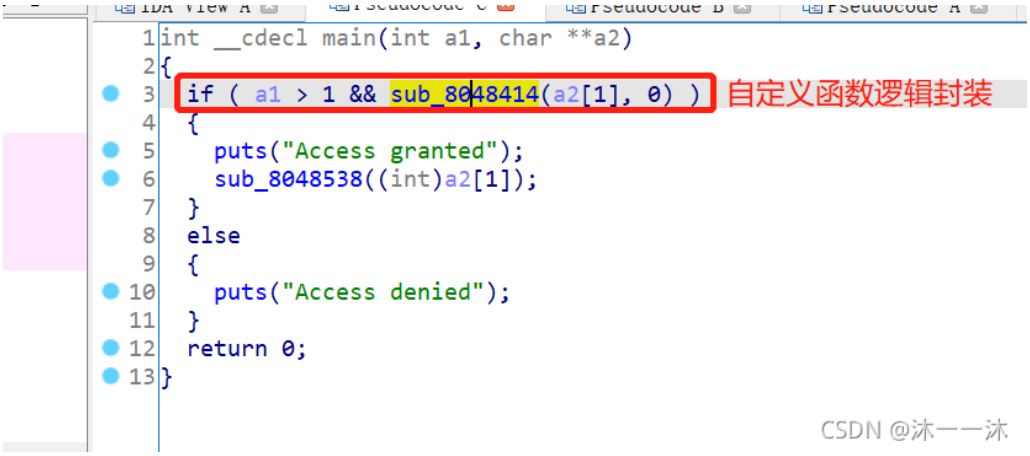
好像很简单，但是提交的时候显示错误：

这里就犯下第二个错误了，既然题目没有flag模板，我加flag变成flag{53fc275d81053ed5be8cdaf29f59034938ae4efd}还是提交错误，那这里就应该去掉花括号啊，结果就是53fc275d81053ed5be8cdaf29f59034938ae4efd

攻防世界elrond32：（argv[]外部调用输入参数符合条件、函数逻辑封装、递归调用算法）

32位ELF文件，无壳，扔入32位IDA中查看伪代码信息，有Main函数看main函数：

额，看上去好像比较简单，int __cdecl main(int a1, char **a2)中a1是命令行传入参数的个数，起始值为1，a2是命令行传入参数的数组，a2[0]存的是程序名称，所以才有a1的起始1。我们传入的参数从a2[1]开始。



跟踪sub_8048414函数：（递归调用算法）

int __cdecl sub_8048414(_BYTE *input_flag, int a2)

{

int result; // eax

switch (a2) // a2=0，从0开始，然后后面递归重新调用此函数时会对a2重新赋值，每次+1，按顺序对应case的不同情况，以此按顺序锁定flag每个字符。

{

case 0:

if (*input_flag == 'i')

goto LABEL_19;

result = 0;

break;

case 1:

```
if ( *input_flag == 'e' )
goto LABEL_19;
result = 0;
break;
case 3:
if ( *input_flag == 'n' )
goto LABEL_19;
result = 0;
break;
case 4:
if ( *input_flag == 'd' )
goto LABEL_19;
result = 0;
break;
case 5:
if ( *input_flag == 'a' )
goto LABEL_19;
result = 0;
break;
case 6:
if ( *input_flag == 'g' )
goto LABEL_19;
result = 0;
break;
case 7:
if ( *input_flag == 's' )
goto LABEL_19;
result = 0;
break;
case 9:
if ( *input_flag == 'r' )
```

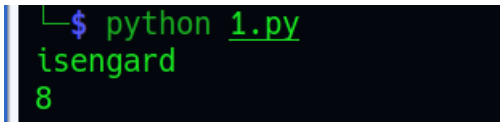
```
LABEL_19:
result = sub_8048414(input_flag + 1, 7 * (a2 + 1) % 11); // 修改input_flag的地址，a2重新赋值，递归调用。
else
result = 0;
break;
default:
result = 1;
break;
}
return result;
}
```

这里我们先写脚本正向逆向（仿写）这个逻辑先，只要保证每个返回的都是1即可，递归调用我们用大量循环来写，反正不符合结果就会跳出：

```
a2=0
flag=""
for i in range(32):
a=a2
if a==0:
flag+='i'
elif a==1:
flag+='e'
elif a==3:
flag+='n'
elif a==4:
flag+='d'
elif a==5:
flag+='a'
elif a==6:
flag+='g'
elif a==7:
flag+='s'
```

```
elif a==9: #python写c语言的
flag+='r'
else:
break
a2=7 * (a2 + 1) % 11
print(flag)
print(len(flag))
```

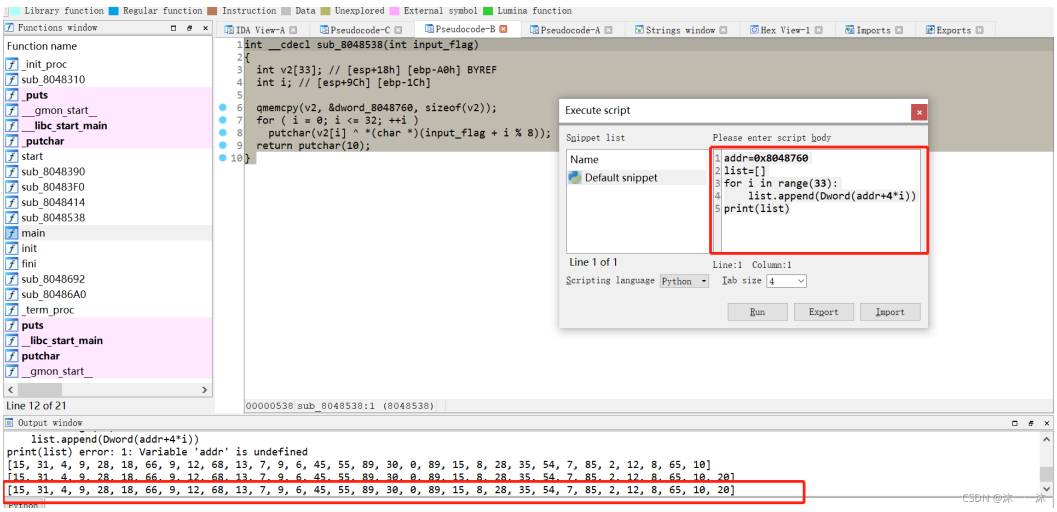
结果，生成一个8位的字符串就退出了：



跟踪下一个函数，sub_8048538((int)a2[1])，发现flag要对前面生成的8位字符进一步操作才行。

```
int __cdecl sub_8048538(int input_flag)
{
int v2[33]; // [esp+18h] [ebp-A0h] BYREF
int i; // [esp+9Ch] [ebp-1Ch]
qmemcpy(v2, &dword_8048760, sizeof(v2));
for ( i = 0; i <= 32; ++i )
putchar(v2[i] ^ *(char *)(input_flag + i % 8)); //一个简单的异或操作然后输出，%8对得上前面输出的8位字符串
return putchar(10);
}
```

写IDA脚本打印dword_8048760数组内容：



复制粘贴数组，重新修改脚本内容：

```
a2=0
```

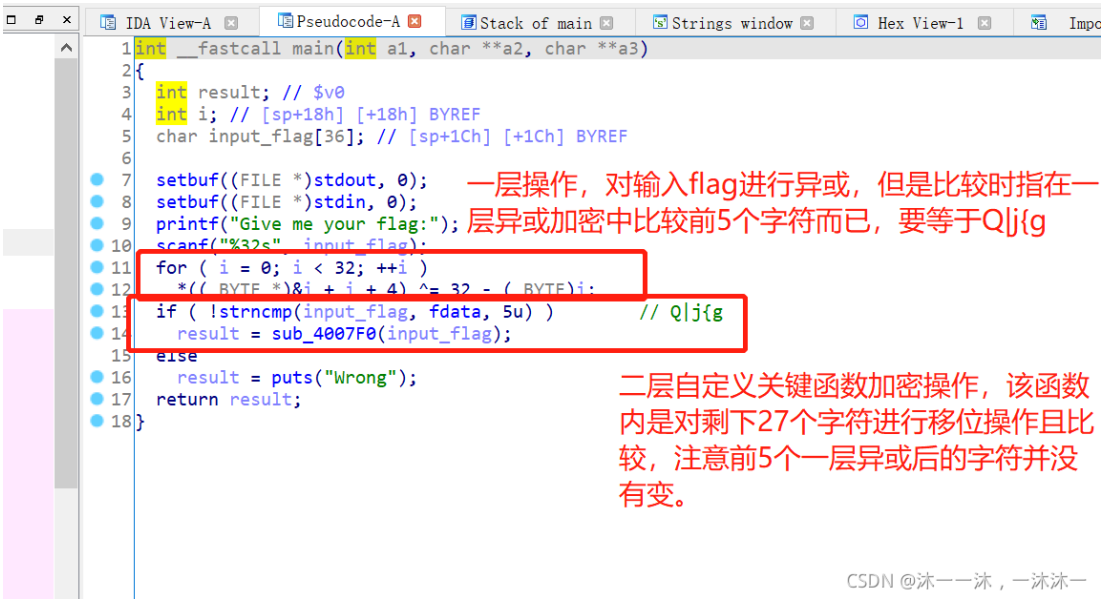
```
flag=""

v2=[15, 31, 4, 9, 28, 18, 66, 9, 12, 68, 13, 7, 9, 6, 45, 55, 89, 30, 0, 89, 15, 8, 28, 35, 54, 7, 85, 2, 12, 8, 65, 10,20]

flag2=""

for i in range(32):

a=a2

if a==0:

flag+='i'

elif a==1:

flag+='e'

elif a==3:

flag+='n'

elif a==4:

flag+='d'

elif a==5:

flag+='a'

elif a==6:

flag+='g'

elif a==7:

flag+='s'

elif a==9: #python写c语言的

flag+='r'

else:

break

a2=7 * (a2 + 1) % 11

print(flag)

print(len(flag))

for i in range(33):

flag2+=chr(v2[i]^ord(flag[i%8]))

print(flag2)
```

结果：

```
isengard
8
flag{s0me7hing_S0me7hinG_t0lki3n}
```

攻防世界babymips：（多层加密操作、函数逻辑封装、移位算法积累、取限制位数算法、奇数偶数判断算法）

32位无壳，照例扔入IDA32中查看伪代码，有main函数看main函数：



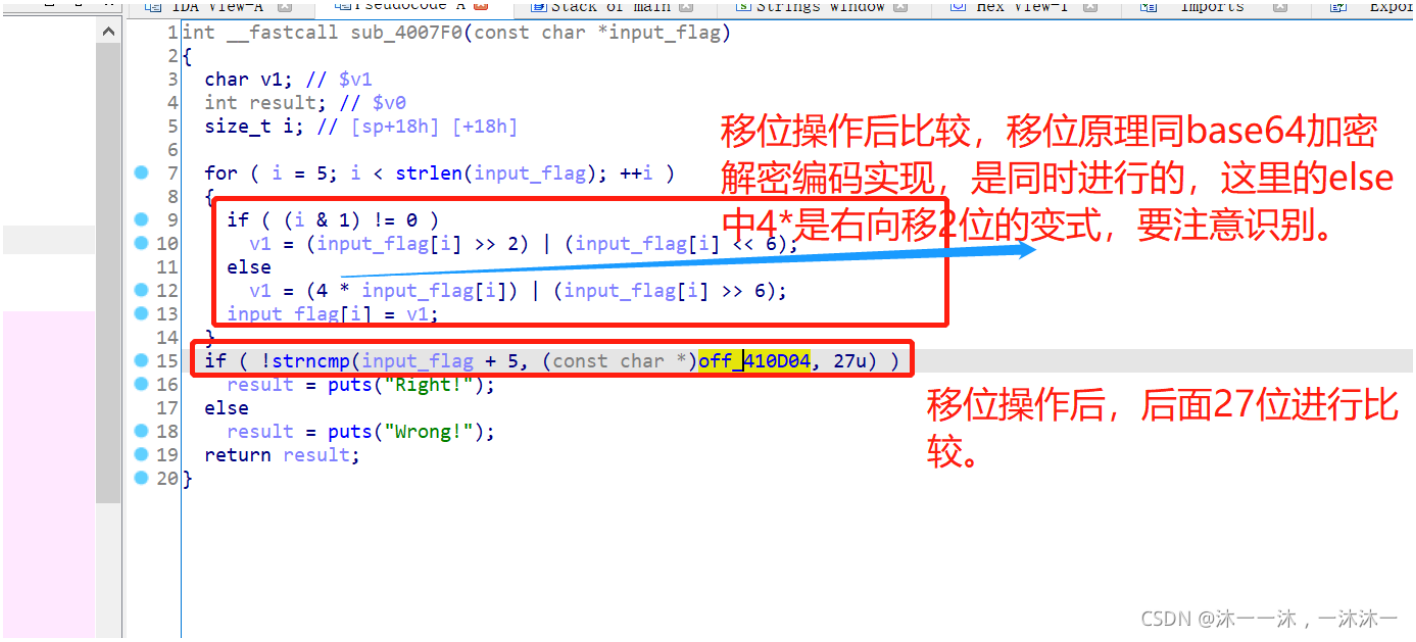
(这里积累第一个经验)

第一个框是对用户输入进行每位的异或操作，因为i的地址和input_flag的地址在main函数栈中相差4而已。

第二个框就是取异或后的前五个字符与明文比较，如果相同就继续对剩下的27个字符继续操作。这里要注意的是sub_4007F0(input_flag)函数是在前面对input_flag异或后的基础上进一步操作，也就是双层操作，而input_flag是单层的对前5个字符的异或操作。

我犯的错误就是逆向后面27位字符后忘记了前面还有一层异或操作，以至于生成错误的答案。

跟踪sub_4007F0函数，一个移位一个比较：

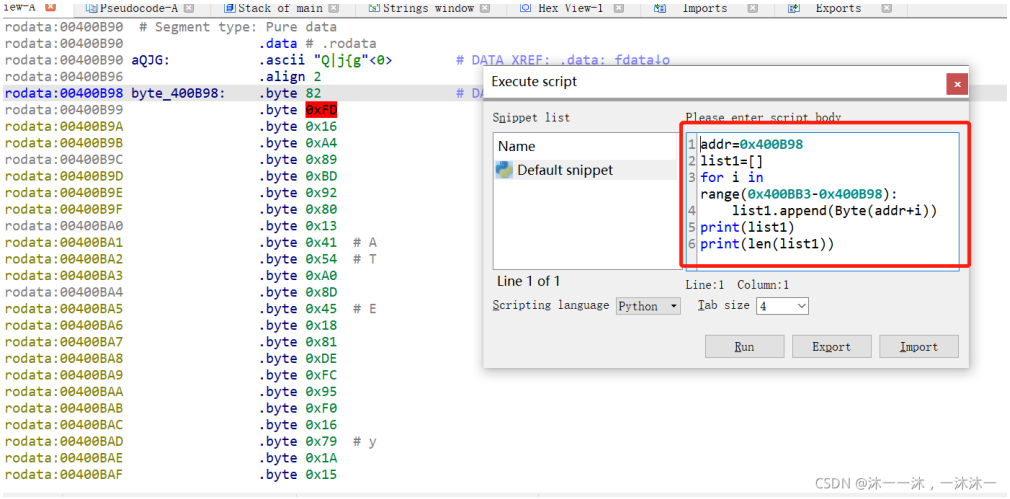


附上移位的汇编指令笔记：

SHL	左移	ROR	循环右移
SHR	右移	RCL	带进位的循环左移
SAL	算术左移	RCR	带进位的循环右移
SAR	算术右移	SHLD	双精度左移
ROL	循环左移	SHRD	双精度右移

然后移完位之后和数组off_410D04内容比较，这里嵌入IDA脚本dump下来：

这里.byte一开始我也懵了一下，后来发现用Byte也可以获取，.byte可能是MIPS的一个特性吧：



知道是移位操作之后就可以写脚本了，原代码逻辑中(i&1)其实是判断奇数还是偶数来的，照抄即可，然后移位逻辑是以8位为一个循环的，所以我们要用&FF来限制在8位才行：

key1="Q|j{g"

flag=list(map(ord,key1))

flag+=[82, 253, 22, 164, 137, 189, 146, 128, 19, 65, 84, 160, 141, 69, 24, 129, 222, 252, 149, 240, 22, 121, 26, 21, 91, 117, 31]

print(flag)

for a in range(5,32,1): #这里循环从5开始，因为这里解决的是二层循环

if (a&1)!=0:

flag[a]=((flag[a]<<2)|(flag[a]>>6))&0xFF #移位操作是8位循环的，必须要有位数限制

else:

flag[a]=((flag[a]>>2)|(flag[a]<<6))&0xFF

for i in range(32,):

flag[i]=chr(flag[i]^(32-i))

print(''.join(flag))

main函数中有与本地文件相关的操作类型：

攻防世界getit：（IDA动态调试、GDB动态调试）

64位ELF文件，无壳，然后扔入IDA查看伪代码信息：


```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char v3; // al
  __int64 v5; // [rsp+0h] [rbp-40h]
  int i; // [rsp+4h] [rbp-3Ch]
  FILE *stream; // [rsp+8h] [rbp-38h]
  char filename[8]; // [rsp+10h] [rbp-30h]
  unsigned __int64 v9; // [rsp+28h] [rbp-18h]

  v9 = __readfsqword(0x28u);
  LODWORD(v5) = 0;
  while ( (signed int)v5 < strlen(s) )
  {
    if ( v5 & 1 )
      v3 = 1;
    else
      v3 = -1;
    *(&t + (signed int)v5 + 10) = s[(signed int)v5] + v3;
    LODWORD(v5) = v5 + 1;
  }
  strcpy(filename, "/tmp/flag.txt");
  stream = fopen(filename, "w");
  fprintf(stream, "%s\n", u, v5);
  for ( i = 0; i < strlen(&t); ++i )
  {
    fseek(stream, p[i], 0);
    fputc(*(&t + p[i]), stream);
    fseek(stream, 0LL, 0);
    fprintf(stream, "%s\n", u);
  }
  fclose(stream);
  remove(filename);
```

这里说写入了/tmp/flag.txt，也因为是tmp文件，所以不是管理员也可以写入，后面显眼的fclose(stream);remove(filename);也说明了一运行完程序就删除文件，所以我们没法在运行完程序后找到该文件

现在分析中间的循环写入语句：

for (i = 0; i < strlen(&t); ++i) //我是认为&t是flag的长度

{

fseek(stream, p[i], 0); //定位到开头偏移p[i]位置处，

fputc(*(&t + p[i]), stream); //在上一句的定位处写入&t起始字符串的p[i]偏移的单个字节

fseek(stream, 0LL, 0); //重新定位到0且没有偏移

fprintf(stream, "%s\n", u); //写入完整字符u来覆盖前面写入的一个字符，双击跟踪u发现是**********，就是一个覆盖干扰

}

这里的p[i]双击跟踪长这个样子：(p[i]鼠标长时间停留显示int[43]类型)：

可以看到p[i]数组存放的是无序的整数，但是这些整数都是唯一的flag的位数。也就是说每次在/tmp/flag.txt文件中写入的flag是不按顺序写的，且每次只出现一个字符，需要自己排序。

```
.data:0000000000601160 p                dd 1Eh
.data:0000000000601160
.data:0000000000601164                  db  18h
.data:0000000000601165                  db    0
.data:0000000000601166                  db    0
.data:0000000000601167                  db    0
.data:0000000000601168                  db  19h
.data:0000000000601169                  db    0
.data:000000000060116A                  db    0
.data:000000000060116B                  db    0
.data:000000000060116C                  db  20h
.data:000000000060116D                  db    0
.data:000000000060116E                  db    0
.data:000000000060116F                  db    0
.data:0000000000601170                  db  28h ; (
.data:0000000000601171                  db    0
.data:0000000000601172                  db    0
.data:0000000000601173                  db    0
.data:0000000000601174                  db  24h ; $
.data:0000000000601175                  db    0
```


开头的循环判断中的&t在前面出现过，长这个样子：

```
*(&t + (signed int)v5 + 10) = s[(signed int)v5] + v3;
```

后来发现前面的循环才是生成的flag代码，后面的p[i]只是将flag分成单个单个字符写进去而已：

v9 = __readfsqword(40u); //这里我也不清楚，是windows的API函数,从偏移量的指定位置读取内存相对 FS 段开头

LODWORD(v5) = 0; //这里把__in64的v5(longlong型)转地址类型为32为DWORD，即取低32位为0

while ((signed int)v5 < strlen(s)) //这里signed int是有符号整形，s双击跟踪是c61b68366edeb7bdce3c6820314b7498这样的无序字符串

{

if (v5 & 1) //这里是v5的1位和1进行与操作，因为后面有LODWORD(v5) = v5 + 1，所以v3在这里会1、-1这样反复横条strlen(s)次

v3 = 1;

else

v3 = -1;

*(&t + (signed int)v5 + 10) = s[(signed int)v5] + v3; //这里是赋值，因为v5一开始是0，所以从&t+10开始赋值，&t双击跟踪是SharifCTF{??????????????????????????????}，第10位刚好是第一个?,赋值是从s[0]开始+1,s[1]-1，s[2]+1这样赋值，分别对应不断递增的v5和反复横跳的v3，所以这里也可以手动计算第一个?是b，第二个?是7这样。

LODWORD(v5) = v5 + 1; //v5底32位加1，其实就是v5加1

}

有好几种做题方法，

第一个静态计算，就像我上面分析flag生成代码手动计算一样。

手动计算即可。

第二种动态截停，截止flag生成后的位置，查看寄存器即可。

GDB动态调试，首先我们知道了下面strlen(&t)的t是flag程序运行后生成的flag,那我们把鼠标放在那一行上看一下下面的反汇编行数，如下所示是400824，那么我们在反汇编窗口跟上。

```
.text:0000000000400819                  mov     eax, [rbp+var_3C]
.text:000000000040081C                  movsxd  rbx, eax
.text:000000000040081F                  mov     edi, offset t   ; s
.text:0000000000400824                  call    _strlen
.text:0000000000400829                  cmp     rbx, rax
.text:000000000040082C                  jnb     loc_4008B5
```

可以看到反汇编中400824行的确是_strlen函数，而它上面就是把&t移入了edi，所以在GDB中我们断点400824，然后查看edi寄存器即可。

GDB所需命令：（可以看到flag就在RDI寄存器里）

b *0x400824

r

第三种静态计算，仿写c语言脚本或python脚本安装一样的算法生成flag。

key1="c61b68366edeb7bdce3c6820314b7498"

v5=0

flag=""

while v5 < len(key1):

if v5 & 1:

v3=1

else:

v3=-1

flag+=chr(ord(key1[v5])+v3)

v5+=1

print(flag)

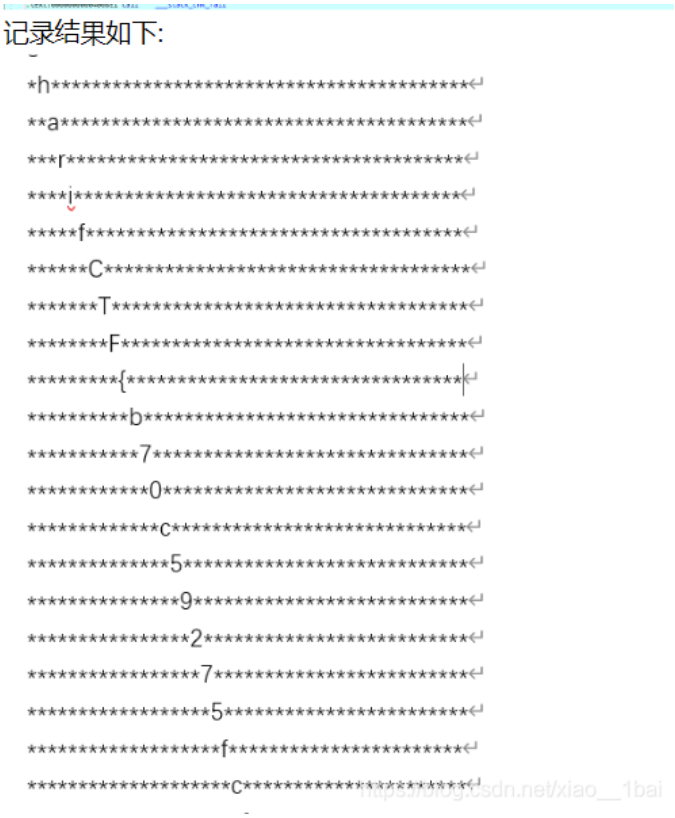
第四种动态截停，在IDA远程调试中截停在remove(filename);最后这里，或者return 0;也行，然后在IDA中写python脚本命令输出&t地址的字符串即可。

```
.data:00000000006010E0 ; char t
.data:00000000006010E0 t db 'S'                        ; DATA XREF: main+65↑w
.data:00000000006010E0                                 ; main+C9↑o ...
.data:00000000006010E1 aHarifctf db 'harifCTF{b70c59275fcfa8aebf2d5911223c6589}',0
.data:000000000060110C align 20h
.data:0000000000601120 public u
.data:0000000000601120 u db '****************************************',0
```

直接双击t看到?已经被替换成flag了，这里是断点在return。

第五个动态截停，在linux中用GDB或IDA远程调试断点断在fprintf(stream, “%s\n”, u); 这里，然后每次记录写入的一个flag字符。

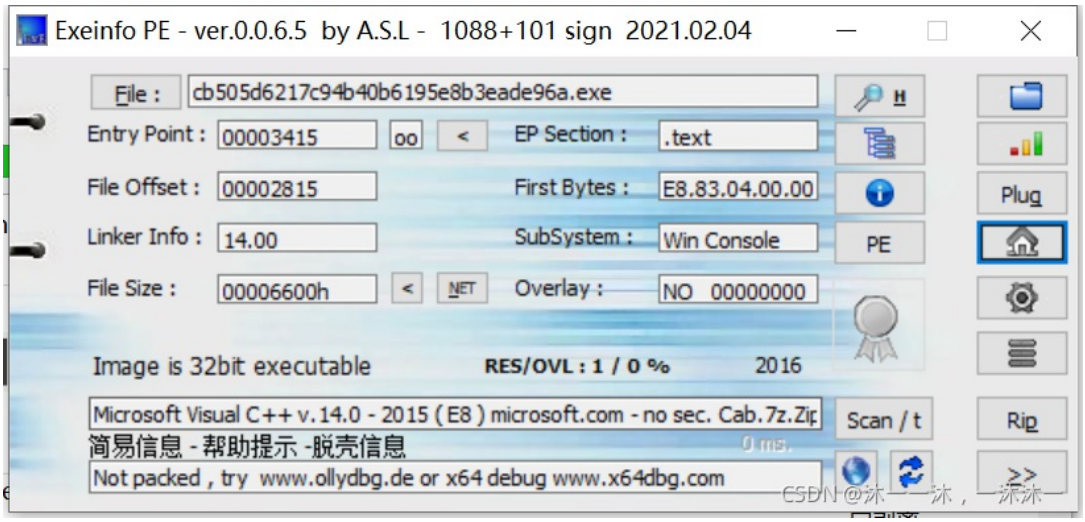
嗯~第五种就不演示了，大致像这个样子吧，不过他这个是整理过的，真实的是不按顺序出现的。



攻防世界key：（不能直接运行、多层交叉引用查看、OD动态调试、寄存器传参、同地址变量、动调验证值猜想、冗余中锁定关键代码、运算符优先级注意、大量判断少赋值的字符串比较算法、不用输入类型）

这道题，嗯~搞了好久，主要是我的IDA是7.5版本的，其他博客上的大多都是7.0或以前的版本，导致很多伪代码的变量名和函数表象对不上，然后这题的直接读取文件也算是长一次见识了。这题逆向逻辑不难，就是找到关键位置我卡了好久好久，一直在梳理各种残缺信息。然后就是要懂得熟悉OD动态调试来梳理程序逻辑和看复杂函数的输出才行。

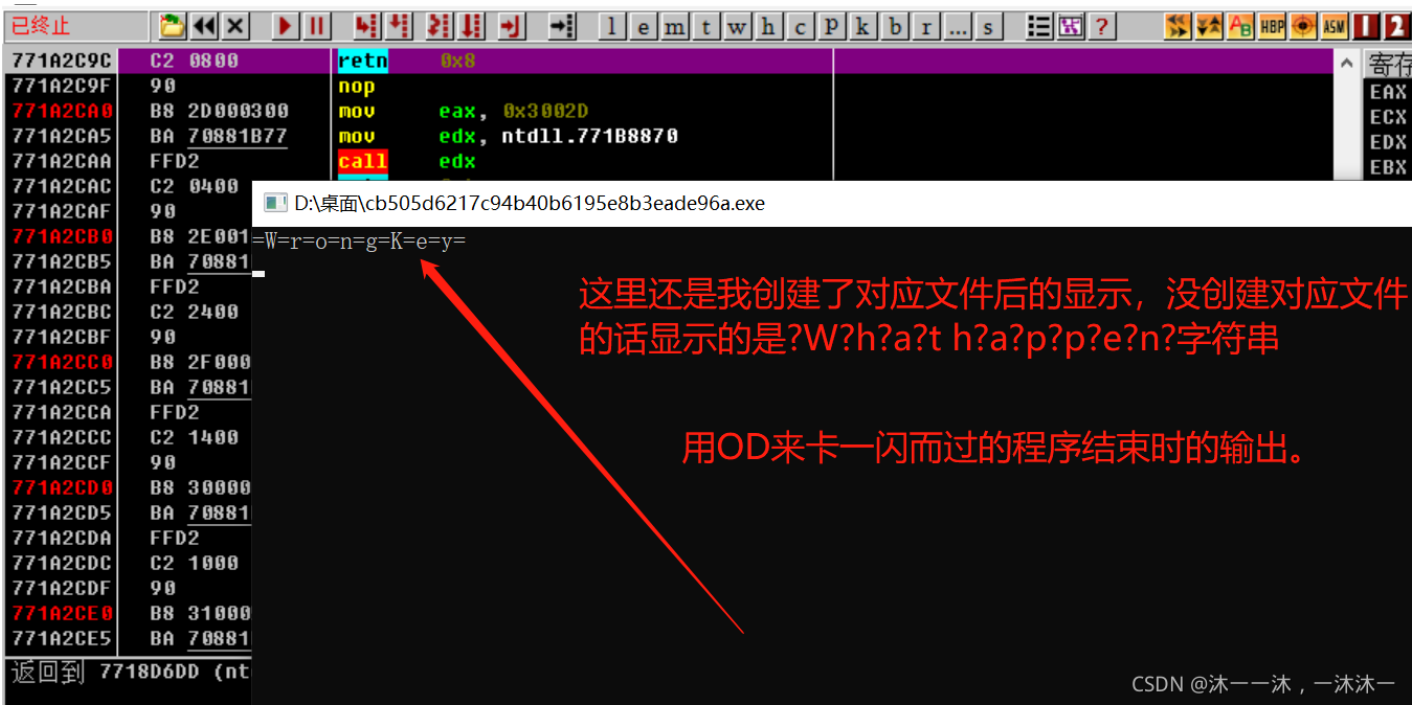
下载附件，照例扔入exeinfope中查看信息，32位无壳：



照例双击运行查看一下主要字符串，结果是一闪而过，查了一下程序会一闪而过的原因，如下所示：

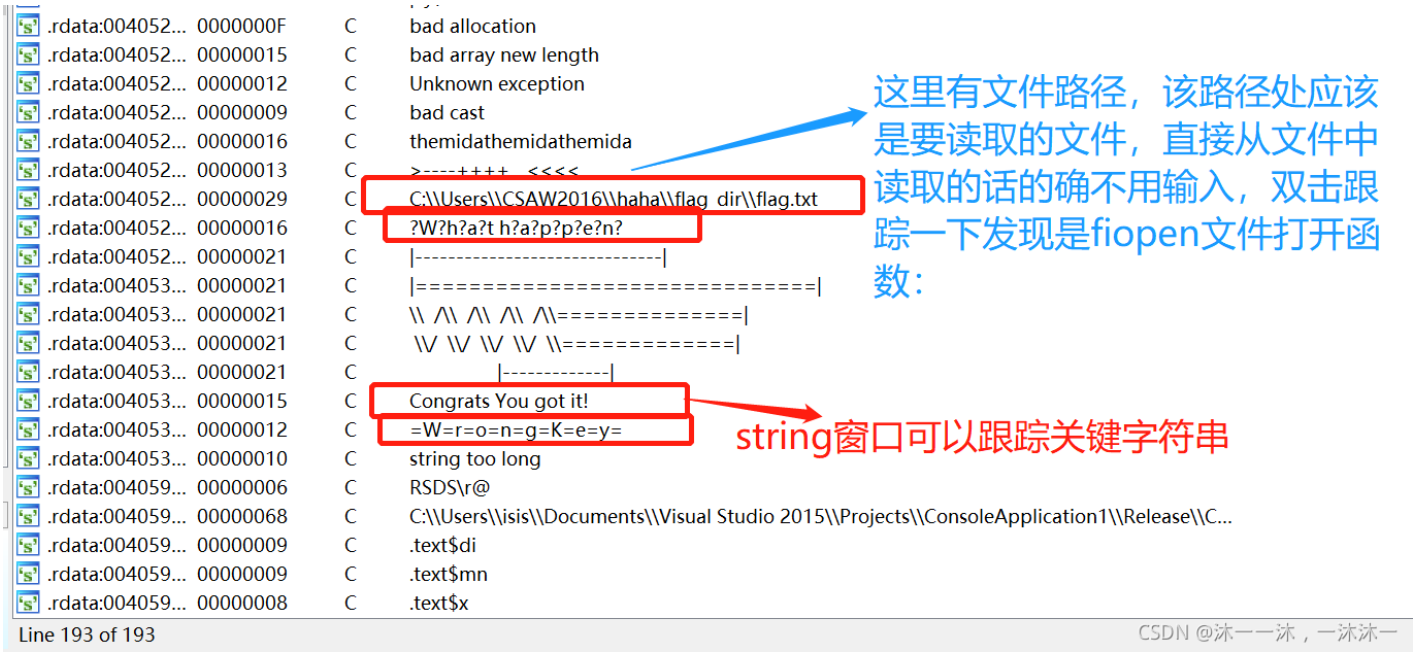
（拿到题目点开就秒退，推测是直接检测某种flag的形式而不是输入flag进行check。）

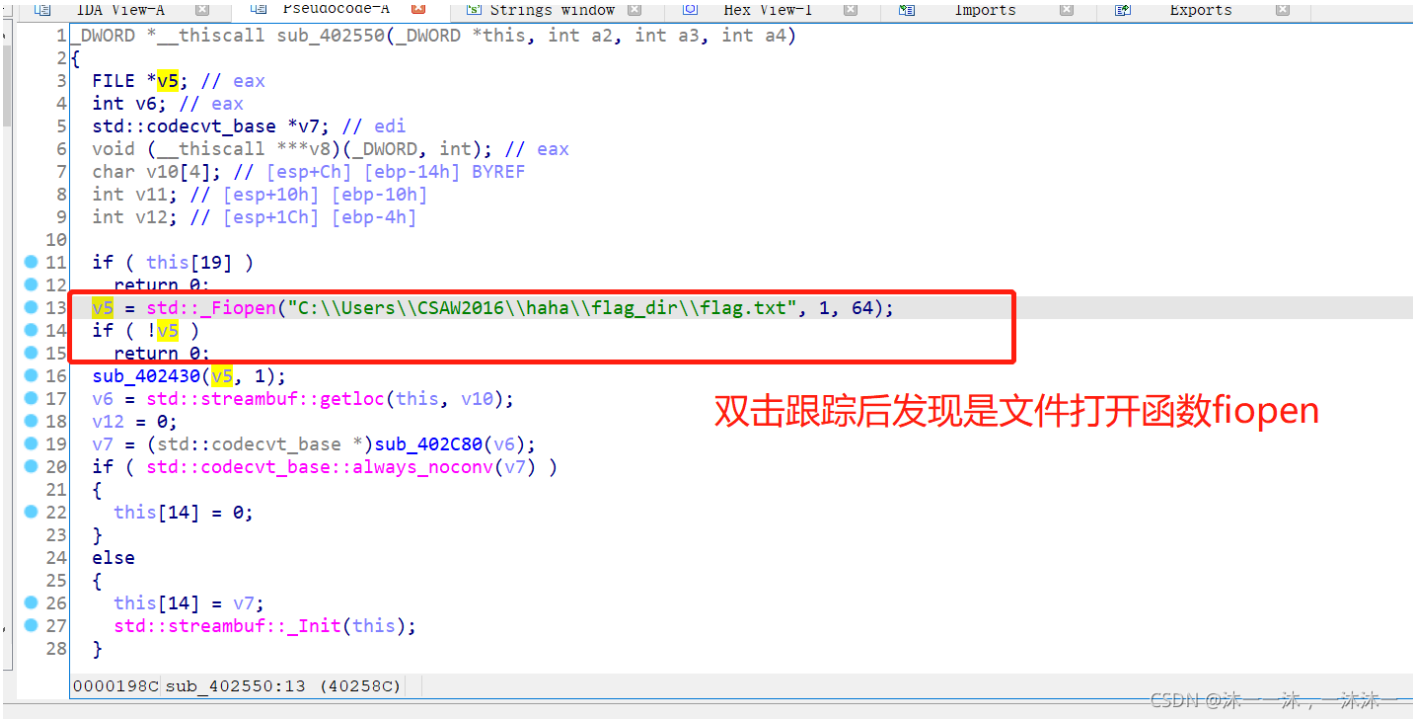
不用输入，那就说得通了。然后用OD运行来卡一下结束时输出了什么字符串。



照例扔入IDA32中查看信息，先看strings窗口吧：

基本关键字符串都在这里了，主要是上图中第一个红框中的路径处应该是要读取的文件，直接从文件中读取的话的确不用输入，双击跟踪一下发现是fiopen文件打开函数：





上图中并没有?W?h?a?t h?a?p?p?e?n?字符串，所以该字符串应该是在打开函数之前调用，所以也双击跟踪一下。再不断地通过菜单view-function call查看函数调用，最终也确定了第一个红框处sub_201620函数处调用了前面fopen函数打开文件：

```
56  v1 = 0;
57  v34[1] = 15;
58  v34[0] = 0;
59  LOBYTE(Block[0]) = 0;
60  LOBYTE(v36) = 2;
61  do
62    sub_4021E0(1u, *((_BYTE *)v29 + v1++) + 9);
63  while ( v1 < 18 );
64  memset(v28, 0, sizeof(v28));
65  sub_401620(v2, v16, v17, v18);
66  LOBYTE(v36) = 3;
67  if ( (*((_BYTE *)&v28[3] + *(_DWORD *)(v28[0] + 4)) & 6) != 0 )
68  {
69    v3 = sub_402A00(sub_402C50);                // ?W?h?a?t h?a?p?p?e?n?
70    std::ostream::operator<<(v3, v19);
71    exit(-1);
72  }
73  sub_402E90();
74  v4 = &v28[4];
75  if ( v28[23] )
76  {
77    if ( !(unsigned __int8)sub_4022F0() )
78      v4 = 0;
79    if ( fclose((FILE *)v28[23]) )
80      v4 = 0;
81  }
```

不断交叉引用跟踪到的主逻辑函数的上半部分

这里是fiopen函数所在处，也就是这里打开文件

第一个关键回显字符串处，应该是打不开文件就退出。

00000636 sub_401100:67 (401236) (Synchronized with IDA View-A)

```
__thiscall std::streambuf::snextc(_DWORD);
__thiscall std::streambuf::sgetc(_DWORD);
```


继续交叉引用跟踪其它关键字符串，可以发现前面的其它关键字符串congratus和wrong等字符串都在下面判断语句这里，所以从这里开始往前走：

```
85  }
86  LOBYTE(v28[22]) = 0;
87  BYTE1(v28[19]) = 0;
88  std::streambuf::_Init(&v28[4]);
89  v28[20] = dword_408590;
90  v28[23] = 0;
91  v28[21] = dword_408594;
92  v28[18] = 0;
93  if ( !v4 )
94    std::ios::setstate(v28 + *(v28[0] + 4), 2, 0);
95  v6 = Block;
96  if ( v34[1] >= 0x10u )
97    v6 = Block[0];
98  if ( !sub_4020C0(v5, v31, v6, v34[0]) )
99  {
100   v7 = sub_402A00(sub_402C50);
101   std::ostream::operator<<(v7, v20);
102   v8 = sub_402A00(sub_402C50);
103   std::ostream::operator<<(v8, v21);
104   v9 = sub_402A00(sub_402C50);
105   std::ostream::operator<<(v9, v22);
106   v10 = sub_402A00(sub_402C50);
107   std::ostream::operator<<(v10, v23);
108   v11 = sub_402A00(sub_402C50);
109   std::ostream::operator<<(v11, v24);
110   v12 = sub_402A00(sub_402C50);
111   std::ostream::operator<<(v12, v25);
112   v13 = sub_402A00(sub_402C50);
113   std::ostream::operator<<(v13, v26);
114   std::ostream::operator<<(std::cout, sub_402C50);
115 }
116 v14 = sub_402A00(sub_402C50);
117 std::ostream::operator<<(v14, v27);
118 sub_401570();
119 std::ios::~ios<char,std::char_traits<char>>(&v28[28]);
120 if ( v34[1] >= 0x10u )
121   sub_402630(Block[0], v34[1] + 1);
122 result = v32;
123 if ( v32 >= 0x10 )
124   result = sub_402630(v30, v32 + 1);
125 return result;
126}
```

不断交叉引用跟踪到的主逻辑函数的下半部分

前面代码比较多和难懂，只能从关键判断的if语句开始反推。

congratus等字符串输出在这里，但是IDA没有直接反编译出来，所以比较难看。

00000703 sub_401100:94 (401303)

(这里积累第三个经验)

判断函数sub_4020C0(v5, v31, v6, v34[0])一开始我跟踪进去没看懂，然后开始分析该函数参数的由来，首先是v5，v5后来通过OD动态调试梳理明白后是文件中的内容：

```
3  int v0; // esi
4  int v1; // esi
5  int v2; // ecx
6  int v3; // eax
7  int *v4; // esi
8  int v5; // ecx
9  void **v6; // eax
10 int v7; // eax
11 int v8; // eax
12 int v9; // eax
13 int v10; // eax
14 int v11; // eax
15 int v12; // eax
16 int v13; // eax
17 int v14; // eax
18 int result; // eax
19 int v16; // [esp-Ch] [ebp-144h]
20 int v17; // [esp-8h] [ebp-140h]
21 int v18; // [esp-4h] [ebp-13Ch]
22 int v19; // [esp-4h] [ebp-13Ch]
23 int v20; // [esp-4h] [ebp-13Ch]
24 int v21; // [esp-4h] [ebp-13Ch]
25 int v22; // [esp-4h] [ebp-13Ch]
```

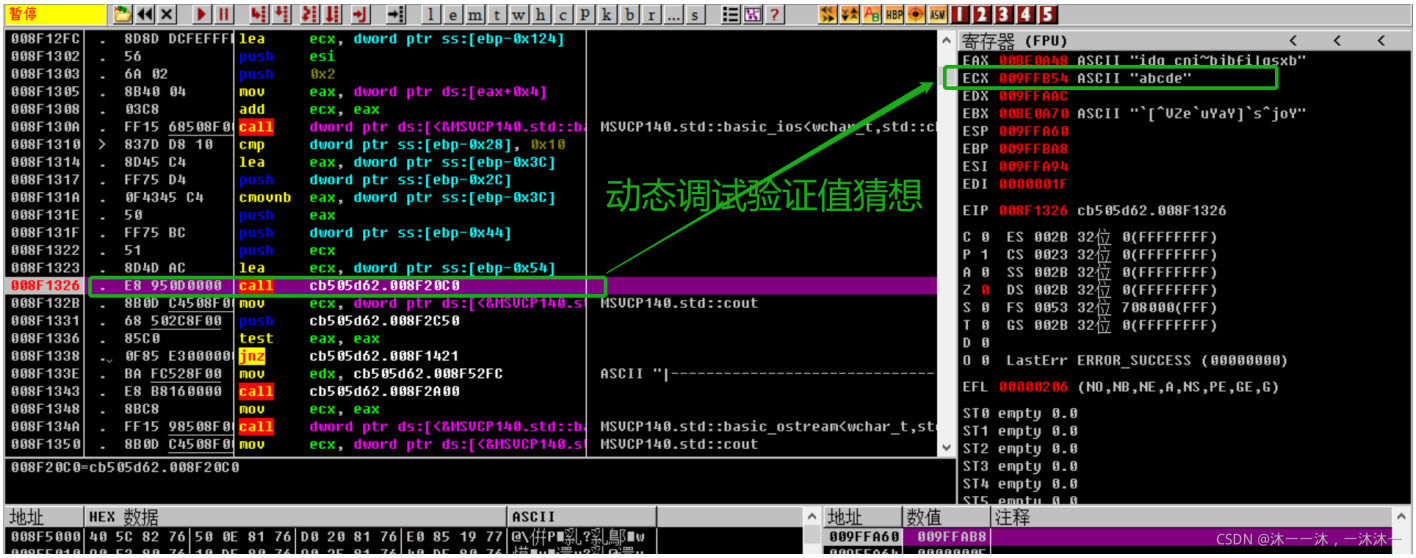
同样的寄存器在这里竟然是不同的变量名，一开始不知道IDA7.5会这样反编译，真的干扰了好久，所以还是要多看看寄存器变量啊！


```
59  LOBYTE(Block[0]) = 0;
60  LOBYTE(v36) = 2;
61  do
62    sub_4021E0(1u, *(v29 + v1++) + 9);
63  while ( v1 < 18 );
64  memset(v28, 0, sizeof(v28));
65  sub_401620(v2, v16, v17, v18);
66  LOBYTE(v36) = 3;
67  if ( (*(&v28[3] + *(v28[0] + 4)) & 6) != 0 )
68  {
69    v3 = sub_402A00(sub_402C50);
70    std::ostream::operator<<(v3, v19);
71    exit(-1);
72  }
73  sub_402E90();
74  v4 = &v28[4];
75  if ( v28[23] )
76  {
77    if ( !sub_4022F0() )
78      v4 = 0;
79    if ( fclose(v28[23]) )
80      v4 = 0;
81  }
82  else
83  {
84    v4 = 0;
85  }
86  LOBYTE(v28[22]) = 0;
87  BYTE1(v28[19]) = 0;
88  std::streambuf::_Init(&v28[4]);
89  v28[20] = dword_408590;
90  v28[23] = 0;
91  v28[21] = dword_408594;
92  v28[18] = 0;
93  if ( !v4 )
94    std::ios::setstate(v28 + *(v28[0] + 4), 2, 0);
95  v6 = Block;
96  if ( v34[1] >= 0x10u )
97    v6 = Block[0];
98  if ( !sub_4020C0(v5, v31, v6, v34[0]) )
99  {
100   v7 = sub_402A00(sub_402C50);
101   std::ostream::operator<<(v7, v20);
0000060A sub_401100:64 (40120A)
```

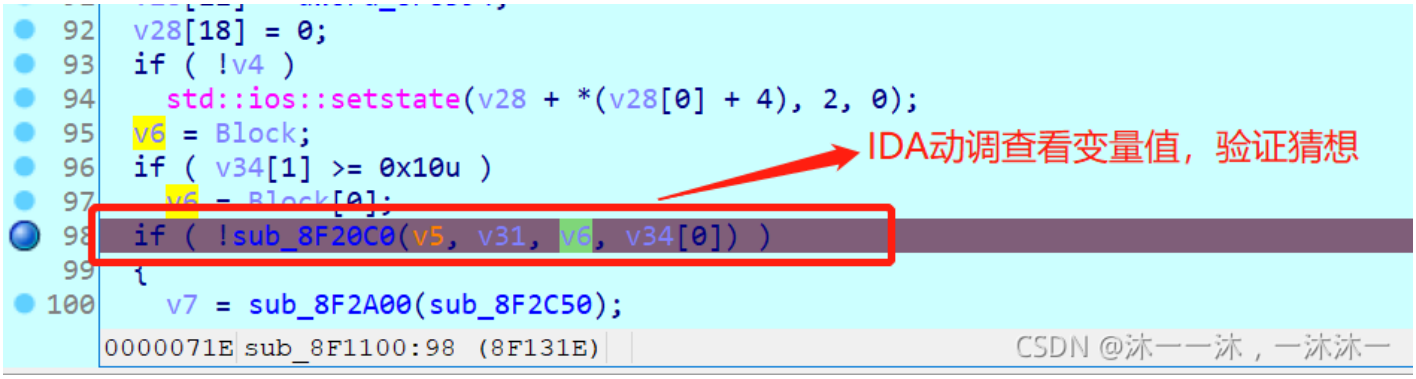
所以前面读文件函数sub_401620读取文件内容后给了v2(ecx)，然后判断函数sub_4020C0也是用v5(ecx)做参数

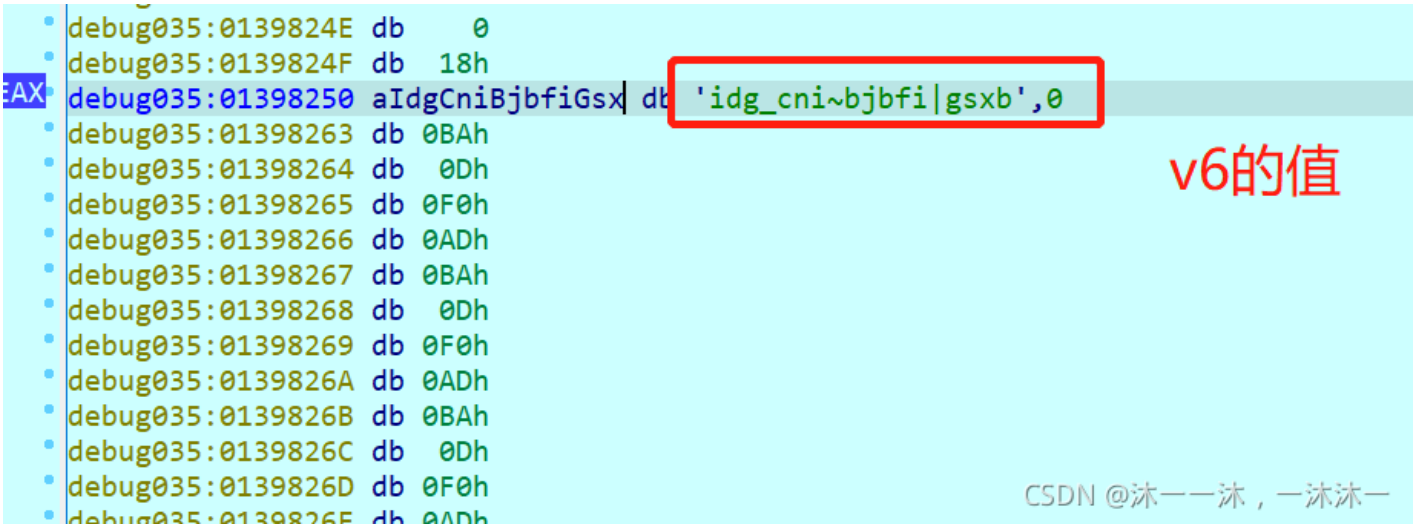
用OD动态调试验证v5是文件内容的猜想，首先修改IDA中的基地址与OD保持一致(Edit—Segments----Rebase programme)，为动态调试提供参考，注意前1000字节是文件头等信息，从2000字节开始才是代码.text.段。

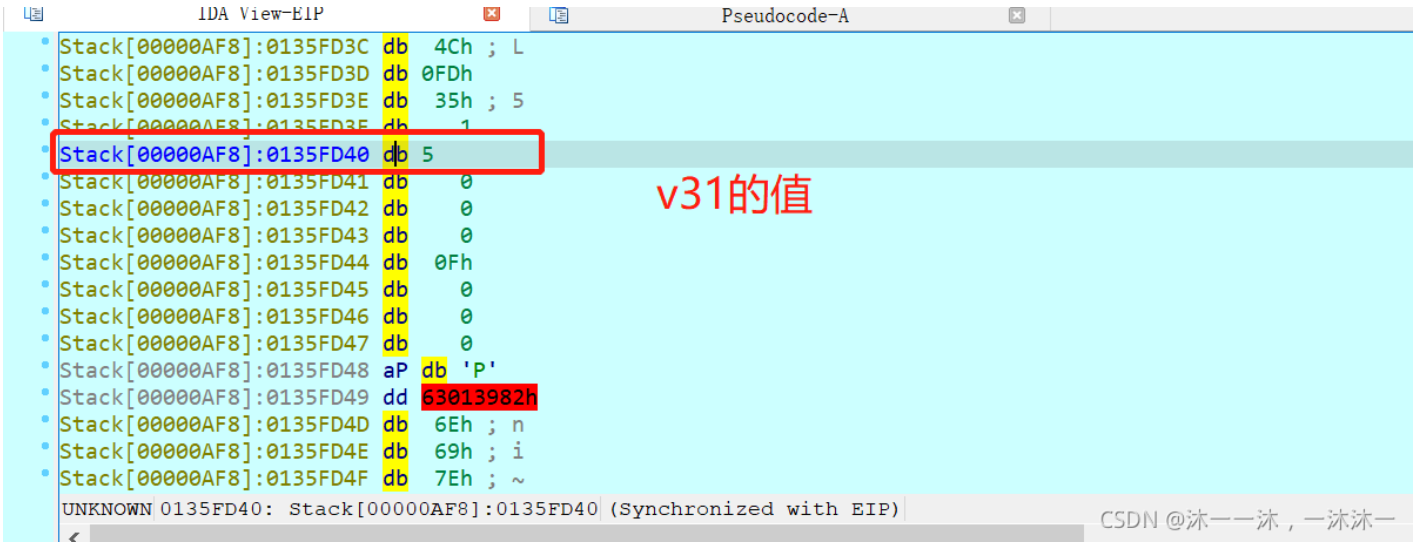
根据目录创建对应文件夹，在flag.txt中写入abcd然后在sub_8F1620地址处下断点，OD运行后看参数v5(ECX)内容：



v5猜想验证，然后就是v31，v6，v34[0]参数，这里用IDA动态调试，因为可以直接双击查看变量的值：







知道了参数的值该是什么后看一下前面的这些参数有没有操作，这里对Block的函数虽然分了两个变量，但是都是一致型的操作，至于为什么分成了两个变量，应该是IDA反编译的时候中间的LOBYTE(Block[0]) = 0;影响了它的反编译吧。

Pseudocode-A

```
46  LOBYTE(v29[0]) = 0;
47  v0 = 0;
48  qmemcpy(Block, "themidathemidathemida", 84);
49  strcpy(v35, ">----++++....<<<<.");
50  do
51  {
52    sub_8F21E0(1u, (*(Block + v0) ^ v35[v0]) + 22);
53    ++v0;
54  }
55  while ( v0 < 18 );
56  v1 = 0;
57  v34[1] = 15;
58  v34[0] = 0;
59  LOBYTE(Block[0]) = 0;
60  LOBYTE(v36) = 2;
61  do
62    sub_8F21E0(1u, *(v29 + v1++) + 9);
63  while ( v1 < 18 );
64  memset(v28, 0, sizeof(v28));
65  sub_8F1620(v2, v16, v17, v18);
66  LOBYTE(v36) = 3;
67  if ( (*(&v28[3] + *(v28[0] + 4)) & 6) != 0 )
68  {
69    v3 = sub_8F2A00(sub_8F2C50);
70    std::ostream::operator<<(v3, v19);
71    exit(-1);
72  }
73  sub_8F2E90();
74  v4 = &v28[4];
75  if ( v28[23] )
76  {
77    if ( !sub_8F22F0() )
78      v4 = 0;
79    if ( fclose(v28[23]) )
80      v4 = 0;
81  }
82  else
83  {
84    v4 = 0;
85  }
86  LOBYTE(v28[22]) = 0;
87  BYTE1(v28[19]) = 0;
88  std::streambuf::_Init(&v28[4]);
89  v28[20] = dword_8F8590;
```

00000650 sub_8F1100:68 (8F1250) (Synchronized with IDA View-A)

继续往前分析

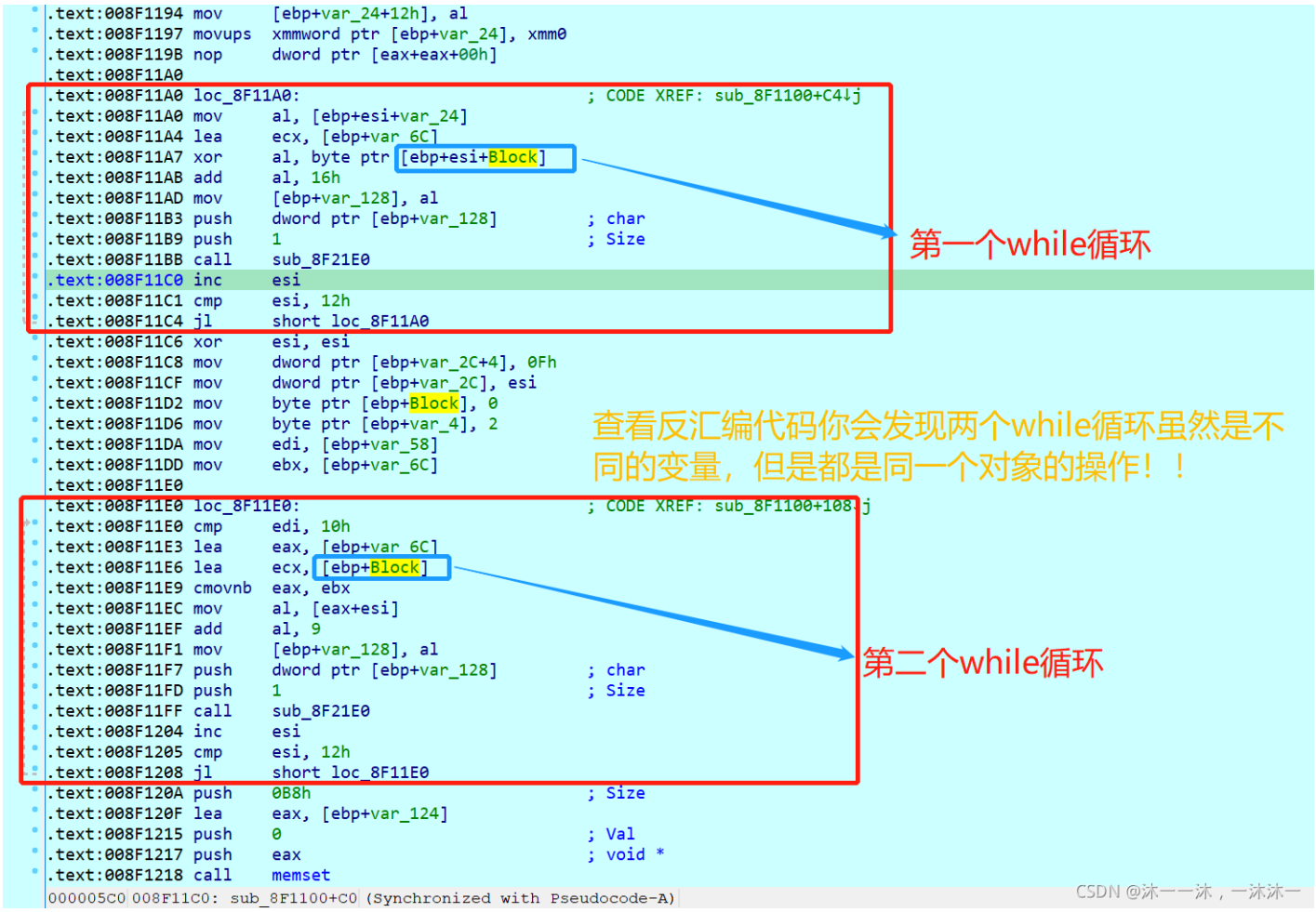
这两个相同的函数相同的循环当它是简单的异或吧，sub_8F21E0函数难看懂，但是通过动态调试获取值可以发现就是简单的异或和相加。

这部分是跟文件读取类有关的操作，已经分析完了。

```
24  int v21; // [esp-4h] [ebp-13Ch]
25  int v22; // [esp-4h] [ebp-13Ch]
26  int v23; // [esp-4h] [ebp-13Ch]
27  int v24; // [esp-4h] [ebp-13Ch]
28  int v25; // [esp-4h] [ebp-13Ch]
29  int v26; // [esp-4h] [ebp-13Ch]
30  int v27; // [esp-4h] [ebp-13Ch]
31  int v28[46]; // [esp+14h] [ebp-124h] BYREF
32  void *v29[6]; // [esp+CCh] [ebp-6Ch]
33  void *v30; // [esp+E4h] [ebp-54h]
34  int v31; // [esp+F4h] [ebp-44h]
35  unsigned int v32; // [esp+F8h] [ebp-40h]
36  void *Block[4]; // [esp+FCh] [ebp-3Ch] BYREF
37  _DWORD v34[2]; // [esp+10Ch] [ebp-2Ch]
38  char v35[20]; // [esp+114h] [ebp-24h] BYREF
39  int v36; // [esp+134h] [ebp-4h]
40
```

上面两个相同循环操作的对象虽然属于不同地址，不同变量，但是反汇编代码中竟然是同一个对象！！！

属于不同变量



所以v6参数的值要找到对应的操作了，剩下的v31和v34[0]就是一个常数，现在知道了参数后我们接着看看判断函数sub_8F20C0(v5, v31, v6, v34[0])：

真的，一开始我看了好久都没看懂这个函数，一堆的判断然后执行一些杂七杂八的操作，后来结合别人的理解发现这个函数的确都是判断，而且对this的赋值修改操作只有下图中的红框而已。

那反过来想一想，没有修改操作，那不就可以当成判断每个字符是否相等的函数吗？事实上也的确是这样，所以判断的就是从文件中读出的v5和前面加密修改的字符串v6：

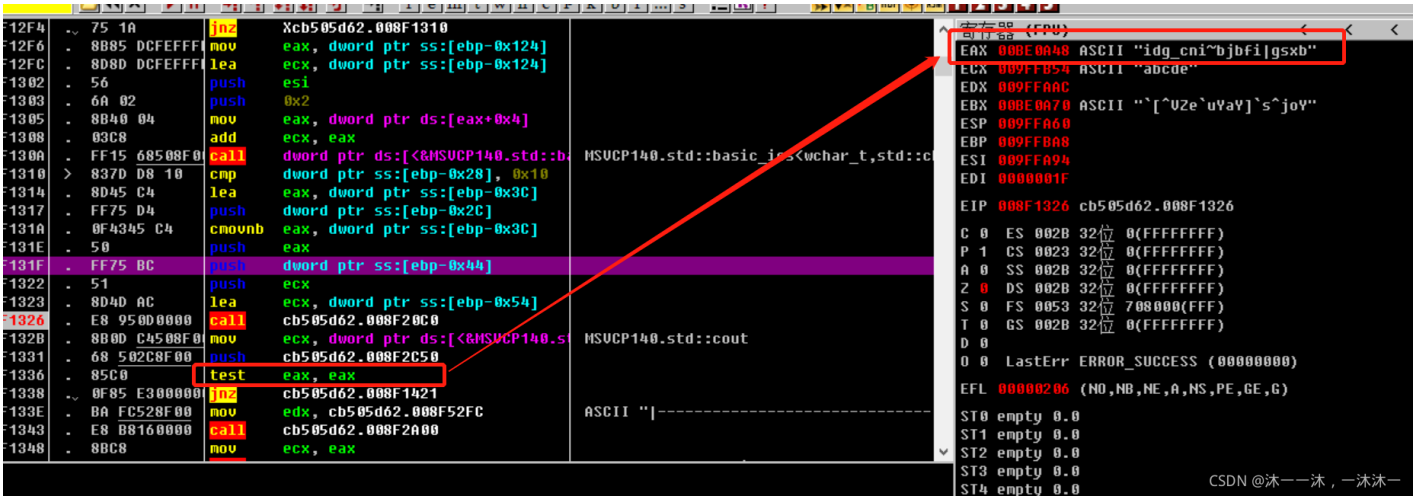
```
10  int result; // eax
11
12  v5 = a3;
13  if ( this[4] < a3 )
14    v5 = this[4];
15  if ( this[5] >= 0x10u )
16    this = *this;
17  v6 = a5;
18  if ( v5 < a5 )
19    v6 = v5;
20  if ( v6 )
21  {
22    v9 = v6 < 4;
23    v8 = v6 - 4;
24    if ( v9 )
25    {
26LABEL_11:
27      if ( v8 == -4 )
28        goto LABEL_20;
29    }
30    else
31    {
32      while ( *this == *a4 )
33      {
34        ++this;
35        a4 += 4;
36        v9 = v8 < 4;
37        v8 -= 4;
38        if ( v9 )
39          goto LABEL_11;
40      }
41    }
42    v9 = *this < *a4;
43    if ( *this != *a4
44      || v8 != -3
45      && ((v10 = *(this + 1), v9 = v10 < *(a4 + 1), v10 != *(a4 + 1))
46       || v8 != -2
47       && ((v11 = *(this + 2), v9 = v11 < *(a4 + 2), v11 != *(a4 + 2))
48        || v8 != -1 && (v12 = *(this + 3), v9 = v12 < *(a4 + 3), v12 != *(a4 + 3)))) )
49    {
50      result = v9 ? -1 : 1;
51      goto LABEL_21;
52    }
53LABEL_20:
000014C0 sub_8F20C0:10 (8F20C0) (Synchronized with IDA View-A)
```

前面一堆判断没有修改的话那只能判断文件字符v5和异或加密后的v6是否相等了。

所以前面分析v6是由异或加密而来的有什么用？其实没什么用，知道结果v6即可，但是可以对整个程序逻辑更清晰。

真正赋值的就这两个

那么最后flag就出来了，一个是动态调试时的v6的值：



一个是自己写脚本运行的v6的值(注意+号和异或符号的优先级)：

key1="themidathemidathemida"

key2=">----++++....<<<<."

flag2=""

flag=""

for i in range(18):

flag2+=chr((ord(key1[i])^ord(key2[i]))+22) #加号+运算符优先级大于异或+，要加括号啊，！！！！刚才直接输出错误

for i in range(18):

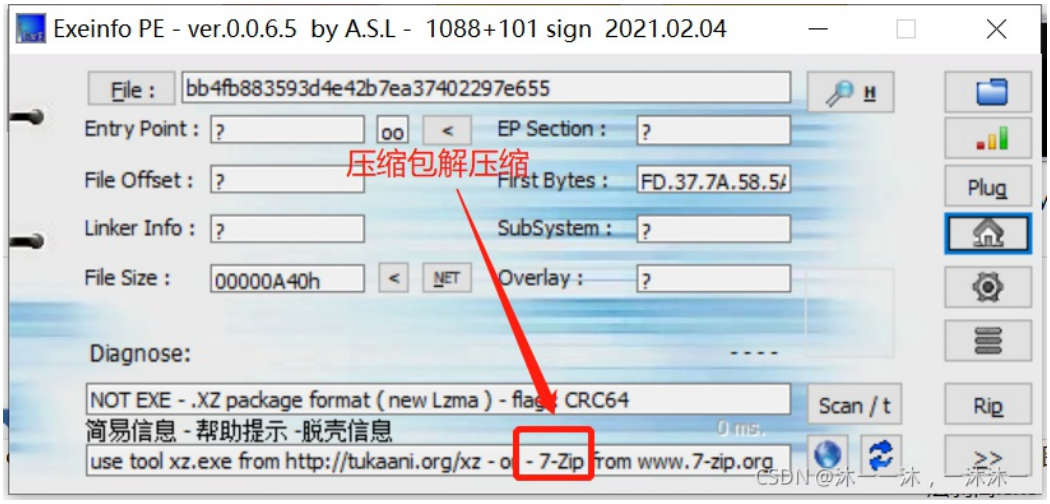
```
flag+=chr(ord(flag2[i])+9)

print(flag)
```

```
└─$ python 1.py
idg_cni~bjbfi|gsxb
```

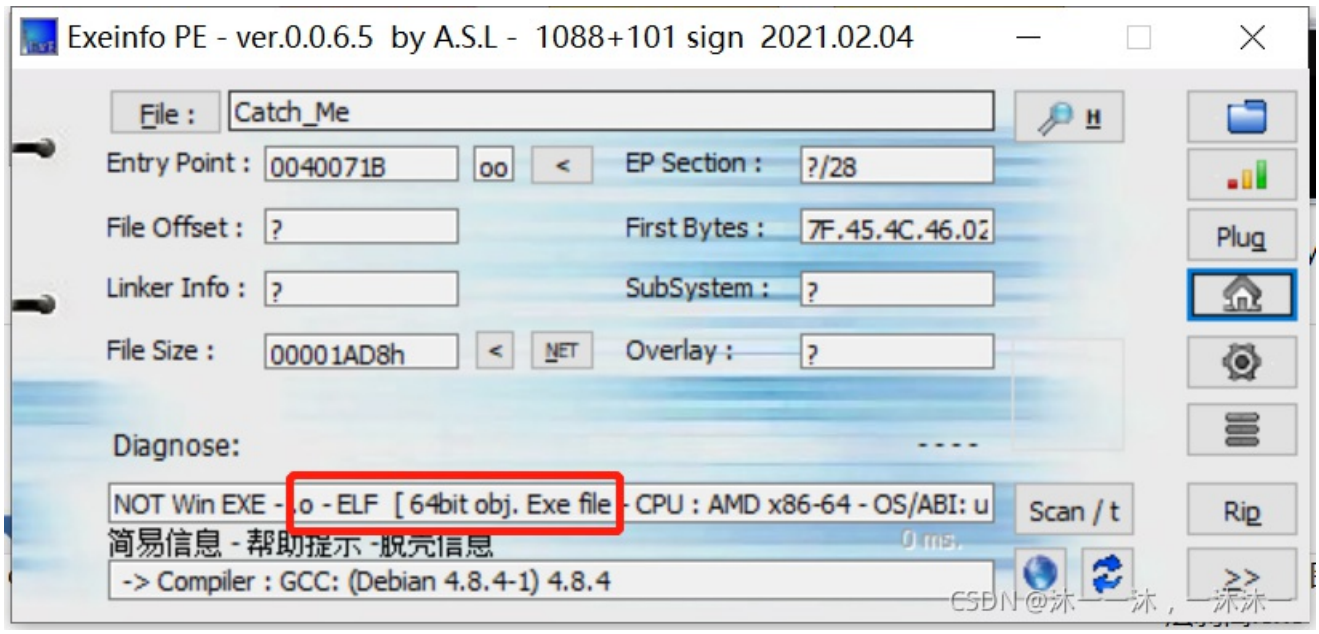
main函数中有与本地环境变量相关的操作类型：

攻防世界catch-me：（不能直接运行、不用输入类型、main函数主逻辑平铺、冗余中锁定关键代码、运算符优先级注意、IDA动态调试、小端转大端存储算法）

照例下载附件，结果两次都提示是用7z打开(解压)，所以要解压两次才能得到真正文件：



最后得到的文件是一个64位的ELF文件：



照例运行一下程序查看主要回显信息：

```
└─$ ./1
Flag: ASIS{bad_bad_bad_bad_bad_bad}
this flag is absolutely fake
```

(这里积累第一个经验)

啥也没输入就直接程序结束了，回顾一下前面做的不用输入的类型，那是检查文件中flag的，而且没有对应文件就闪退的。

那么这题也是不用输入的，这种不用输入的程序都是直接利用程序外部的一些东西来作为条件继续执行的，所以我们这次也是要修改程序外的一些东西，来符合程序的执行。（这题修改的是环境变量）

```
__m128i v10; // xmm1
__m128i v11; // xmm4
__m128i v12; // xmm0
__m128i v13; // xmm0

v3 = ((__int64 (__fastcall *)(_QWORD, char **, char **))sub_400820)((unsigned int)dword_6012B4, a2, a3);
byte_6012A8[0] = HIBYTE(v3);
byte_6012A9 = BYTE2(v3) & 0xFD;
byte_6012AA = BYTE1(v3) & 0xDF;
byte_6012AB = v3 & 0xBF;
if ( getenv("ASIS") && (*(_DWORD *)getenv("CTF") ^ v3) == 0xFEEBFEEB )
  dword_6012AC = *(_DWORD *)getenv("ASIS");
for ( i = 0LL; i != 33; ++i )
  haystack[i] ^= byte_6012A8[i & 7];          // 对haystack的异或赋值操作
v5 = _mm_load_si128((const __m128i *)haystack);
v6 = _mm_unpacklo_epi8(v5, (__m128i)0LL);
v7 = _mm_unpackhi_epi8(v5, (__m128i)0LL);
v8 = _mm_add_epi32(
       _mm_unpackhi_epi16(v7, (__m128i)0LL),
       _mm_add_epi32(
         _mm_add_epi32(_mm_unpackhi_epi16(v6, (__m128i)0LL), _mm_unpacklo_epi16(v6, (__m128i)0LL)),
         _mm_unpacklo_epi16(v7, (__m128i)0LL)));
v9 = _mm_load_si128((const __m128i *)&xmmword_601290);
v10 = _mm_unpackhi_epi8(v9, (__m128i)0LL);
v11 = _mm_unpacklo_epi8(v9, (__m128i)0LL);
v12 = _mm_add_epi32(
        _mm_add_epi32(
          _mm_add_epi32(
            _mm_add_epi32(v8, _mm_unpacklo_epi16(v11, (__m128i)0LL)),
            _mm_unpackhi_epi16(v11, (__m128i)0LL)),
          _mm_unpacklo_epi16(v10, (__m128i)0LL)),
        _mm_unpackhi_epi16(v10, (__m128i)0LL));
v13 = _mm_add_epi32(v12, _mm_srli_si128(v12, 8));
if ( _mm_cvtsi128_si32(_mm_add_epi32(v13, _mm_srli_si128(v13, 4))) != 2388 )
  strcpy(haystack, "bad_bad_bad_bad_bad_bad");
printf("Flag: ASIS{%s}\n", haystack);
if ( strstr(haystack, "bad_") )
  puts("this flag is absolutely fake ");
return 0LL;
}
```

这里有对haystack的操作。

注意这里的判断语句在动态调试中没有执行，所以应该执行了这条语句后的haystack才是真的flag。

这部分都是SEE的一些位移动函数，但是好像并没有改变过haystack

满足条件才会输出flag，可以看出haystack就是flag。

(这里积累第三个经验)

所以在前面v3是直接生成的，这里能获取程序外部的只有getenv函数，那么关键地方就是这里if (getenv("ASIS") && (*(_DWORD *)getenv("CTF") ^ v3) == 0xFEEBFEEB)

v3动态调试发现值是0xB11924E1，那么我们要设置满足条件的ASIS和CTF环境变量才能执行if语句，才有可能获取flag。

这里其实很多其它博客都忽略了一点就是getenv("ASIS")应该是要爆破出来的，首先附上C语言运算符的优先级，从图中可以看出只要前面(*(_DWORD *)getenv("CTF") ^ v3) == 0xFEEBFEEB满足，那么后面getenv("ASIS")只要不是0就可以。

(注意这里&&不要和&位运算搞混了，&&是逻辑与，&是位与。)

运算符优先级：

类别	运算符	结合性
后缀	() [] -> . ++ --	从左到右
一元	+ - ! ~ ++ -- (type)*(指针) & sizeof	从右到左
乘除	* / %	从左到右
加减	+ -	从左到右
移位	<< >>	从左到右
关系	< <= > >=	从左到右
相等	== !=	从左到右
位与 AND	&	从左到右
位异或 XOR	^	从左到右
位或 OR	|	从左到右
逻辑与 AND	&&	从左到右
逻辑或 OR	||	从左到右
条件	?:	从右到左
赋值	= += -= *= /= %=>>= <<= &= ^= |=	从右到左
逗号	,	从左到右

先执行括号内的(*(_DWORD *)getenv("CTF") ^ v3)

再执行(*(_DWORD *)getenv("CTF") ^ v3) == 0xFEEBFEEB)

最后才执行i getenv("ASIS") &&

getenv(“CTF”)=0xFEEBFEEB^v3，可以算出等于0x4ff2da0a，那么getenv("ASIS")是多少呢？只能猜测它和getenv("CTF")一样都是0x4ff2da0a试一试了。

（这里积累第四个经验）

所以我们导入linux的环境变量，这里也补充一些基本知识：（注意：export只能在当前终端中有效，终端关闭就没用了）

(通过printf可以导入十六进制整数类型，值得积累)

export ASIS="$(printf "\x0a\xda\xf2\x4f")" #注意参数是从低位到高位的，因为是内存操作

export CTF="$(printf "\x0a\xda\xf2\x4f")" #$()执行括号内计算公式，属于命令替换

Linux中的$符号的三种常见用法:

用法一：显示脚本参数（$0、$?、$*、$@、$#、$$、$!）（本质上属于变量替换）
参考上面shell的参数传递。

用法二：${ }获取变量与环境变量的值
如：path=2，则echo $path 或者echo${path}显示的都是path的值。(也就是说花括号{}加不加都一样)

在linux及unix的sh中，以$开头的字符串表示的是sh中定义的变量，这些变量可以是系统自动增加的，也可以是用户自己定义的$PATH表示的是系统的命令搜索路径，和windows的%path%是一样的$HOME则表示是用户的主目录，也就是用户登录后工作目录。

用法三：shell中$(( ))、$( )、`` 命令执行和命令替换:
$(( ))属于执行计算公式，等价于$[ ]，$( )和`属于命令替换，${ }属于变量替换

（1）$( )与``(反引号)：返回括号中命令的结果
先完成引号里的命令行，然后将其结果替换出来，再重组成新的命令行

应用示例:
$ echo today is $(date "+%Y-%m-%d")，显示：today is 2014-07-01

然后直接运行，果不其然getenv("ASIS")等于getenv("CTF")，得出真实的flag了：

（这里积累第5个经验）

另一种方法就是我们既然知道了getenv("ASIS")的值，我们也可以直接自己生成flag啊。

首先第一个要注意的是v3的0xB11924E1本来应该是小端顺序逆序存储在内存中的，可是这里红框中HIBYTE、BYTE2、BYTE1、BYTE直接把v3按大端顺序存储了，这点要特别注意，而且还经过了位与&处理。

然后就是黄框中在v3数组后拼接了小端的getenv("ASIS")，最后我们直接用Export data直接导出haystack原始的32位值进行处理即可。

附上脚本，注意源代码中是[i & 7]差点看错成[i % 7]了：

key1=0xFEEBFEEB

key2=0xB11924E1

key3=key1^key2 #0x4ff2da0a

print(hex(key3&(key3^key2)))

list1=[0xb1,0x19&0xfd,0x24&0xdf,0xe1&0xbf,0x0a,0xda,0xf2,0x4f]#v3按大端顺序被截取了

flag=[0x87, 0x29, 0x34, 0xC5, 0x55, 0xB0, 0xC2, 0x2D, 0xEE, 0x60, 0x34, 0xD4, 0x55, 0xEE, 0x80, 0x7C,0xEE, 0x2F, 0x37, 0x96, 0x3D, 0xEB,0x9C, 0x79, 0xEE, 0x2C, 0x33, 0x95, 0x78, 0xED, 0xC1, 0x2B]

for i in range(32):

flag[i]^=list1[i&7]

print('ASIS{'+''.join(map(chr,flag))+'}')

main函数主逻辑分析（C++）

main函数中嵌入大量冗余代码，拆分代码混淆：

攻防世界dmd-50：（函数积累、地址小端存放与正向、md5加密/解密算法、出人意料的flag）

64位ELF文件，无壳，照例扔入IDA64位中查看伪代码，有main函数看main函数：

看到一堆变量，一堆系统函数，我知道系统函数通常不是考点，可这系统函数多到我都差点找不到主要代码：

```
v43 = __readfsqword(0x28u);
std::operator<<<std::char_traits<char>>(&std::cout, "Enter the valid key!\n", envp);
std::operator>><char,std::char_traits<char>>(&edata, v42);
std::allocator<char>::allocator(&v38);
std::string::string(v39, v42, &v38);
md5((MD5 *)v40, (const std::string *)v39);
v41 = std::string::c_str((std::string *)v40);
std::string::~string((std::string *)v40);
std::string::~string((std::string *)v39);
std::allocator<char>::~allocator(&v38);
if ( *(_WORD *)v41 == 0x3837
  && *(_BYTE *)(v41 + 2) == 0x30
  && *(_BYTE *)(v41 + 3) == 0x34
  && *(_BYTE *)(v41 + 4) == 0x33
  && *(_BYTE *)(v41 + 5) == 0x38
  && *(_BYTE *)(v41 + 6) == 0x64
  && *(_BYTE *)(v41 + 7) == 0x35
  && *(_BYTE *)(v41 + 8) == 0x62
  && *(_BYTE *)(v41 + 9) == 0x36
  && *(_BYTE *)(v41 + 10) == 0x65
  && *(_BYTE *)(v41 + 11) == 0x32
```


突然看见字符串Enter the valid key!\n 猜测题型是与用户输入相关的判断型flag。代码分析：

v43 = __readfsqword(0x28u); //一个内存段偏移，无用

std::operator<<<std::char_traits<char>>(&std::cout, "Enter the valid key!\n", envp);

std::operator>><char,std::char_traits<char>>(&edata, v42); //这里应该是输入

std::allocator<char>::allocator(&v38); //内存空间分配，空间分配就是用于输入或复制的

std::string::string(v39, v42, &v38); // v42 输入复制给 v39。这里借鉴了别人的博客，这里犯下第一个错误，因为系统函数太多了，我也就只注意到了下面的md5，没去查这个函数用法，结果这个函数是赋值函数，也是关键

md5((MD5 *)v40, (const std::string *)v39); //一个md5加密函数， 把 v39 进行 MD5 后保存在 v40，我记得C语言没有md5函数的，这里可能是调用了外部的库函数

v41 = std::string::c_str((std::string *)v40);

std::string::~string((std::string *)v40);

std::string::~string((std::string *)v39);

std::allocator<char>::~allocator(&v38);

继续分析判断语句：

if (*(_WORD *)v41 == 0x3837 //这里犯下第二个错误，这里本来是14390的整数的，这里类型是16位WORD，后面都是8位的BYTE，也就是这里应该分出一个*(_BYTE *)v41和*(_BYTE *)(v41 + 1)的，可是我不会分，想起数在内存中是小端的十六进制数，就改为了十六进制，那么前面(_BYTE *)v41就是0x37，后面*(_BYTE *)(v41 + 1)就是0x38了。

&& *(_BYTE *)(v41 + 2) == 0x30

&& *(_BYTE *)(v41 + 3) == 0x34

&& *(_BYTE *)(v41 + 4) == 0x33

&& *(_BYTE *)(v41 + 5) == 0x38

```
&& *(_BYTE *)(v41 + 6) == 0x64
&& *(_BYTE *)(v41 + 7) == 0x35
&& *(_BYTE *)(v41 + 8) == 0x62
&& *(_BYTE *)(v41 + 9) == 0x36
&& *(_BYTE *)(v41 + 10) == 0x65
&& *(_BYTE *)(v41 + 11) == 0x32
&& *(_BYTE *)(v41 + 12) == 0x39
&& *(_BYTE *)(v41 + 13) == 0x64
&& *(_BYTE *)(v41 + 14) == 0x62
&& *(_BYTE *)(v41 + 15) == 0x30
&& *(_BYTE *)(v41 + 16) == 0x38
&& *(_BYTE *)(v41 + 17) == 0x39
&& *(_BYTE *)(v41 + 18) == 0x38
&& *(_BYTE *)(v41 + 19) == 0x62
&& *(_BYTE *)(v41 + 20) == 0x63
&& *(_BYTE *)(v41 + 21) == 0x34
&& *(_BYTE *)(v41 + 22) == 0x66
&& *(_BYTE *)(v41 + 23) == 0x30
&& *(_BYTE *)(v41 + 24) == 0x32
&& *(_BYTE *)(v41 + 25) == 0x32
&& *(_BYTE *)(v41 + 26) == 0x35
&& *(_BYTE *)(v41 + 27) == 0x39
&& *(_BYTE *)(v41 + 28) == 0x33
&& *(_BYTE *)(v41 + 29) == 0x35
&& *(_BYTE *)(v41 + 30) == 0x63
&& *(_BYTE *)(v41 + 31) == 0x30 )
```

再后面判断后的结果按照以前经验转成ASCII字符，发现都是成功失败类的字符串，那这里的系统函数作用应该只是简单的赋值和输出吧，就不用深究了：

```
v3 = std::operator<<<std::char_traits<char>>(&std::cout, 'T');
v4 = std::operator<<<std::char_traits<char>>(v3, 'h');
v5 = std::operator<<<std::char_traits<char>>(v4, 'e');
v6 = std::operator<<<std::char_traits<char>>(v5, ' ');
v7 = std::operator<<<std::char_traits<char>>(v6, 'k');
v8 = std::operator<<<std::char_traits<char>>(v7, 'e');
v9 = std::operator<<<std::char_traits<char>>(v8, 'y');
v10 = std::operator<<<std::char_traits<char>>(v9, ' ');
v11 = std::operator<<<std::char_traits<char>>(v10, 'i');
v12 = std::operator<<<std::char_traits<char>>(v11, 's');
v13 = std::operator<<<std::char_traits<char>>(v12, ' ');
v14 = std::operator<<<std::char_traits<char>>(v13, 'v');
v15 = std::operator<<<std::char_traits<char>>(v14, 'a');
v16 = std::operator<<<std::char_traits<char>>(v15, 'l');
v17 = std::operator<<<std::char_traits<char>>(v16, 'i');
v18 = std::operator<<<std::char_traits<char>>(v17, 'd');
v19 = std::operator<<<std::char_traits<char>>(v18, ' ');
v20 = std::operator<<<std::char_traits<char>>(v19, ':');
v21 = std::operator<<<std::char_traits<char>>(v20, ')');
std::ostream::operator<<(v21, &std::endl<char,std::char_traits<char>>);
result = 0;
```


所以思路清晰了，对用户输入进行md5加密后一位一位比较，那么我们用md5后的值在在线工具中反向解密即可：

key1=
[0x37,0x38,0x30,0x34,0x33,0x38,0x64,0x35,0x62,0x36,0x65,0x32,0x39,0x64,0x62,0x30,0x38,0x39,0x38,0x62,0x

md=""

for i in key1:

md+=chr(i)

print(len(md))

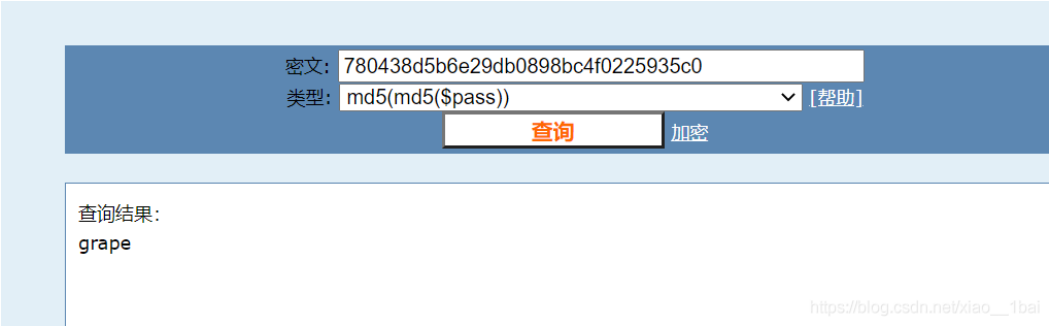
print(md)

md5结果：

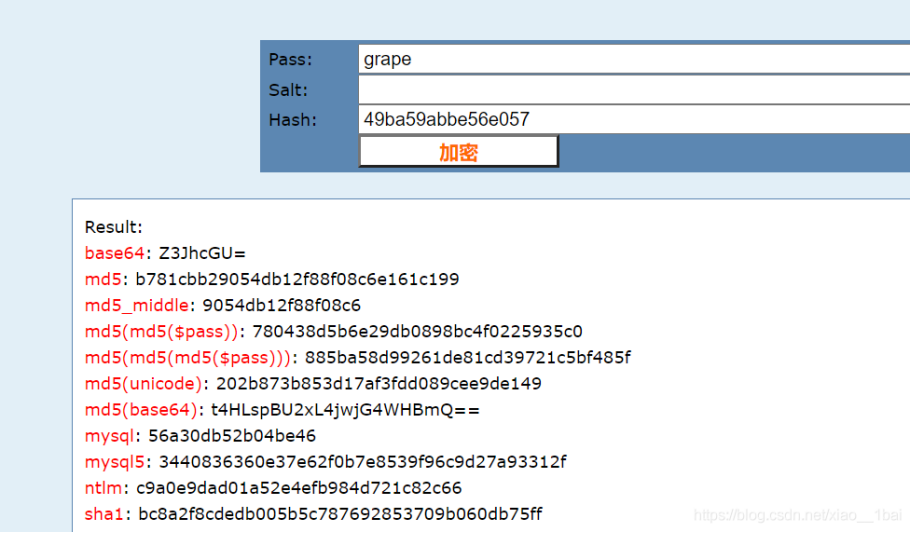
```
└─$ python 1.py
32
780438d5b6e29db0898bc4f0225935c0
```

在线解密网址：

https://www.cmd5.com/



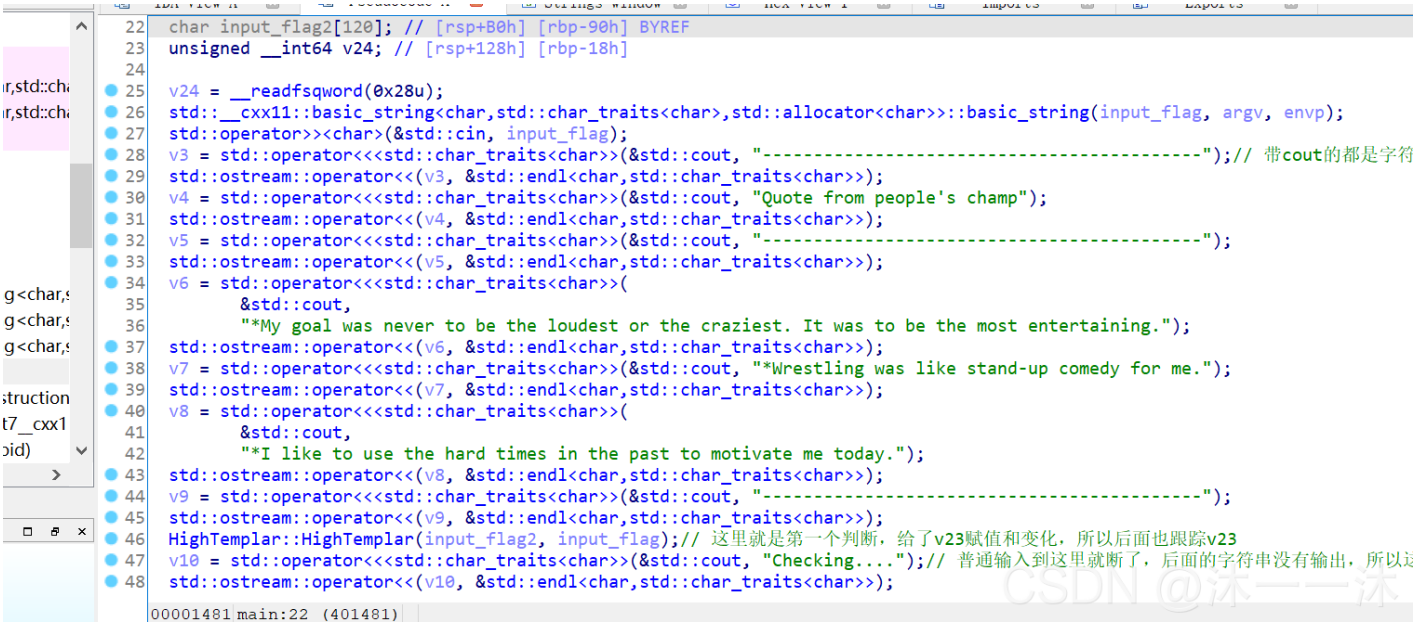
额，我们不需要两次解密，一次就可以了，所以对grape再加密一次：



攻防世界crazy：（函数名称暗示、地址赋值算法积累、非预期行为、出人意料的flag）

64位ELF文件，扔入对应IDA中查看信息，有main函数看main函数：

可以看到，一堆眼花缭乱的系统函数，有些则是用类名调用的普通C++函数。



```
22  char input_flag2[120]; // [rsp+B0h] [rbp-90h] BYREF
23  unsigned __int64 v24; // [rsp+128h] [rbp-18h]
24
25  v24 = __readfsqword(0x28u);
26  std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::basic_string(input_flag, argv, envp);
27  std::operator>><char>(&std::cin, input_flag);
28  v3 = std::operator<<<std::char_traits<char>>(&std::cout, "----------------------------------------");// 带cout的都是字符
29  std::ostream::operator<<(v3, &std::endl<char,std::char_traits<char>>);
30  v4 = std::operator<<<std::char_traits<char>>(&std::cout, "Quote from people's champ");
31  std::ostream::operator<<(v4, &std::endl<char,std::char_traits<char>>);
32  v5 = std::operator<<<std::char_traits<char>>(&std::cout, "----------------------------------------");
33  std::ostream::operator<<(v5, &std::endl<char,std::char_traits<char>>);
34  v6 = std::operator<<<std::char_traits<char>>(
35        &std::cout,
36        "*My goal was never to be the loudest or the craziest. It was to be the most entertaining.");
37  std::ostream::operator<<(v6, &std::endl<char,std::char_traits<char>>);
38  v7 = std::operator<<<std::char_traits<char>>(&std::cout, "*Wrestling was like stand-up comedy for me.");
39  std::ostream::operator<<(v7, &std::endl<char,std::char_traits<char>>);
40  v8 = std::operator<<<std::char_traits<char>>(
41        &std::cout,
42        "*I like to use the hard times in the past to motivate me today.");
43  std::ostream::operator<<(v8, &std::endl<char,std::char_traits<char>>);
44  v9 = std::operator<<<std::char_traits<char>>(&std::cout, "----------------------------------------");
45  std::ostream::operator<<(v9, &std::endl<char,std::char_traits<char>>);
46  HighTemplar::HighTemplar(input_flag2, input_flag);// 这里就是第一个判断，给了v23赋值和变化，所以后面也跟踪v23
47  v10 = std::operator<<<std::char_traits<char>>(&std::cout, "Checking....");// 普通输入到这里就断了，后面的字符串没有输出，所以
48  std::ostream::operator<<(v10, &std::endl<char,std::char_traits<char>>);
00001481 main:22 (401481)
```

这里积累第一个经验：(别人博客的一句话)

代码看着很乱，有很多很长的命令，解决办法：依据英文意思去猜测。

找关键变量的方法：从后往前找，看flag和输入关系。复杂代码本质应该是简洁的，这样才叫出题。

但是从后往前找与flag的有关变量还是很麻烦，所以我们用运行程序方法不断查看显示信息，锁定关键位置。(调试的话不知道断点下在那里可能要遍历很长时间)

第一次乱输入：

```
└─$ ./crazy
123
------------------------------------------
Quote from people's champ
------------------------------------------
*My goal was never to be the loudest or the craziest. It was to be the most entertaining.
*Wrestling was like stand-up comedy for me.
*I like to use the hard times in the past to motivate me today.
------------------------------------------
Checking....
Too short or too long
```


可以看到运行到checking...后显示too short or too long处，还有就是用户输入在字符串输出之前。返回IDA查看代码：

这里有个cin，也是主函数中在其它字符串之前的，cin是c++的输入函数。

```
v24 = __readfsqword(0x28u);
std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::basic_string(input_flag, argv, envp);
std::operator>><char>(&std::cin, input_flag);
```

从这里积累第二个经验：该程序的长字符传命令中从后往前找到的熟悉的C++函数就是我们要关注的命令，比如这里的cin函数，像下面截图中字符串前面也有cout函数：

```
= __readfsqword(0x28u);
::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::basic_string(input_flag, argv, envp);
::operator>><char>(&std::cin, input_flag);
= std::operator<<<std::char_traits<char>>(&std::cout, "----------------------------------------");// 带cout的都是字符输出函数
::ostream::operator<<(v3, &std::endl<char,std::char_traits<char>>);
= std::operator<<<std::char_traits<char>>(&std::cout, "Quote from people's champ");
::ostream::operator<<(v4, &std::endl<char,std::char_traits<char>>);
= std::operator<<<std::char_traits<char>>(&std::cout, "----------------------------------------");
::ostream::operator<<(v5, &std::endl<char,std::char_traits<char>>);
= std::operator<<<std::char_traits<char>>(
    &std::cout,
    "*My goal was never to be the loudest or the craziest. It was to be the most entertaining.");
::ostream::operator<<(v6, &std::endl<char,std::char_traits<char>>);
= std::operator<<<std::char_traits<char>>(&std::cout, "*Wrestling was like stand-up comedy for me.");
::ostream::operator<<(v7, &std::endl<char,std::char_traits<char>>);
= std::operator<<<std::char_traits<char>>(
    &std::cout,
    "*I like to use the hard times in the past to motivate me today.");
::ostream::operator<<(v8, &std::endl<char,std::char_traits<char>>);
= std::operator<<<std::char_traits<char>>(&std::cout, "----------------------------------------");
::ostream::operator<<(v9, &std::endl<char,std::char_traits<char>>);
hTemplar::HighTemplar(input_flag2, input_flag);// 这里就是第一个判断，给了v23赋值和变化，所以后面也跟踪v23
= std::operator<<<std::char_traits<char>>(&std::cout, "Checking....");// 普通输入到这里就断了，后面的字符串没有输出，所以这里有判断
::ostream::operator<<(v10, &std::endl<char,std::char_traits<char>>);
```

看上图中最后的check...字符串之后的函数：

```
v10 = std::operator<<<std::char_traits<char>>(&std::cout, "Checking....");// 普通输入到这里就断了，后
std::ostream::operator<<(v10, &std::endl<char,std::char_traits<char>>);
std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::basic_string(v19, in
func1((__int64)v20, (__int64)v19);            // 这些函数里面好像没有v23，应该是混淆的，当然不排除通过
func2((__int64)v21, (__int64)v20);
func3((__int64)v21, 0);
std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::~basic_string(v21);
std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::~basic_string(v20);
std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::~basic_string(v19);
HighTemplar::calculate((HighTemplar *)input_flag2);// V23在这里用到了
if ( !(unsigned int)HighTemplar::getSerial((HighTemplar *)input_flag2) )// 对修改后的字符串进行检查操
{
  v11 = std::operator<<<std::char_traits<char>>(&std::cout, "////////////////////////////////");
  std::ostream::operator<<(v11, &std::endl<char,std::char_traits<char>>);
  v12 = std::operator<<<std::char_traits<char>>(&std::cout, "Do not be angry. Happy Hacking :)");
```

下图中的自定义函数

可以看到 checking 到 if 判断语句之间还是有很多函数的，可以用前面的依据英文意思猜测的方法去看函数，也可以在strings窗口查找too short or too long处的函数位置：(我选择后者)

```
bool __fastcall HighTemplar::calculate(HighTemplar *this)
{
  __int64 v1; // rax
  _BYTE *v2; // rbx
  bool result; // al
  _BYTE *v4; // rbx
  int i; // [rsp+18h] [rbp-18h]
  int j; // [rsp+1Ch] [rbp-14h]

  if ( std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::length((char *)this + 16) != 32 )
  {
    v1 = std::operator<<<std::char_traits<char>>(&std::cout, "Too short or too long");
    std::ostream::operator<<(v1, std::endl<char,std::char_traits<char>>);
    exit(-1);
  }
  for ( i = 0;
        i <= (unsigned __int64)std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::length((char *)this + 1
        ++i )
  {
    v2 = (_BYTE *)std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::operator[](
                    (char *)this + 16,          // 这是一个两步操作，v2取的是对应字符的地址，*v2是指在原v2地址上修改。把修改input_flag字符
                    i);
    *v2 = (*(_BYTE *)std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::operator[](
                       (char *)this + 16,
                       i) ^ 80)
        + 23;
  }
```


这里跟踪到HighTemplar::calculate函数，根据英文名是计算函数，但是我看不懂这里的this+16，凭借意思我推测是我们输入flag的地址，把我们输入的flag经过两个简单的循环异或加密后输出：

bool __fastcall HighTemplar::calculate(HighTemplar *this) //接受用户输入作为参数

{

__int64 v1; // rax

_BYTE *v2; // rbx

bool result; // al

_BYTE *v4; // rbx

int i; // [rsp+18h] [rbp-18h]

int j; // [rsp+1Ch] [rbp-14h]

if (std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::length((char *)this + 16) != 32) //判断长度

{

v1 = std::operator<<<std::char_traits<char>>(&std::cout, "Too short or too long"); //输出判断结果语句

std::ostream::operator<<(v1, std::endl<char,std::char_traits<char>>);

exit(-1);

}

for (i = 0;

i <= (unsigned __int64)std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::length((char *)this + 16);

++i) //在长度范围内的for循环操作

{

v2 = (_BYTE *)std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::operator[](

(char *)this + 16, // 这里积累第三个经验：这是一个两步操作，v2取的是对应字符的地址，*v2是指在原v2地址上对值进行修改，也就是对Input_flag进行修改。把修改input_flag字符分成了两步做，让不熟悉的我载了跟头。（这里算是地址赋值算法积累）

i);

v2 = ((_BYTE *)std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::operator[](

(char *)this + 16,

i) ^ 80) //对input_flag每个字符异或80

+ 23;

}

for (j = 0; ; ++j)

{

result = j <= (unsigned __int64)std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::length((char *)this + 16); //又是在input_flag的长度范围内的for语句

if (!result)

break;

v4 = (_BYTE *)std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::operator[](

(char *)this + 16,

j); //简单分配空间，取地址

v4 = ((_BYTE *)std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::operator[](

(char *)this + 16,

j) ^ 19) 对input_flag每个字符异或19

+ 11;

}

return result;

}

然后由于HighTemplar::calculate函数的下一个就是if判断函数，所以我们只能跟踪if判断函数内的HighTemplar::getSerial了：

```
__int64 v5; // rax
unsigned int i; // [rsp+1Ch] [rbp-14h]

for ( i = 0;
      (int)i < (unsigned __int64)std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::l
      ++i )
{
  v1 = *(_BYTE *)std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::operator[](
                   (char *)this + 80,          // this+80地址处是327a6c4304ad5938eaf0efb6cc3e53dc的字符串
                   (int)i);
  if ( v1 != *(_BYTE *)std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::operator[](
                         (char *)this + 16,    // this+80处如果不等于this+16处就不通过
                         (int)i) )
  {
    v4 = std::operator<<<std::char_traits<char>>(&std::cout, "You did not pass ");
    v5 = std::ostream::operator<<(v4, i);
    std::ostream::operator<<(v5, &std::endl<char,std::char_traits<char>>);
    *((_DWORD *)this + 3) = 1;
    return *((unsigned int *)this + 3);
  }
  v2 = std::operator<<<std::char_traits<char>>(&std::cout, "Pass ");
  v3 = std::ostream::operator<<(v2, i);
  std::ostream::operator<<(v3, &std::endl<char,std::char_traits<char>>);
}
```


前面this+16我推测是我们输入flag的地方，但是这个this+80存了什么东西我是真不知道了。双击跟踪堆栈也是未赋值的状态。

这里积累第四个经验，逆向中不符合预期的运算结果基本都是中间做了其它操作，如之前遇到的HOOK，这里很多函数我还没跟踪，那说明的确会有未发现的操作。

回到一开始cin函数的地方，表黄输入变量，看哪里还引用过该变量：

```
unsigned __int64 v24; // [rsp+128h] [rbp-18h]

v24 = __readfsqword(0x28u);
std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::basic_string(input_flag, argv, envp);
std::operator>><char>(&std::cin, input_flag);
v3 = std::operator<<<std::char_traits<char>>(&std::cout, "----------------------------------------");// 带cout的都是字符输
std::ostream::operator<<(v3, &std::endl<char,std::char_traits<char>>);
v4 = std::operator<<<std::char_traits<char>>(&std::cout, "Quote from people's champ");
std::ostream::operator<<(v4, &std::endl<char,std::char_traits<char>>);
v5 = std::operator<<<std::char_traits<char>>(&std::cout, "----------------------------------------");
std::ostream::operator<<(v5, &std::endl<char,std::char_traits<char>>);
v6 = std::operator<<<std::char_traits<char>>(
       &std::cout,
       "*My goal was never to be the loudest or the craziest. It was to be the most entertaining.");
std::ostream::operator<<(v6, &std::endl<char,std::char_traits<char>>);
v7 = std::operator<<<std::char_traits<char>>(&std::cout, "*Wrestling was like stand-up comedy for me.");
std::ostream::operator<<(v7, &std::endl<char,std::char_traits<char>>);
v8 = std::operator<<<std::char_traits<char>>(
       &std::cout,
       "*I like to use the hard times in the past to motivate me today.");
std::ostream::operator<<(v8, &std::endl<char,std::char_traits<char>>);
v9 = std::operator<<<std::char_traits<char>>(&std::cout, "----------------------------------------");
std::ostream::operator<<(v9, &std::endl<char,std::char_traits<char>>);
HighTemplar::HighTemplar(input_flag2, input_flag);// 这里就是第一个判断，给了v23赋值和变化，所以后面也跟踪v23
v10 = std::operator<<<std::char_traits<char>>(&std::cout, "Checking....");// 普通输入到这里就断了，后面的字符串没有输出，所以这!
std::ostream::operator<<(v10, &std::endl<char,std::char_traits<char>>);
std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::basic_string(v19, input_flag);// 把输入字符串给
00001543 main:38 (401543)
```

可以看到在checking前面还引用了一下，而该函数我们并没有分析，双击跟踪分析：

这里我们可以看到把输入的flag分别给了this+16和this+48地址处，在this+80地址处给了327a6c4304ad5938ea

f0efb6cc3e53dc这个字符串，那么前面对this+80的疑惑就解释得通了。

unsigned __int64 __fastcall HighTemplar::HighTemplar(DarkTemplar *a1, __int64 input_flag)

{

char v3; // [rsp+17h] [rbp-19h] BYREF

unsigned __int64 v4; // [rsp+18h] [rbp-18h]

v4 = __readfsqword(0x28u);

DarkTemplar::DarkTemplar(a1);

*(_QWORD *)a1 = &off_401EA0;

*((_DWORD *)a1 + 3) = 0;

std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::basic_string(

(char *)a1 + 16, // C++函数，basic_string（字符串类模板），不是复制就是比较，这里是复制输入字符串给a1+16开始的地址的数组中

input_flag);

std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::basic_string(

(char *)a1 + 48, // C++函数，basic_string（字符串类模板），不是复制就是比较，复制输入字符串给a1+48开始的地址的数组中，与前面隔了32个字符

input_flag);

std::allocator<char>::allocator(&v3);

std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::basic_string(

(char *)a1 + 80,

"327a6c4304ad5938eaf0efb6cc3e53dc", // C++函数，basic_string（字符串类模板），不是复制就是比较，复制输入字符串给a1+80开始的地址的数组中，与前面还是隔了32个字符，这个v3不清楚

&v3);

std::allocator<char>::~allocator(&v3);

return __readfsqword(0x28u) ^ v4; //内存操作，暂时不用管

}

现在直接写脚本逆向逻辑即可：

key1="327a6c4304ad5938eaf0efb6cc3e53dc"

flag1=""

flag=""

print(len(key1))

for i in range(len(key1)):

flag1+=chr((ord(key1[i])-11)^19)

for i in range(len(flag1)):

flag+=chr((ord(flag1[i])-23)^80)

print(flag)

结果：(这里积累第5个经验：现在的flag真的是越来越古灵精怪了，还有花括号，我一开始都以为我写错脚本了，现在看来，什么类型的flag都可以！一次不行就修改再交几次)

```
└─$ python 1.py
32
tMx~qdstOs~crvtwb~aOba}qddtbrtcd
```

最后，下面这三个函数有什么用呢，我判断它是没什么用的，因为参数没有传入我输入的flag，跟踪里面也没有我输入的flag地址，除非是偏移地址间接引用我输入的flag，不过那样的话题就很难了！

```
std::ostream::operator<<(v9, &std::endl<char,std::char_traits<char>>);
HighTemplar::HighTemplar(input_flag2, input_flag);// 这里就是第一个判断，给了v23赋值和变化，所以后面也跟踪
v10 = std::operator<<<std::char_traits<char>>(&std::cout, "Checking....");// 普通输入到这里就断了，后面的
std::ostream::operator<<(v10, &std::endl<char,std::char_traits<char>>);
std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::basic_string(v19, input_
func1((__int64)v20, (__int64)v19);                    // 这些函数里面好像没有v23，应该是混淆的，当然不排除通过地址
func2((__int64)v21, (__int64)v20);
func3((__int64)v21, 0);
std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::~basic_string(v21);
std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::~basic_string(v20);
std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::~basic_string(v19);
HighTemplar::calculate((HighTemplar *)input_flag2);// V23在这里用到了
```

main函数逻辑平铺：

攻防世界re2-cpp-is-awesome：（字符ASCII码做索引、函数逻辑封装、argv[]外部调用输入参数符合条件、align错误反汇编、模板复制操作）

64位ELF文件，无壳，运行一下查看主要显示字符串：

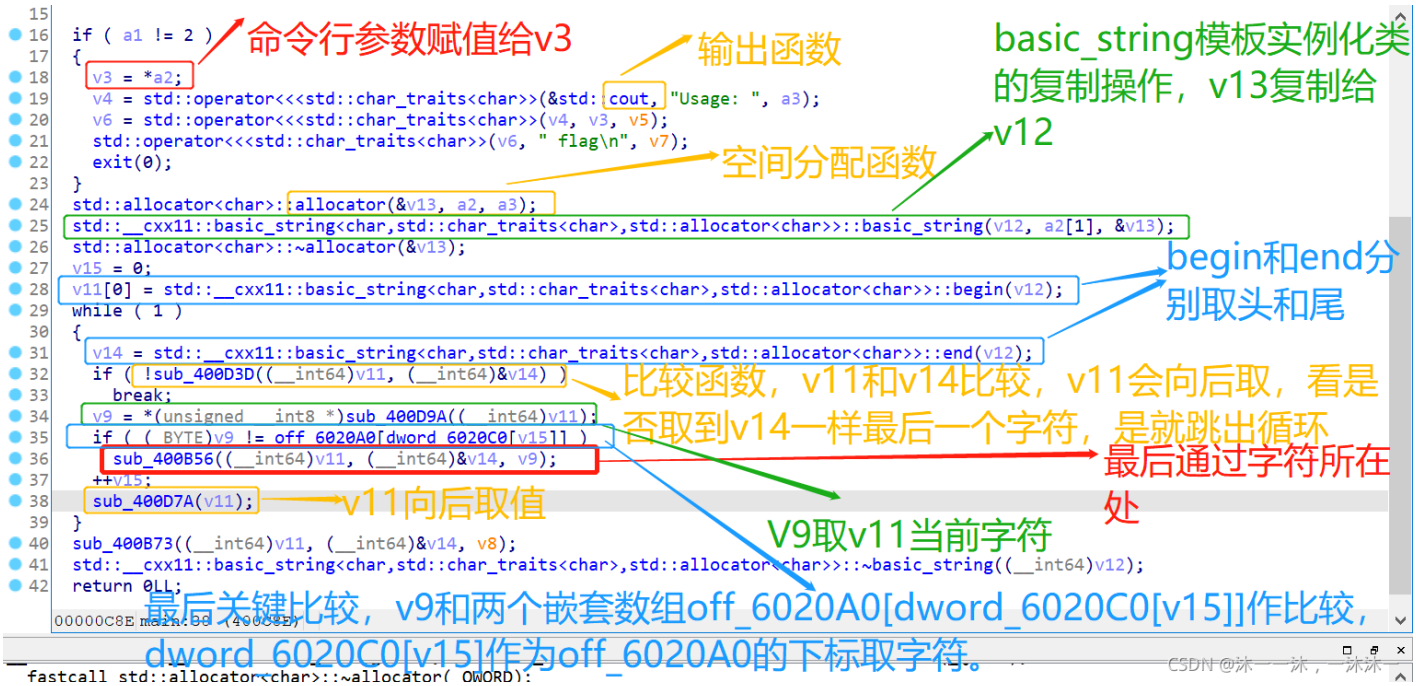
```
└─$ ./1 666
Better luck next time
```

照例扔入IDA中查看伪代码，有main函数看main函数，C++面向对象的代码有点乱，也有点杂，在string中追踪Better luck next time字符串跟踪出在下图的第一个红框处：

```
15
16  if ( a1 != 2 )
17  {
18    v3 = *a2;
19    v4 = std::operator<<<std::char_traits<char>>(&std::cout, "Usage: ", a3);
20    v6 = std::operator<<<std::char_traits<char>>(v4, v3, v5);
21    std::operator<<<std::char_traits<char>>(v6, " flag\n", v7);
22    exit(0);
23  }
24  std::allocator<char>::allocator(&v13, a2, a3);
25  std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::basic_string(v12, a2[1], &v13);
26  std::allocator<char>::~allocator(&v13);
27  v15 = 0;
28  v11[0] = std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::begin(v12);
29  while ( 1 )
30  {
31    v14 = std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::end(v12);
32    if ( !sub_400D3D((__int64)v11, (__int64)&v14) )
33      break;
34    v9 = *(unsigned __int8 *)sub_400D9A((__int64)v11);
35    if ( ( BYTE)v9 != off_6020A0[dword_6020C0[v15]] )
36      sub_400B56((__int64)v11, (__int64)&v14, v9);
37    ++v15;
38    sub_400D7A(v11);
39  }
40  sub_400B73((__int64)v11, (__int64)&v14, v8);
41  std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::~basic_string((__int64)v12);
42  return 0LL;
00000C8E main:38 (400C8E)
fastcall std::allocator<char>::~allocator( QWORD);
```

```
IDA V1…  Pseudoco…  Stack of …  Pseudoco…  Pseudoco…  Strings wi…  He
1 void __fastcall __noreturn sub_400B56(__int64 a1, __int64 a2, __int64 a3)
2 {
3   std::operator<<<std::char_traits<char>>(&std::cout, "Better luck next time\n", a3);
4   exit(0);
5 }
```

C++代码比较陌生，但是根据前面做题中积累的经验中在长类名静态调用的最后一个是要使用的函数名，比如std::__cxx11::basic_string<char,std::char_traits,std::allocator>::begin调用的就是begin这个函数。




```
bool __fastcall sub_400D3D(__int64 a1, __int64 a2)
{
  __int64 v2; // rbx

  v2 = *(_QWORD *)sub_400DAC(a1);
  return v2 != *(_QWORD *)sub_400DAC(a2);
}
```

```
_QWORD *__fastcall sub_400D7A(_QWORD *a1)
{
  ++*a1;
  return a1;
}
```

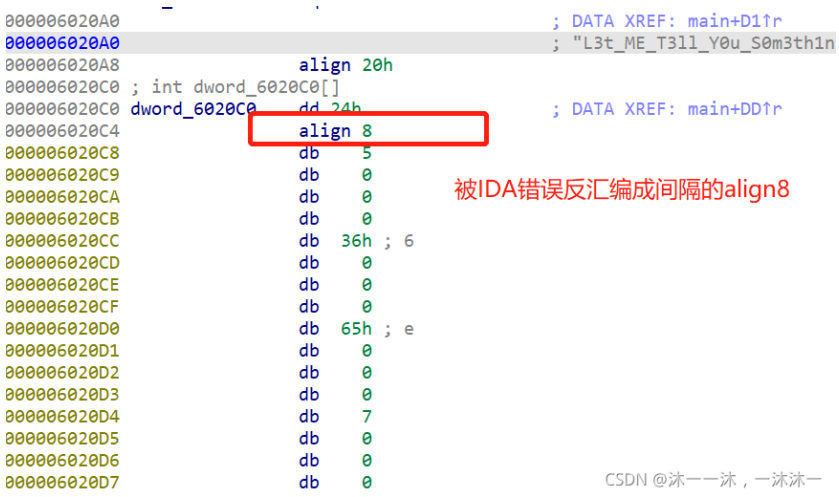
先定位off_6020A0数组：

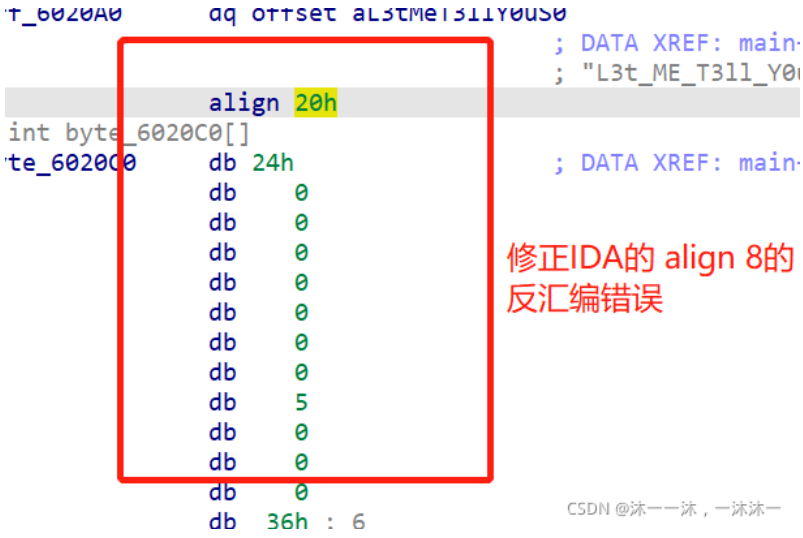
```
0000000000400E55                db    0
0000000000400E56                db    0
0000000000400E57                db    0
0000000000400E58 aL3tMeT3llY0uS0 db 'L3t_ME_T3ll_Y0u_S0m3th1ng_1mp0rtant_A_{FL4G}_W0nt_b3_3X4ctly_th4t'
0000000000400E58                                        ; DATA XREF: .data:off_6020A0↓o
0000000000400E58                db ' 345y_t0_c4ptur3_H0wev3r_1T_w1ll_b3_C00l_1F_Y0u_g0t_1t',0
0000000000400ED0 aBetterLuckNext db 'Better luck next time',0Ah,0
0000000000400ED0                                        ; DATA XREF: sub_400B56+4↑o
```

然后打印dword_6020C0数组：

(这里积累第三个经验)

这里有个align 8，align num是让后面的字节都对齐num，也就是这里都对齐8才对，中间补7个0。可是这里下一个数和上一个数明明间隔4而已！后来查了很多资料才发现是IDA自动把多个0判断成对齐操作了，这里align 8是因为前面dd 24h中本来是db 24 0 0 0 然后后面一个双字是dd 0 也就是db 0 0 0 0，IDA把这连着的7个0当成了间隔，那上一个数和下一个数间隔就是8了，所以IDA生成了align 8。我们只要鼠标右键undefine或把上面的dd 24改一下数据大小即可重定义align 8，重新生成数据了。(前面的align 20h 也是同样的道理)

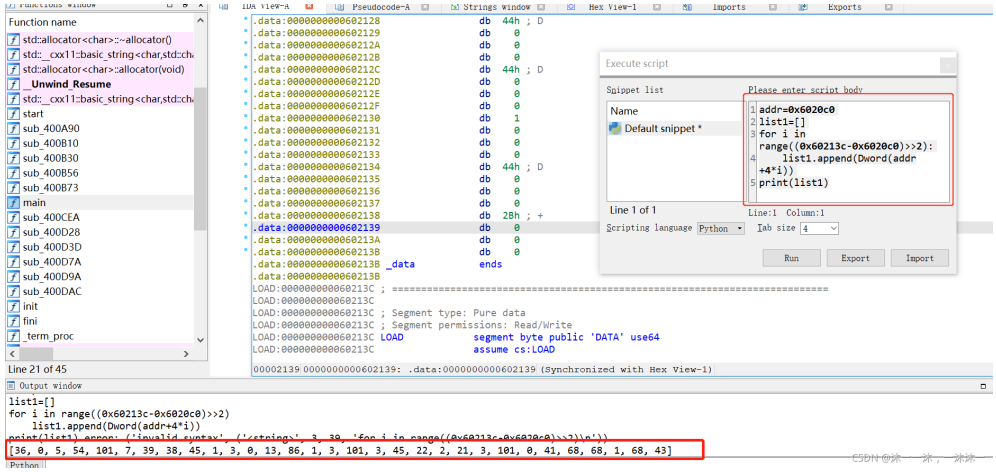




然后开始嵌入python脚本dump下数组内容：

(这里积累第四个经验)

IDA库函数Dword是以当前地址向后四个作为Dword数据，所以我们地址要addr+4*i来保持4的间隔。(0x60213c-0x6020c0)>>2是因为地址之间以4为一个单位，>>2在前面博客中总结过就是除4且保留整数部分。



逆向逻辑脚本：

key1=[36, 0, 5, 54, 101, 7, 39, 38, 45, 1, 3, 0, 13, 86, 1, 3, 101, 3, 45, 22, 2, 21, 3, 101, 0, 41, 68, 68, 1, 68, 43]

key2="L3t_ME_T3ll_Y0u_S0m3th1ng_1mp0rtant_A_{FL4G}_W0nt_b3_3X4ctly_th4t_345y_t0_c4ptur3_H0wev

flag=""

for i in key1:

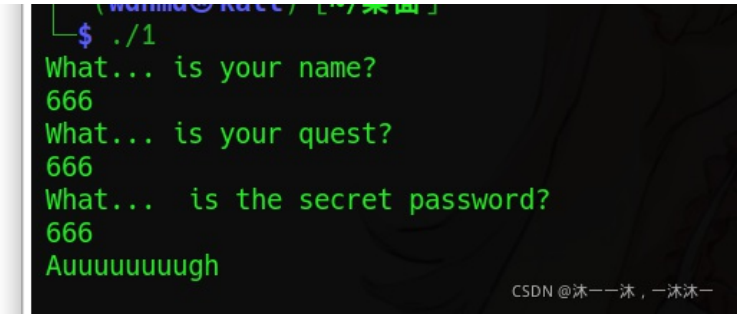
flag+=key2[i]

print(flag)

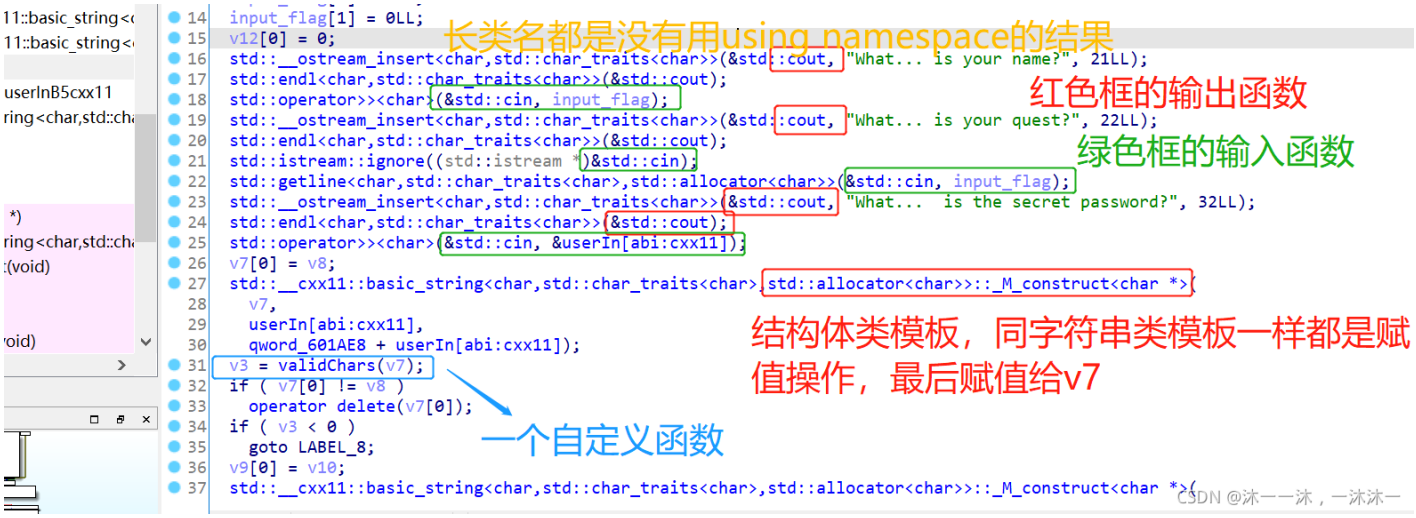
结果：

攻防世界reverse-for-the-holy-grail-350：（冗余中锁定关键代码、模板赋值操作、多层加密、倍数条件算法积累、地址差值操作、超位数循环截取算法、正向爆破）

64位ELF文件，无壳，运行一下程序看一下主要显示字符串信息：



然后照例扔入IDA64中查看伪代码信息，有main函数看main函数：

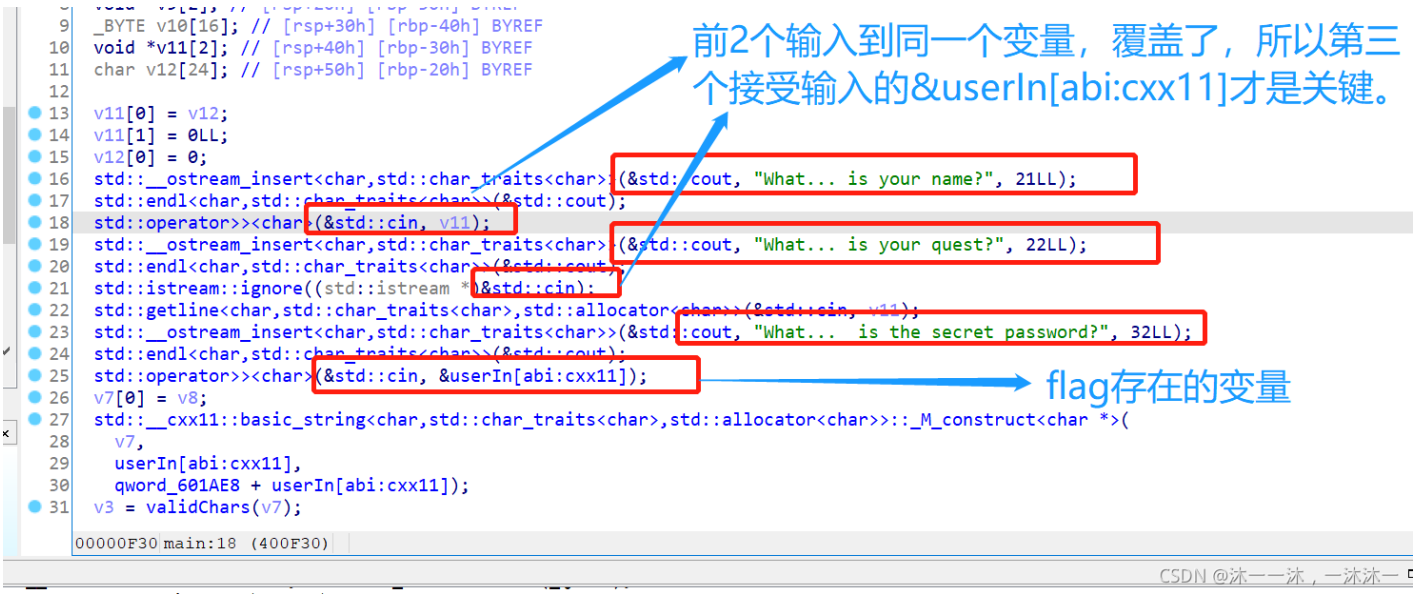


C++前面做题经验中说过了看得是最后那个函数名，因为C++是面向对象的，没有使用"using namespace 类名;"时就没有对应的命名空间，就只能用长类名导入函数。

附上以前的笔记：

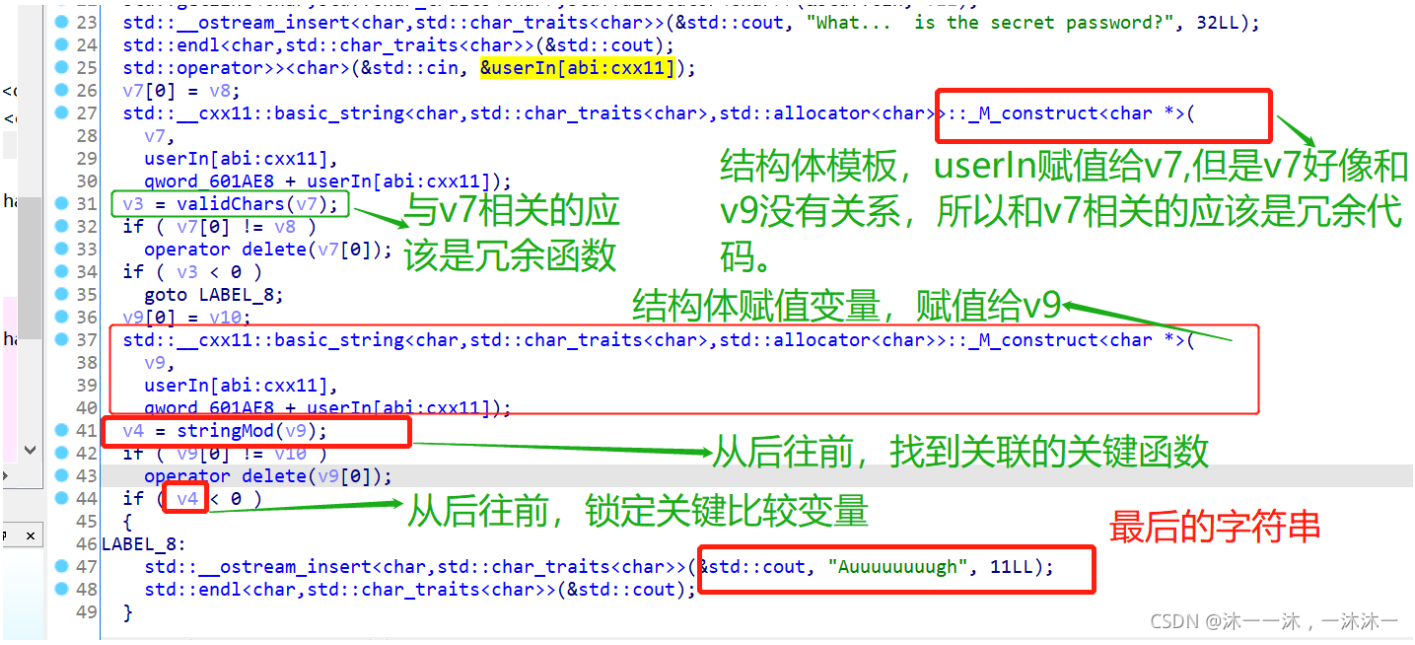
using 指令：
您可以使用 using namespace 指令，这样在使用命名空间时就可以不用在前面加上命名空间的名称。这个指令会告诉编译器，后续的代码将使用指定的命名空间中的名称。

所以下图1~4的红框都是C++的cout输出函数和cin输入函数，但是cin输入的都是v11，v11能被覆盖就说明不是关键，4、5红框中cin的对象是&userIn[abi:cxx11]，这是一个地址，应该就是我们要的flag了



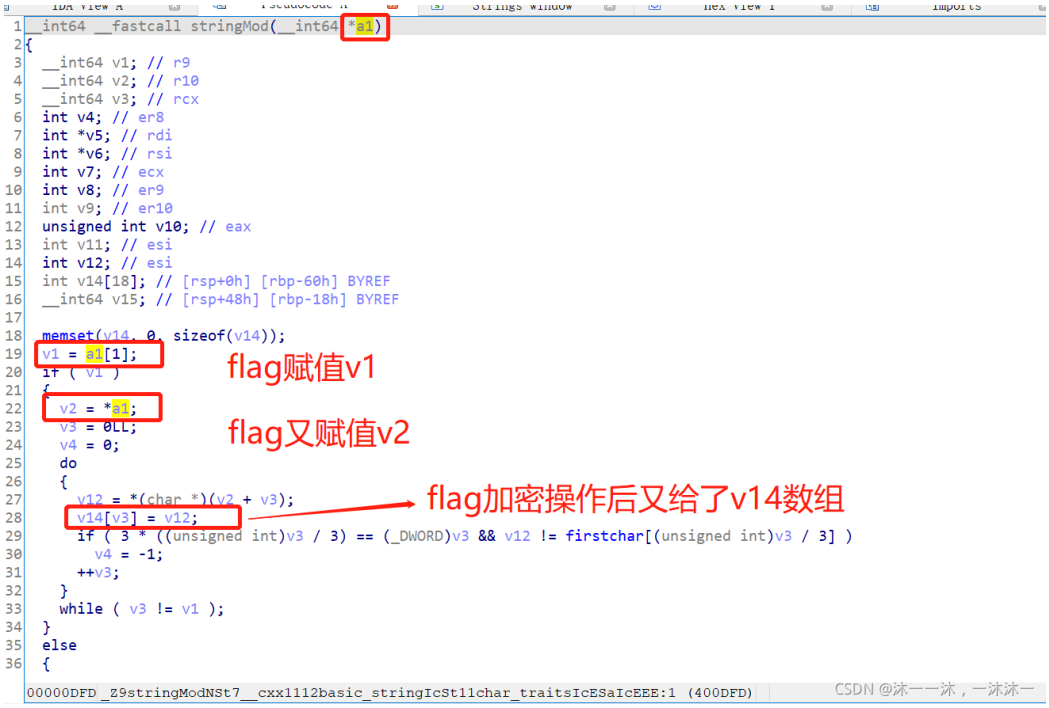
(这里积累第二个经验)

然后后面代码就渐渐看不懂了，只能从后面往前推，最后一个Auuuuuuuugh是我们前面运行时显示的，由它出发跟踪到v4，前面v4 = stringMod(v9);有一个自定义函数，而自定义函数通常是关键。然后再前面的_M_construct<char *>结构体应该是简单的赋值吧。与v7相关的应该是冗余代码了。



(这里积累第三个经验)

双击跟踪入stringMod函数，代码有点长，以为有冗余代码，按照以前的经验，查找与输入相关的代码，如下面红框所示：a1是输入的flag，a1给了v2，v2给了v14，v14给了v5和v6，这些被赋值了flag地址的变量串起了整个函数，所以这里没有冗余代码，都是关键代码：

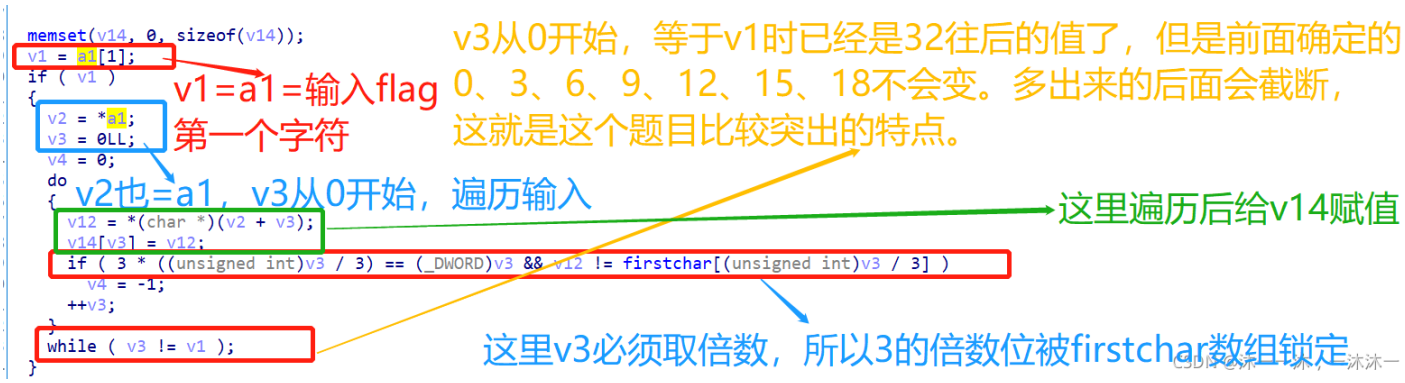


(这里积累第四个经验)

分析第一个循环，有点意思，循环条件是v3 != v1 ，而v1 = a1[1]，v3是从0开始的数，v1是一个字符，怎么也是从32开始的。所以这里一开始用从0一直循环到字符的ASCII码，明显超出了flag输入的位数，一开始我也很懵，后来发现后面有flag位数限制，所以这里超了范围又有什么关系呢，后面限制回来不就行了？(后面限制了这个flag位数为18位)

然后就是这里的if语句，除法符号 '/' 会有余数，所以这里必须是3的倍数，这里的v4=-1一开始我不确定执行有没有影响，后来发现最后return (unsigned int)(v7 * v4); 语句表明这个if语句不能执行，所以 i 为 3 的倍数时flag必须等于firstchar数组内的元素。

那么第一个循环就确认了flag的0、3、6、9、12、15、18位。

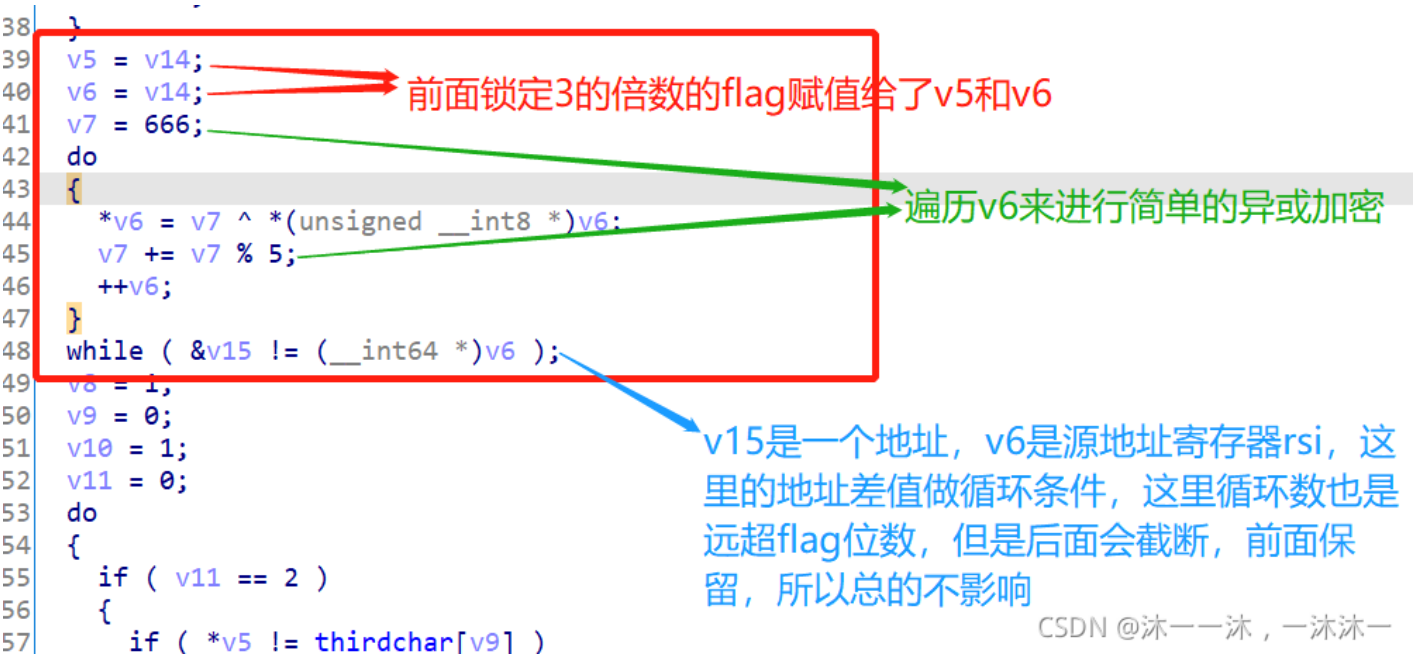


(这里积累第五个经验)

接下来分析第二个循环，这里必经一个循环，循环条件也是很有意思&v15 != (__int64 *)v6 取v15的地址和v6作比较，v6也是一个地址，关键是v6是rsi源地址寄存器，没法跟踪栈内位置。后来查了很多资料，有人说rsi相当于rbp的位置，如第二幅图所示，v15地址是rbp-18，所以循环条件是18，也就是输入的flag的位数。

但是也有人说rsi是rsp的位置，所以这里相差是48，和前面一样超出了flag的位数，但是最后的循环中限制了取18位进行flag操作，所以也不影响。(我更支持这个！)

那么这里必经的循环就是简单的异或操作的，和以前总结的一层、二层加密一样，第一个循环的数没有加密，后面循环的数都是二层加密，解密时要考虑异或回来。



(这里积累第6个经验)

分析最后一个循环，这个循环比较复杂，所以放上总图一步步来：

先看第一个嵌套的 if 条件后面就是不同位数的逻辑代码了，因为只有18位，所以一个个手算其实更方便：

先看v11的变化，v11从0开始：

0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0

再看v9的变化，v9从0开始，下面v9与v11对应：

0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5 6 6 6 7 7 7 8 8 8 9 9 9 10 10 10

然后看v5的变化，v5是输入flag的首地址但是经过异或加密，和第一个循环一样if成立则对应固定的异或后的位数：

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18

标注的是v5对应的位数，也就是thirdchar数组对应着异或后的flag的2，5，8，11，14，17位

同样的在看第二个if条件对应的位数，这里v10和v15的关系有前又有后，其实跟踪起来还是很绕的，但是前面是2，5，8、0，3，6。也应该要想到这里是1，4，7。18位的flag三层循环对应三份位数。

v11：

0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0

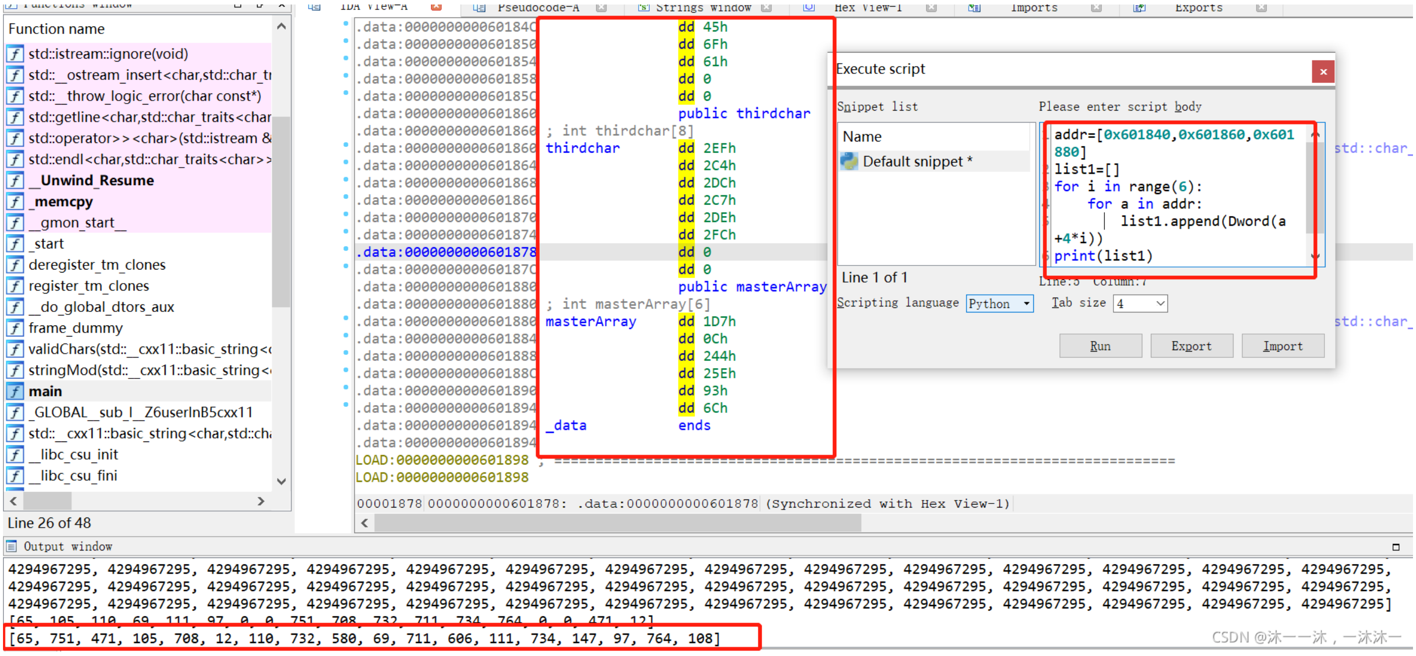
v10从1开始，v10 *= *v5，因为v10在v5之前，所以v10永远比v5少一：

1 v5[0] v5[0]v5[1] 1 v5[3] v5[3]v5[4] 1 v5[6] v5[6]v5[7] 1 v5[9] v5[9]v5[10] 1 v5[12] v5[12]v5[13] 1 v5[15] v5[15]v5[16]

v5，因为v5在v10赋值之后，所以v5永远比v10多一：

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18

流程基本梳理完了，现在可以写脚本了，在那之前先dump下三个数组内容：



逆向逻辑脚本：

key1=[65, 105, 110, 69, 111, 97,]

key2=[751, 708, 732, 711, 734, 764]

key3=[471, 12, 580, 606, 147, 108]

flag=[0 for i in range(18)]

index=0

for i in range(0,18,3): #一层截取，锁定0、3、6、9、12、15、18

flag[i]=key1[index]

index+=1

print(flag)

v7=666

xor=[]

for i in range(18): #二层全员异或加密

xor.append(v7)

v7+=v7%5

print(xor)

index2=0

for i in range(2,18,3): #三层异或后的截取，锁定2、5、8、11、14、17

flag[i]=key2[index2]^xor[i]

index2+=1

print(flag)

index3=0

for i in range(1,18,3): #三层异或后爆破，逻辑正向流程复现，要处理二层的异或加密。锁定1、4、7、10、13、16

for a in range(32,128,1): #ascii字符遍历爆破

if ((flag[i-1]^xor[i-1])*(a^xor[i]))%(flag[i+1]^xor[i+1])==key3[index3]:

flag[i]=a

index3+=1

break

print('tuctf{'+''.join(map(chr,flag))+'}')

(这里积累第7个经验)

注意后面的逆向要考虑回一层的异或加密，所以这里直接把异或加密位数导出来，因为flag只有18位，所以我们也取前18位即可。

```
v7=666
xor=[]
for i in range(18):
    xor.append(v7)
    v7+=v7%5
print(xor)
```

然后就是这个v10 % *v5 != masterArray[v9]的逆向，这个很难逆，我看了很多资料都显示直接正向爆破。因为 i-1和 i+1在前面循环中都获取了，所以这里可以直接用，然后注意时刻考虑一层的异或加密即可。

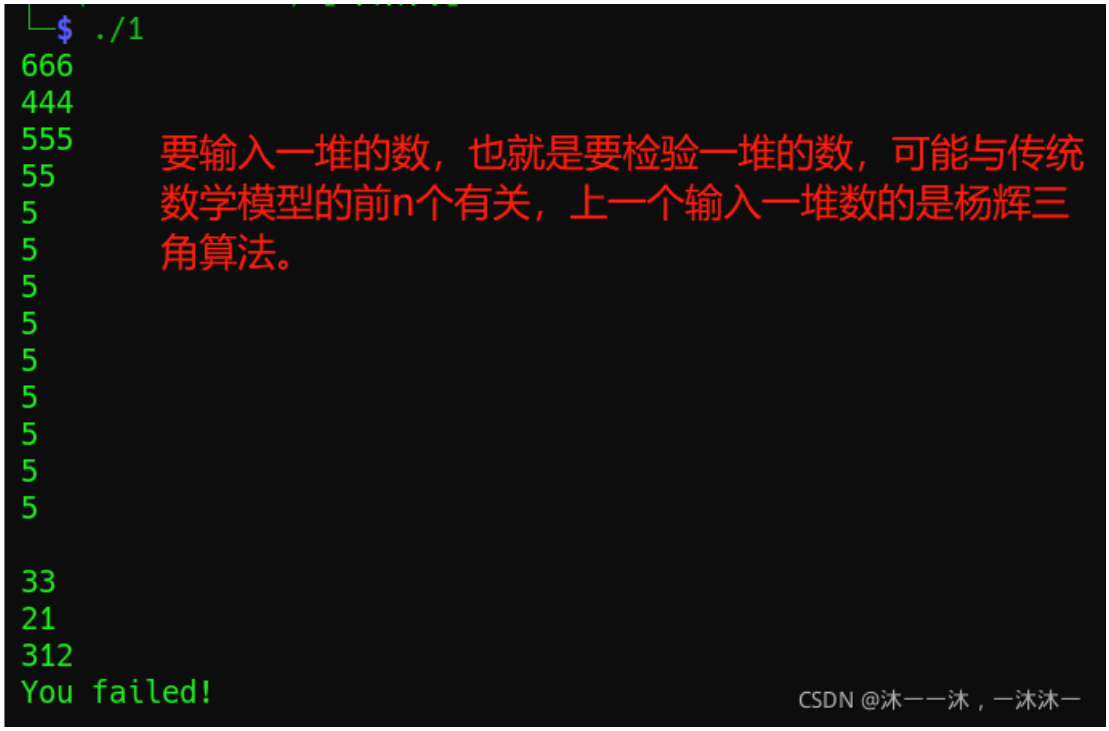
```
for i in range(1,18,3):
    for a in range(32,128,1):
        if ((flag[i-1]^xor[i-1])*(a^xor[i]))%
(flag[i+1]^xor[i+1])==key3[index3]:
```

结果：

```
└─$ python 1.py
[65, 0, 0, 105, 0, 0, 110, 0, 0, 69, 0, 0, 111, 0, 0, 97, 0, 0]
[666, 667, 669, 673, 676, 677, 679, 683, 686, 687, 689, 693, 696, 697, 699, 703, 706, 707]
[65, 0, 114, 105, 0, 97, 110, 0, 114, 69, 0, 114, 111, 0, 101, 97, 0, 63]
tuctf{AfricanOrEuropean?}
```

攻防世界easyCpp：（函数积累、函数逻辑封装、lambda自定义函数、IDA动态调试、动调验证值猜想、斐波那契数列算法）

64位ELF文件，无壳，照例运行一下程序，看看主要回显信息：



一开始啥也没有，就一直乱输入，然后弹出个you failed字符串，信息够了。照例扔入IDA64中查看伪代码，有main函数看main函数：（这里我一开始分析时注释了一些代码，这并不影响）：

（这里积累第一个经验）

一开始大致浏览了一下，发现很多的STL（标准模板库）中关于向量模板，容器的vector考点。刚好有笔记记录过，所以先搬上一些函数作用：

push_back( ) 成员函数在向量的末尾插入值，如果有必要会扩展向量的大小。

size( ) 函数显示向量的大小。

begin( ) 函数返回一个指向向量开头的迭代器。

end( ) 函数返回一个指向向量末尾的迭代器。

（back_inserter（尾部插）,inserter（插入指定位置）,front_inserter（头部插））

因为代码有很多是看不懂的，所以我搬用了高三英语阅读的方法，懂的全部注释，再慢慢前后连接起来，这部分主要就是fib(j)的斐波那契数列函数要注意了：

```
v32 =   readfsqword(0x28u);
std::vector<int>::vector(v24, argv, envp);      // 初始化向量变量
std::vector<int>::vector(v25, argv, v3);
std::vector<int>::vector(v26, argv, v4);
std::vector<int>::vector(v27, argv, v5);
std::vector<int>::vector(v28, argv, v6);
for ( i = 0; i <= 15; ++i )
{
  scanf("%d", &v31[i]);
  std::vector<int>::push_back(v25, &v31[i]);    // v31的18位数组接受16个输入，然后给了v25的32位数组。
}
for ( j = 0; j <= 15; ++j )
{
  LODWORD(v30[0]) = fib(j);
  std::vector<int>::push_back(v24, v30);        // 一开始没查fib(j)函数的作用，所以以为是简单的把0~15赋值给v24数组，
                                                // 后来发现fib(j)是斐波那契数列生成函数，也就是这里生成了16个斐波那契数
}
```


（这里积累第二个经验）

然后分析下一部分函数，一开始觉得红框部分的v30[0] = std::vector<int>::begin(v25);只是简单地把输入的第0个值给了v30而已，紧接着后面的v9也不明觉厉。动态调试了才发现是v30整个指向了接受输入的v25数组，所以v9也就指向了输入的第二个字符了，这样也联系得通了，所以有时候一些细枝末节真的要前后联系起来解释才行。

然后就是一个lambda表达式了，回顾了笔记发现是自定义函数，一开始并不知道入口在哪里，查了资料才明白函数名在前面紧接着std的transform那里，不过直接双击lambda也是可以跟踪的：

```
std::vector<int>::push_back(v26, v31);          // 把接受输入的v31数组的赋值给了v26数组
v7 = std::back_inserter<std::vector<int>>(v26);// v7指向接受输入的v26数组
v8 = std::vector<int>::end(v25);                // v8指向同样是接受输入的v25数组的最后一位
v30[0] = std::vector<int>::begin(v25);          // v30第一位指向同样接受输入的v25数组的0位
v9 = __gnu_cxx::__normal_iterator<int *,std::vector<int>>::operator+(v30, 1LL);// v9指向v30的第一位，operator +是运算符加操作
std::transform<__gnu_cxx::__normal_iterator<int *,std::vector<int>>,std::back_insert_iterator<std::vector<int>>,main::{lambda(i
  v9,
  v8,                                           // lambda表达式
  v7,
  (__int64)v31);
```


继续后面的大体分析，把接受输入的数组赋值来赋值去，最后和斐波那契数列的v24数组作比较，那么总的流程就是用户输入------>两个lambda自定义函数修改-------->与斐波那契数列数比较

```
std::vector<int>::vector(v31, v9, v11);         // 又初始化向量
v12 = std::vector<int>::end(v28);               // v8低字指向接受输入的v26数组最后一位
v13 = std::vector<int>::begin(v28);             // v11指向接受输入的v26第0位
std::accumulate<__gnu_cxx::__normal_iterator<int *,std::vector<int>>,std::vector<int>,main::{lambda(std::vector<int>,int)#2}>(
  (__int64)v32,
  v13,
  v12,
  (__int64)v31,                                 // lambda表达式
  v14,
  v15,
  v3);
std::vector<int>::operator=(v29, v32);          // v27=v30，现在v27也是指向接受输入的数组了
std::vector<int>::~vector(v32);
std::vector<int>::~vector(v31);                 // 又声明两个向量
if ( (unsigned __int8)std::operator!=<int,std::allocator<int>>(v29, v26) )// 如果接受输入的数组v27!=v24，operator!=才是关键，后面不是
                                                // v24是前面接受斐波那契数列的数组，这里也是第一个错误，32位数组只用前16个被赋值了的作比较
{
  puts("You failed!");
  exit(0);
}
```


(这里积累第三个经验)

分析第一个自定义transform函数，根据外层分析的重命名了很多变量，然后一开始没注意的就是红框处的back_insert_iterator尾部插入函数，因为把注意力都放在了前面的operator*运算符那里，所以这个函数的作用就是从第二个数开始每一个数都加上第一个数然后返回：

```
__int64 __fastcall std::transform<__gnu_cxx::__normal_iterator<int *,std::vector<int>>,std::back_insert_iterator<std::vector<int>>,mai
{
  int *v4; // rax
  __int64 v5; // rax
  _DWORD *input_too; // [rsp+0h] [rbp-30h] BYREF
  __int64 input; // [rsp+8h] [rbp-28h] BYREF
  __int64 last_input; // [rsp+10h] [rbp-20h] BYREF
  __int64 second_input; // [rsp+18h] [rbp-18h] BYREF
  int v11; // [rsp+24h] [rbp-Ch] BYREF
  unsigned __int64 v12; // [rsp+28h] [rbp-8h]

  second_input = a1;
  last_input = a2;
  input = a3;
  input_too = (_DWORD *)a4;
  v12 = __readfsqword(0x28u);
  while ( (unsigned __int8)__gnu_cxx::operator!=<int *,std::vector<int>>(&second_input, &last_input) )// 又是不等于 != 的判断 operator !
  {
    v4 = (int *)__gnu_cxx::__normal_iterator<int *,std::vector<int>>::operator*(&second_input);// 第二个字符开始
    v11 = main::{lambda(int)#1}::operator()(&input_too, *v4);// v11=第一个输入字符加第二个输入字符
    v5 = std::back_insert_iterator<std::vector<int>>::operator*(&input);// v5=第一个输入字符，back_insert_iterator是尾部插入。
    std::back_insert_iterator<std::vector<int>>::operator=(v5, &v11);// v5=v11=第一个输入字符加第二个输入字符，back_insert_iterator是尾部
    __gnu_cxx::__normal_iterator<int *,std::vector<int>>::operator++(&second_input);// 第二个输入字符++1，所以前面v11也会向后取
    std::back_insert_iterator<std::vector<int>>::operator++(&input);// 输入地址++1，back_insert_iterator是尾部插入
  }
  return input;
}
```

一些C++函数的关键函数名在std后面

通常要搭配动态调试才能更准确地判断结果，这个动态调试也是学到了怎么寻找跟踪对象了，而且要注意的是不要跟踪函数内的局部变量，要跟踪函数外才行啊！！！然后就是变量类型对应后的地址跳转也要注意。

```
  std::vector<int>::vector(v27, argv, v5);
  std::vector<int>::vector(v28, argv, v6);
  for ( i = 0; i <= 15; ++i )
  {
    scanf("%d", &v31[i]);
    std::vector<int>::push_back(v25, &v31[i]);
  }
  for ( j = 0; j <= 15; ++j )
  {
    LODWORD(v30[0]) = fib(j);
    std::vector<int>::push_back(v24, v30);
  }
  std::vector<int>::push_back(v26, v31);
  v7 = std::back_inserter<std::vector<int>>(v26);
  v8 = std::vector<int>::end(v25);
  v30[0] = std::vector<int>::begin(v25);
  v9 = __gnu_cxx::__normal_iterator<int *,std::vector<int>>::operator+(v30, 1LL);
  std::transform<__gnu_cxx::__normal_iterator<int *,std::vector<int>>,std::back_insert_iterator<std::vector<int>>,main::{lambda(int)#1}>(
    v9,
    v8,
    v7,
    (__int64)v31);
  std::vector<int>::vector(v29, v8, v10);
  LODWORD(v8) = std::vector<int>::end(v26);
  v11 = std::vector<int>::begin(v25);
  std::accumulate<__gnu_cxx::__normal_iterator<int *,std::vector<int>>,std::vector<int>,main::{lambda(std::vector<int>,int)#2}>(
    (unsigned int)v30,
    v11,
    v8,
    (unsigned int)v29,
    v12,
    v13);
  std::vector<int>::operator=(v27, v30);
  std::vector<int>::~vector(v30);
  std::vector<int>::~vector(v29);
  if ( (unsigned __int8)std::operator!=<int,std::allocator<int>>(v27, v24) )
  {
```

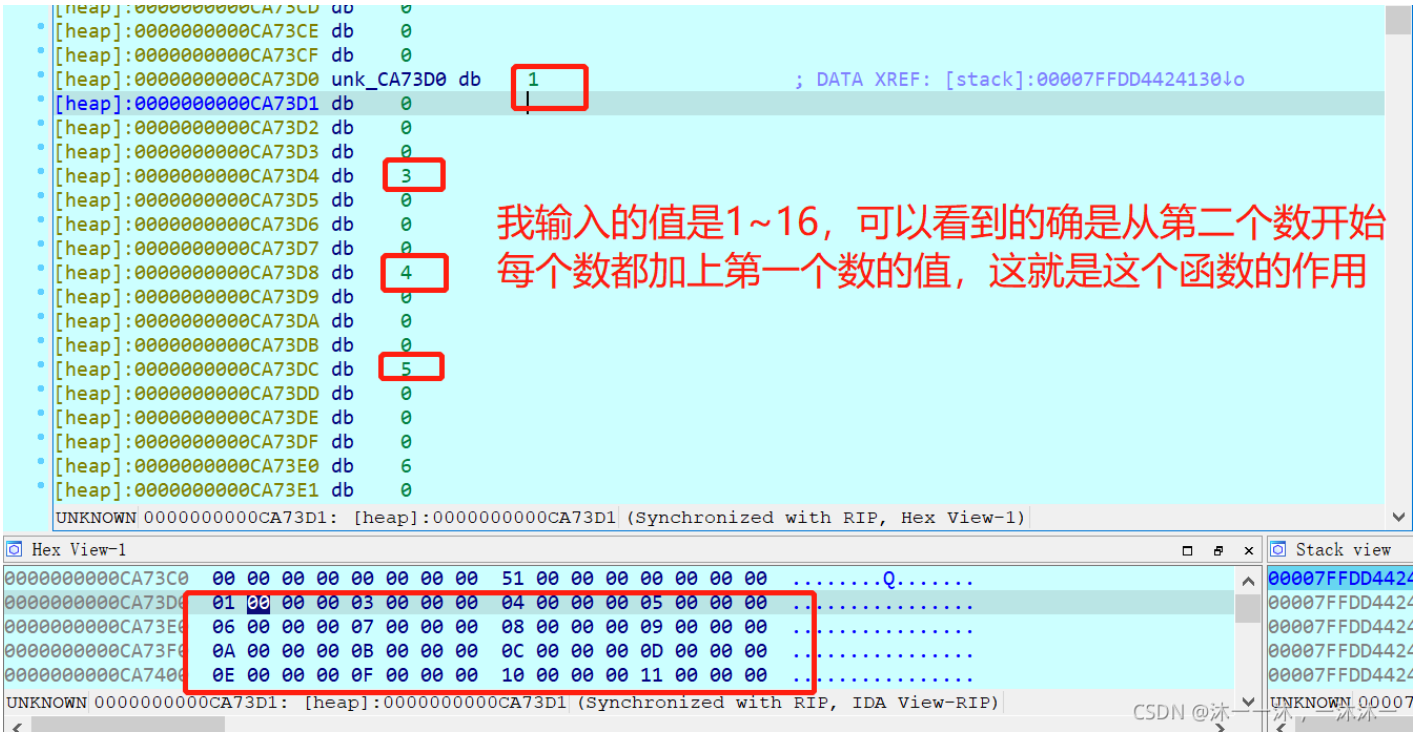
接受输入的v31给了v26，
v26又加在了v7的末尾，相当于v7=v26=v31=用户输入了

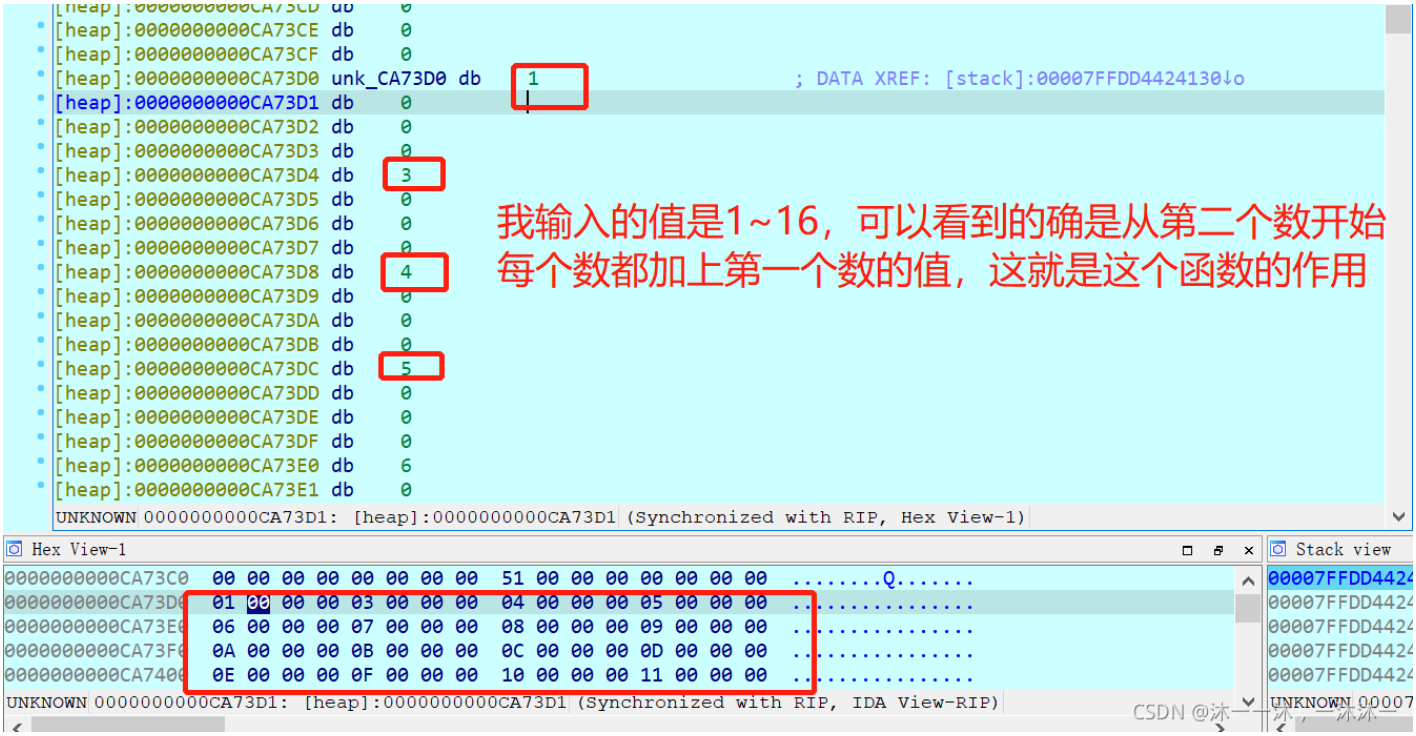
后来发现加密后是返回了v7，但是必须要在transform函数外看栈值才行，函数内看栈就是另外一个值了，因为函数内的v7是局部变量。

一开始不知道transform加密后给了谁，于是索性跟踪v26


```
[stack]:00007FFDD442412E db    0
[stack]:00007FFDD442412F db    0
R12 [stack]:00007FFDD4424130 dq offset unk_CA73D0
RSI [stack]:00007FFDD4424138 dq offset dword_CA7410
    [stack]:00007FFDD4424140 dq offset dword_CA7410
    [stack]:00007FFDD4424148 dq offset _ZSt4wcin
    [stack]:00007FFDD4424150 dq 0
    [stack]:00007FFDD4424158 db    0
    [stack]:00007FFDD4424159 db    0
    [stack]:00007FFDD442415A db    0
    [stack]:00007FFDD442415B db    0
    [stack]:00007FFDD442415C db    0
    [stack]:00007FFDD442415D db    0
    [stack]:00007FFDD442415E db    0
    [stack]:00007FFDD442415F db    0
    [stack]:00007FFDD4424160 db    0
```

注意变量类型，v7是_int64的，这里data值地址直接调转，后续的值也都在跳转那里，不在这里了





分析第二个自定义accumulate函数，函数分析同前面transform函数，由于我很多注释被消掉了，所以只能简单回顾一下了。不过这里跟踪的是v27，外层v30不知道那里被修改过，跟踪不了。

IDA View-A | Pseudocode-A | Strings window | Hex View-1 | Imports | Exports

```
__int64 __fastcall std::accumulate<__gnu_cxx::__normal_iterator<int *,std::vector<int>>,std::vector<int>,main::{lambda(std::vector<int
{
  unsigned int v7; // ebx
  __int64 v10; // [rsp+8h] [rbp-68h] BYREF
  __int64 v11; // [rsp+10h] [rbp-60h] BYREF
  __int64 v12; // [rsp+18h] [rbp-58h]
  char v13[32]; // [rsp+20h] [rbp-50h] BYREF
  char v14[24]; // [rsp+40h] [rbp-30h] BYREF
  unsigned __int64 v15; // [rsp+58h] [rbp-18h]

  v12 = a1;
  v11 = a2;
  v10 = a3;
  v15 = __readfsqword(0x28u);
  while ( (unsigned __int8)__gnu_cxx::operator!=<int *,std::vector<int>>(&v11, &v10) )
  {
    v7 = *(_DWORD *)__gnu_cxx::__normal_iterator<int *,std::vector<int>>::operator*(&v11);
    std::vector<int>::vector(v13, a4);
    main::{lambda(std::vector<int>,int)#2}::operator()(v14, &a7, v13, v7);
    std::vector<int>::operator=(a4, v14);
    std::vector<int>::~vector(v14);
    std::vector<int>::~vector(v13);
    __gnu_cxx::__normal_iterator<int *,std::vector<int>>::operator++(&v11);
  }
  std::vector<int>::vector(v12, a4);
  return v12;
}
0000133F _ZSt10accumulateIN9__gnu_cxx17__normal_iteratorIPiSt6vectorIiSaIiEEEES5_Z4mainEUlS5_iE0_ET0_T_S9_S8_T1_ 0000133F
```

进行操作的只有这个自定义lambda函数。

IDA View-A | Pseudocode-A | Strings window | Hex View-1 | Imports | Exports

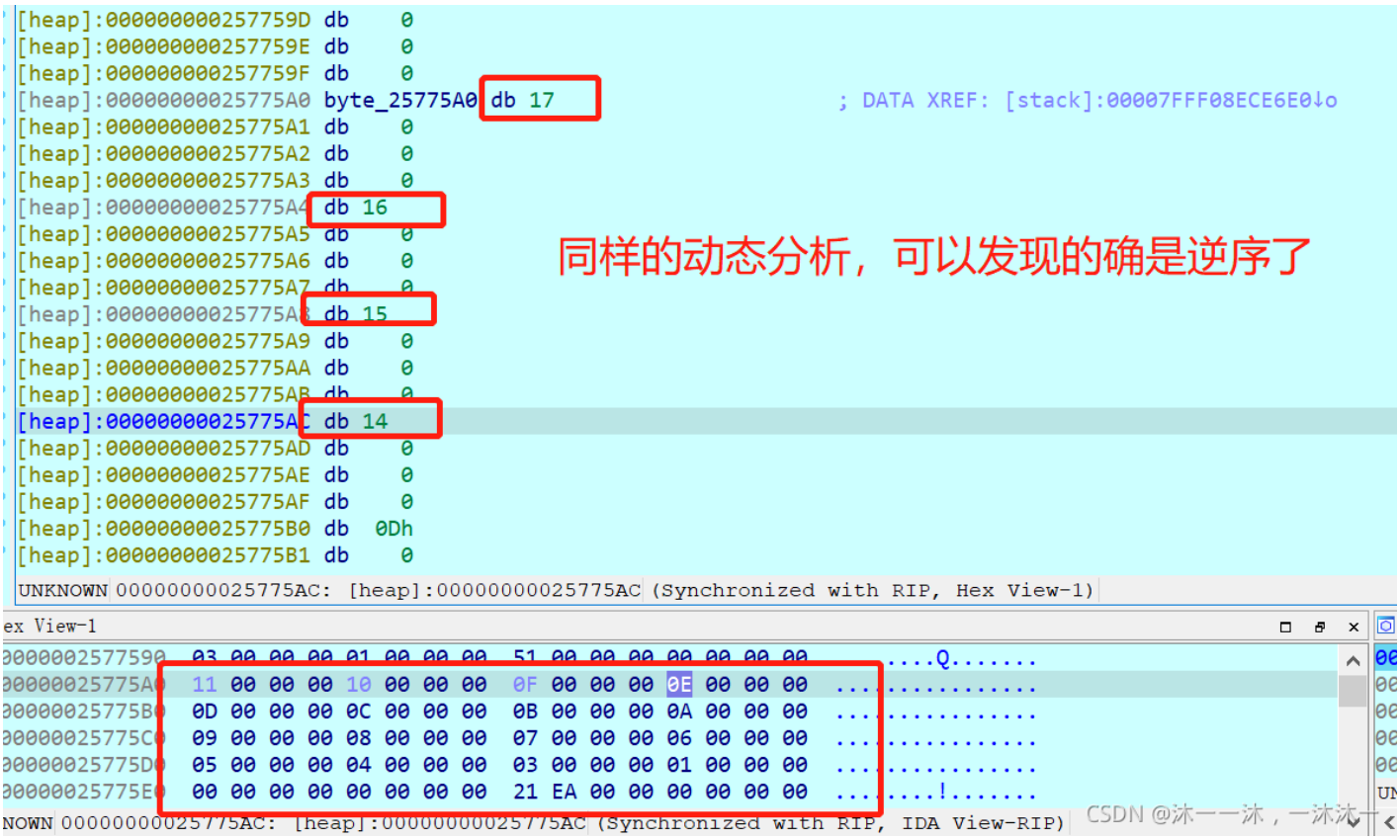
```
__int64 __fastcall main::{lambda(std::vector<int>,int)#2}::operator()(__int64 a1, __int64 a2, __int64 a3, int a4)
{
  __int64 v4; // r12
  __int64 v5; // rbx
  __int64 v6; // rax
  int v8; // [rsp+4h] [rbp-3Ch] BYREF
  __int64 v9; // [rsp+8h] [rbp-38h]
  __int64 v10; // [rsp+10h] [rbp-30h]
  __int64 v11; // [rsp+18h] [rbp-28h]
  unsigned __int64 v12; // [rsp+28h] [rbp-18h]

  v11 = a1;
  v10 = a2;
  v9 = a3;
  v8 = a4;
  v12 = __readfsqword(0x28u);
  std::vector<int>::vector(a1, a2, a3);
  std::vector<int>::push_back(a1, &v8);
  v4 = std::back_inserter<std::vector<int>>(v11);
  v5 = std::vector<int>::end(v9);
  v6 = std::vector<int>::begin(v9);
  std::copy<__gnu_cxx::__normal_iterator<int *,std::vector<int>>,std::back_insert_iterator<std::vector<int>>>(
    v6,
    v5,
    v4);
  return v11;
}
```

关键函数名在前面

lambda函数里面是首位copy的逆序操作


```
    (__int64)v31);
  std::vector<int>::vector(v29, v8, v10);
  LODWORD(v8) = std::vector<int>::end(v26);
  v11 = std::vector<int>::begin(v26);
  std::accumulate<__gnu_cxx::__normal_iterator<int *,std::vector<int>>,std::vector<int>,main::{lambda(std::v
    (unsigned int)v30,
    v11,
    v8,
    (unsigned int)v29,
    v12,
    v13);
  std::vector<int>::operator=(v27, v30);
  std::vector<int>::~vector(v30);
  std::vector<int>::~vector(v29);
  if ( (unsigned __int8)std::operator!=<int,std::allocator<int>>(v27, v24) )
  {
    puts("You failed!");
```

这里跟踪的是v27，v30不明白那里被修改了，跟踪不了哦。



回顾之前总的流程：

用户输入------>第一次修改，从第二个数开始每个数加上第一个数--------->第二次修改，逆序-------->与斐波那契数列数比较

那么逆向逻辑就是：

16个斐波那契数列数---->逆序------->从第二个数开始每个数减第一个数---------->获得输入

开始写脚本，先上网复制个斐波那契数生成函数，然后直接逆向逻辑即可：

```
def fib(n): #网上找的斐波那契数生成函数

return int(1 and n<=2 or fib(n-1)+fib(n-2))

list1=[]

input1=[]

for i in range(1,17):

list1.append(fib(i))

list1=list1[::-1]

input1.append(list1[0])

for i in range(1,len(list1)):

input1.append(list1[i]-list1[0])

print(input1)
```

结果：

```
└─$ python 1.py
[987, -377, -610, -754, -843, -898, -932, -953, -966, -974, -979, -982, -984, -985, -986, -
986]
```

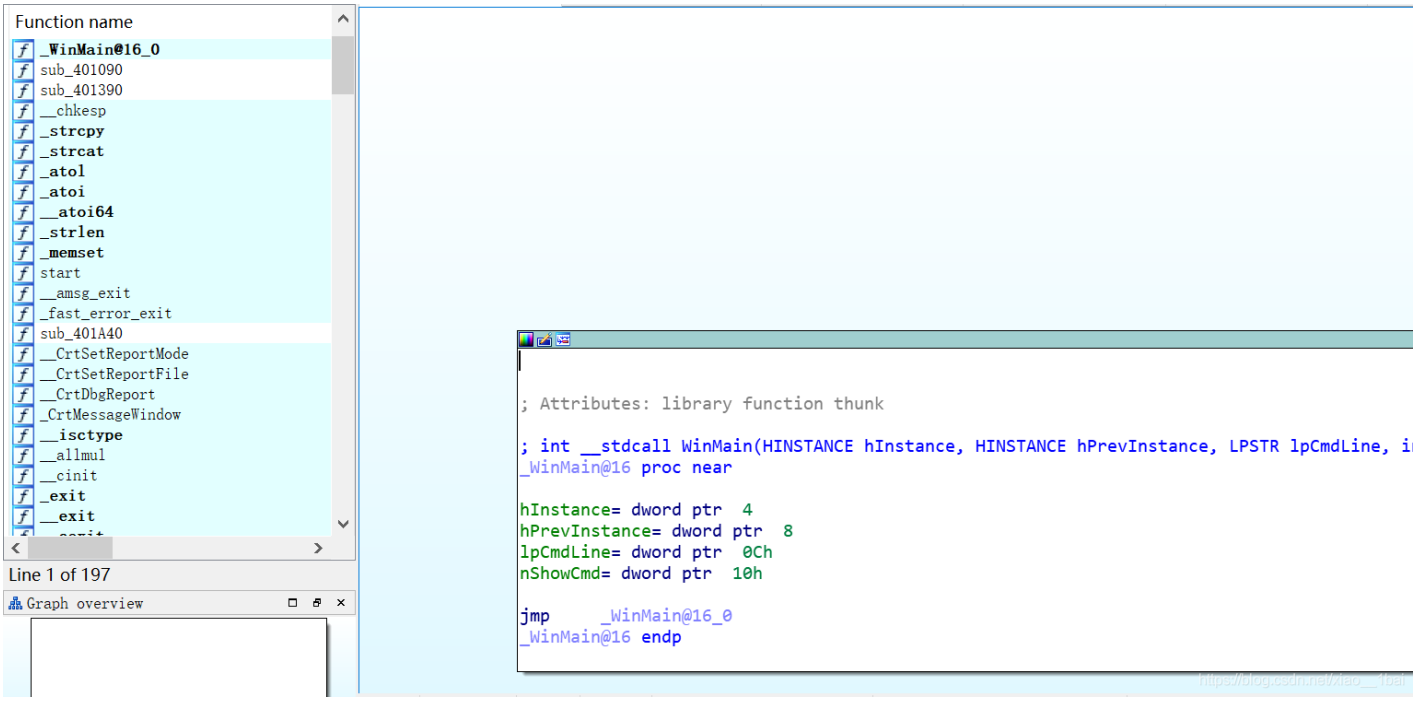
```
└─$ ./1
987
-377
-610
-754
-843
-898
-932
-953
-966
-974
-979
-982
-984
-985
-986
-986
You win!
Your flag is:flag{987-377-843-953-979-985}
```


无main函数分析（C语言）

主逻辑平铺一函数内：

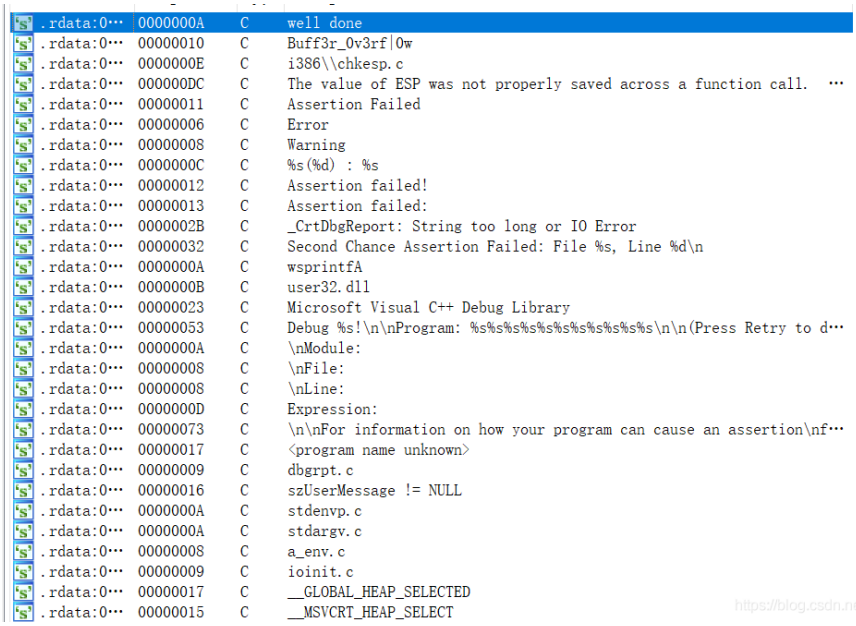
攻防世界Mysterious：（地址小端存放与正向，出人意料的flag）

W32可执行文件，无壳，扔入IDA32中看伪代码判断题目类型：



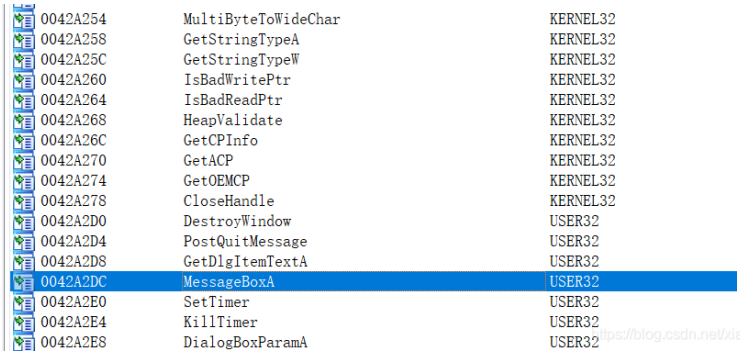
没有主函数，看来是非正常文件，双击程序看看有什么信息可以提取：

一个输入密码型弹框，Crack按钮按不下去，信息够了，查看IDAstring窗口：



.rdata:0…	0000000A	C	well done
.rdata:0…	00000010	C	Buff3r_0v3rf|0w
.rdata:0…	0000000E	C	i386\\chkesp.c
.rdata:0…	000000DC	C	The value of ESP was not properly saved across a function call. …
.rdata:0…	00000011	C	Assertion Failed
.rdata:0…	00000006	C	Error
.rdata:0…	00000008	C	Warning
.rdata:0…	0000000C	C	%s(%d) : %s
.rdata:0…	00000012	C	Assertion failed!
.rdata:0…	00000013	C	Assertion failed:
.rdata:0…	0000002B	C	_CrtDbgReport: String too long or IO Error
.rdata:0…	00000032	C	Second Chance Assertion Failed: File %s, Line %d\n
.rdata:0…	0000000A	C	wsprintfA
.rdata:0…	0000000B	C	user32.dll
.rdata:0…	00000023	C	Microsoft Visual C++ Debug Library
.rdata:0…	00000053	C	Debug %s!\n\nProgram: %s%s%s%s%s%s%s%s%s%s\n\n(Press Retry to d…
.rdata:0…	0000000A	C	\nModule:
.rdata:0…	00000008	C	\nFile:
.rdata:0…	00000008	C	\nLine:
.rdata:0…	0000000D	C	Expression:
.rdata:0…	00000073	C	\n\nFor information on how your program can cause an assertion\nf…
.rdata:0…	00000017	C	<program name unknown>
.rdata:0…	00000009	C	dbgrpt.c
.rdata:0…	00000016	C	szUserMessage != NULL
.rdata:0…	0000000A	C	stdenvp.c
.rdata:0…	0000000A	C	stdargv.c
.rdata:0…	00000008	C	a_env.c
.rdata:0…	00000009	C	ioinit.c
.rdata:0…	00000017	C	__GLOBAL_HEAP_SELECTED
.rdata:0…	00000015	C	__MSVCRT_HEAP_SELECT

没有发现input the password字眼，应该是隐藏了，想起弹框是用了Messagebox的windowsAPI函数，于是双击跟踪该函数：（要在import窗口才能跟踪，string窗口不行，因为import是导入API外部函数的窗口,string窗口那里可能只是刚好有同名字符串而已）



0042A254	MultiByteToWideChar	KERNEL32
0042A258	GetStringTypeA	KERNEL32
0042A25C	GetStringTypeW	KERNEL32
0042A260	IsBadWritePtr	KERNEL32
0042A264	IsBadReadPtr	KERNEL32
0042A268	HeapValidate	KERNEL32
0042A26C	GetCPInfo	KERNEL32
0042A270	GetACP	KERNEL32
0042A274	GetOEMCP	KERNEL32
0042A278	CloseHandle	KERNEL32
0042A2D0	DestroyWindow	USER32
0042A2D4	PostQuitMessage	USER32
0042A2D8	GetDlgItemTextA	USER32
0042A2DC	MessageBoxA	USER32
0042A2E0	SetTimer	USER32
0042A2E4	KillTimer	USER32
0042A2E8	DialogBoxParamA	USER32

```
{
  if ( a3 == 1000 )
  {
    GetDlgItemTextA(hWnd, 1002, &String, 260);
    strlen(&String);
    if ( strlen(&String) > 6 )
      ExitProcess(0);
    v10 = atoi(&String) + 1;
    if ( v10 == 123 && v12 == 120 && v14 == 122 && v13 == 121 )
    {
      strcpy(Text, "flag");
      memset(&v7, 0, 0xFCu);
      v8 = 0;
      v9 = 0;
      _itoa(v10, &v5, 10);
      strcat(Text, "{");
      strcat(Text, &v5);
      strcat(Text, "_");
      strcat(Text, "Buff3r_0v3rf|0w");
      strcat(Text, "}");
      MessageBoxA(0, Text, "well done", 0);
    }
    SetTimer(hWnd, 1u, 0x3E8u, TimerFunc);
  }
  if ( a3 == 1001 )
    KillTimer(hWnd, 1u);
}
return 0;
}
```


逻辑很简单，真的简单，但我就是错了：

错误1：习惯性的字符串反转，这里不是内存操作，就是打印Text这个字符串，所以不用反转：

MessageBoxA(0, Text, "well done", 0);

错误2：我把v10的123转成ASCII字符了，对应的字符是{，于是我就得到一个神奇的flag：

flag{{_Buff3r_0v3rf|0w}

这个当然是错的，关键是我还直接以为这是假的flag代码而去寻找其它函数去了。

后面就引发了一系列问题，比如设想Crack按钮按不下是不是要动态调整调整跳转等等：

结果是失败的，我都不知道它是从那个函数跳出来的！！！

后来查了资料(WP)才发现，flag的确在这里，关键是{写成123即可，不用转ASCII字符，想想也对flag就是数字字符的结合啊！！！

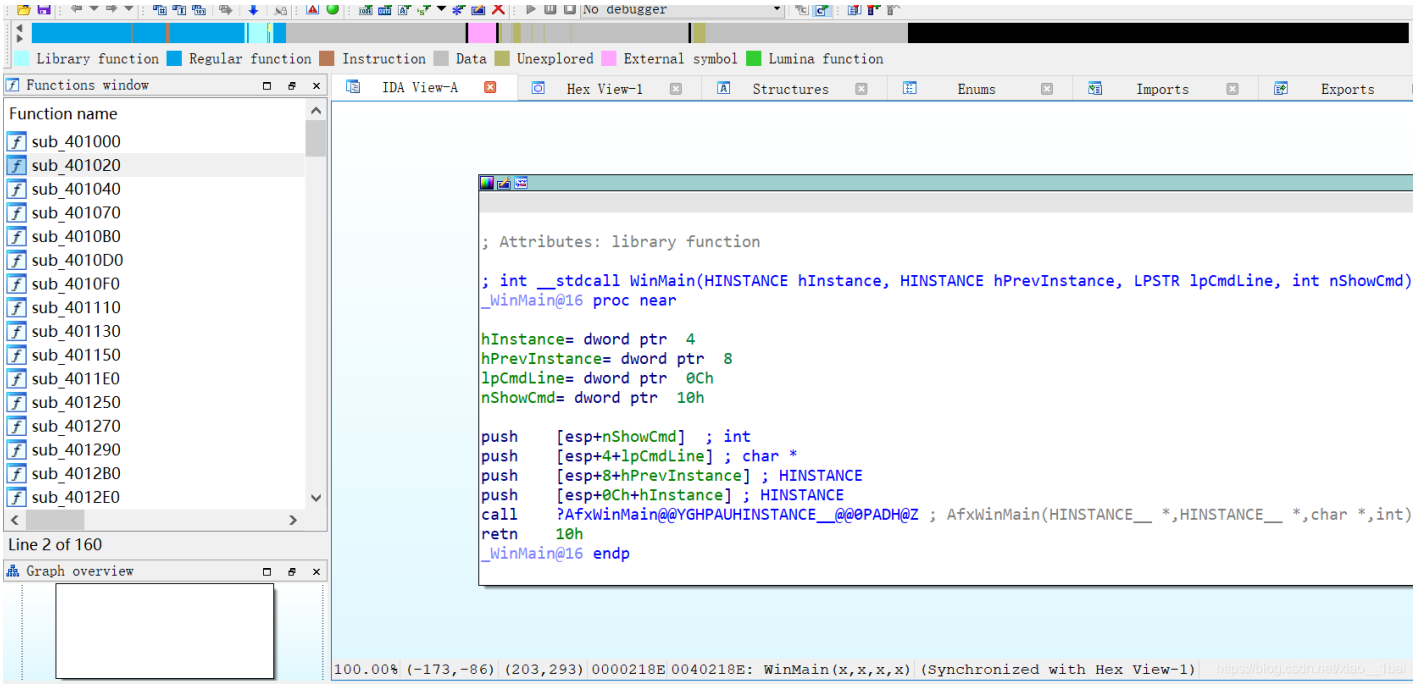
所以最后flag：

flag{123_Buff3r_0v3rf|0w}

至于那个crack按钮为什么按不下，可能考点不在那吧~

攻防世界流浪者：（多层交叉引用查看、函数逻辑封装、范围算法积累、函数积累）

32位PE文件无壳，照例扔入ida32中查看伪代码：



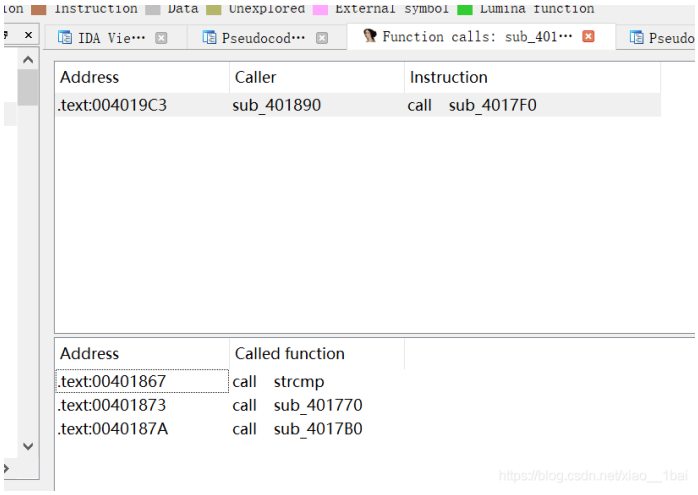
是没有主函数的题型，那就运行程序提取有用信息：

信息提取完了，一个弹框，一个判断，根据字符串我们可以找到弹框所在函数，弹框是messagebox函数，而import中只有一个messagebox，双击跟踪即可：

发现主要逻辑不在该函数处，查看该函数被谁调用，用IDA权威指南学到的新技巧，function call窗口：

```
1  int __cdecl sub_4017F0(int a1)
2  {
3    int result; // eax
4    char Str1[28]; // [esp+D8h] [ebp-24h] BYREF
5    int v3; // [esp+F4h] [ebp-8h]
6    int v4; // [esp+F8h] [ebp-4h]
7
8    v4 = 0;
9    v3 = 0;
10   while ( *(int *)(a1 + 4 * v4) < 62 && *(int *)(a1 + 4 * v4) >= 0 )
11   {
12     Str1[v4] = aAbcdefghiabcde[*(_DWORD *)(a1 + 4 * v4)];
13     ++v4;
14   }
15   Str1[v4] = 0;
16   if ( !strcmp(Str1, "KanXueCTF2019JustForhappy") )
17     result = sub_401770();
18   else
19     result = sub_4017B0();
20   return result;
21 }
```

找到比较函数，但是没有找到输入函数，继续查看被谁调用，继续function call窗口：



找到函数了，按照流程走，我们要判断是和输入有关的生成型flag还是简单的存储型flag，答案是前者，而且是明文密文对照类型，那就开始代码分析：

v4 = (CWnd *)((char *)this + 100);

v1 = CWnd::GetDlgItem(this, 1002); //这些系统函数一开始吓到我了，虽然系统函数一直不是什么考点，但还是认为这里会有和输入相关的东西，后来下载了API的chm文档查了一下作用，的确，后面的GetBuffer应该就是获取用户输入了，但只是获取而已，相当于scanf，知道即可，还是没什么考点

CWnd::GetWindowTextA(v1, v4);

v2 = sub_401A30((char *)v8 + 100);

Str = CString::GetBuffer((CWnd *)((char *)v8 + 100), v2);

if (!strlen(Str))

return CWnd::MessageBoxA(v8, &byte_4035DC, 0, 0); //弹框函数，这里犯下第一个错误，IDA双击进去的数据都是db类型的，而我们一开始看到的弹框显示的是中文，所以我们要改类型为dd类型才可以，不然就显示不了中文，所以&byte_4035DC跟踪进去后要用热键D改为dd类型再转字符。

for (i = 0; Str[i]; ++i) //把输入的字符串逐个判断条件并根据不同条件修改,

{

if (Str[i] > 57 || Str[i] < 48)

{

if (Str[i] > 122 || Str[i] < 97)

{

if (Str[i] > 90 || Str[i] < 65)

sub_4017B0(); //有一个不是弹框范围内就返回失败

else

v5[i] = Str[i] - 29; //范围在65~90中，输出结果在36~61中

}

else

{

v5[i] = Str[i] - 87; //范围在97~122中，输出结果在10~35中

}

}

else

{

v5[i] = Str[i] - 48; //范围在48~57中，输出结果在0~9中

}

}

return sub_4017F0((int)v5); //把修改后的数组结果作为参数赋给后面密文对照函数。

密文对照函数：

int __cdecl sub_4017F0(int a1)

{

int result; // eax

char Str1[28]; // [esp+D8h] [ebp-24h] BYREF

int v3; // [esp+F4h] [ebp-8h]

int v4; // [esp+F8h] [ebp-4h]

v4 = 0;

v3 = 0;

while (*(int *)(a1 + 4 * v4) < 62 && *(int *)(a1 + 4 * v4) >= 0)

{

Str1[v4] = aAbcdefghiabcde[*(_DWORD *)(a1 + 4 * v4)]; //aAbcdefghiabcde双击跟踪是abcdefghiABCDEFGHIJKLMNjklmn0123456789opqrstuvwxyzOPQRSTUVWXYZ'的62位长度字符串，也就不难解释while判断条件的<62了，这里还犯下第二，第三个错误。第二个错误是一开始没看出来这是个数组，*(_DWORD *)(a1 + 4 * v4)是取索引而已，a1是数组头地址，现在要知道带方括号的[]基本都是数组取字符！ 第三个错误是这里*(_DWORD *)(a1 + 4 * v4)型我是真的搞不懂，明明a1是int型，还是数组头地址，它前面的(_DWORD *)把它又变成了uint32型地址，先不说这多此一举，关键是int型地址+1就是加4个字节啊！这里直接+4*v4，那不是一下就跳过4个a1数组元素了吗！关键是我调试IDA既然没有问题！！！好吧，只能认为是IDA分析出错了，

++v4;

}

Str1[v4] = 0;

if (!strcmp(Str1, "KanXueCTF2019JustForhappy")) //从字典中获取的字符与明文对比，符合就是flag

result = sub_401770();

else

result = sub_4017B0();

return result;

}

所以现在就是字典和明文的加密关系逆向题了，这里犯下第四个错误，这类明文字典密文题目逆向要从密文出发，找到对应的字典下标，再用下标数组反逻辑逆向出明文：

第一步从密文出发，找到对应的字典下标：

key1="abcdefghiABCDEFGHIJKLMNjklmn0123456789opqrstuvwxyzOPQRSTUVWXYZ"

key2="KanXueCTF2019JustForhappy"

suoyin=[]

suoyin2=[]

v4=0

for i in range(len(key2)):

for a in range(len(key1)):

if key2[i] == key1[a]:

suoyin.append(a)

print(suoyin)

#print(len(suoyin))

这是我的做法，逐个对比，找出下标，当然后面还学到更好的.index(str)方法，后面会讲。输出：

[19, 0, 27, 59, 44, 4, 11, 55, 14, 30, 28, 29, 37, 18, 44, 42, 43, 14, 38, 41, 7, 0, 39, 39, 48]

这就是字典索引了，然后后面逆向出明文时就犯错了。逆向，是从底部出发向上走，也是是我们一开始掌握的是结果，要从条件中有关结果的判断往上走，而不是从0~1000这样从上往下加密然后提取出对应条件的下标，虽然结果一样，但是流程就差太多了：

suoyin2=[19, 0, 27, 59, 44, 4, 11, 55, 14, 30, 28, 29, 37, 18, 44, 42, 43, 14, 38, 41, 7, 0, 39, 39, 48]

v5=0

flag=""

for i in suoyin2: //这里的判断条件是从从条件中有关结果的判断往上走，因为前面写出了结果的范围，所以我们应该用结果的范围向上走。

if i >= 0 and i <= 9:

v5=i+48

elif i >= 10 and i <= 35:

v5=i+87

elif i >= 36:

v5=i+29

flag+=chr(v5)

print(flag)

输出：

j0rXl4bTeustBilGHeCF70DDM

别人更好的利用.index获取索引下标的脚本：

table = "abcdefghiABCDEFGHIJKLMNjklmn0123456789opqrstuvwxyzOPQRSTUVWXYZ"

s = "KanXueCTF2019JustForhappy"

ff = []

for i in s:

ff.append(table.index(i)) //这里我不得不说真的妙，我是一时想不到，用字符串内置函数.index(str)完美输出索引，比我快多了。

flag = ""

for i in ff:

if 0 <= i <= 9:

flag += chr(i + 48)

elif 9 < i <= 35:

flag += chr(i + 87)

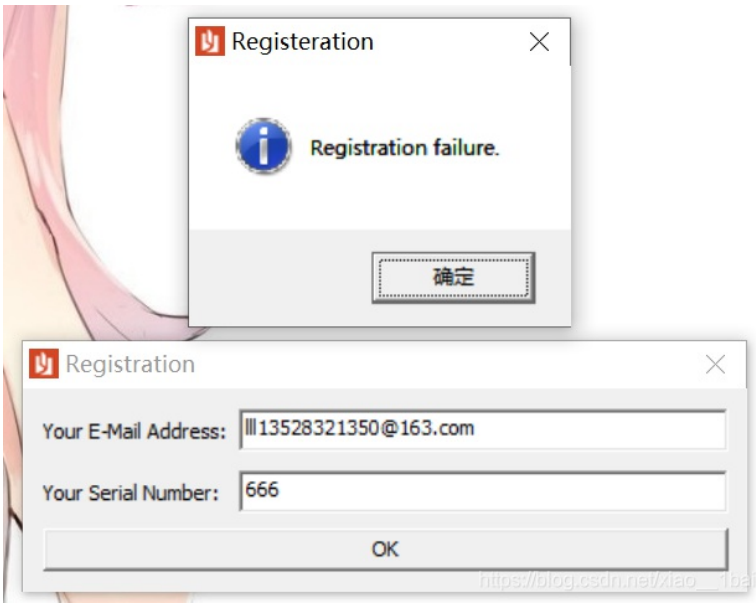
elif i > 36:

flag += chr(i + 29)

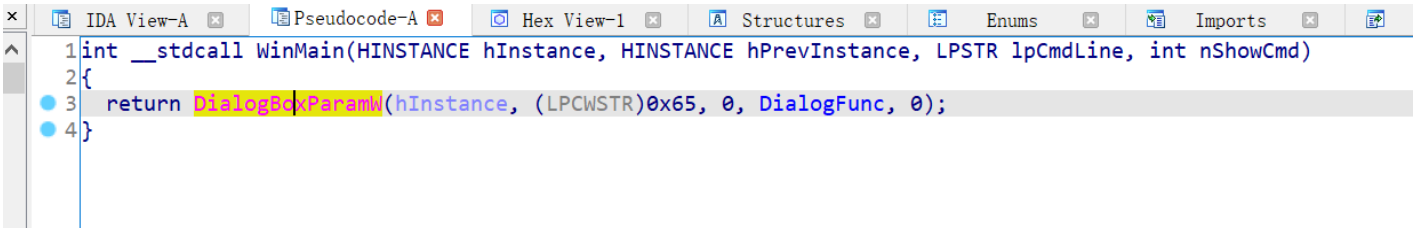
print (flag)

攻防世界srm-50：

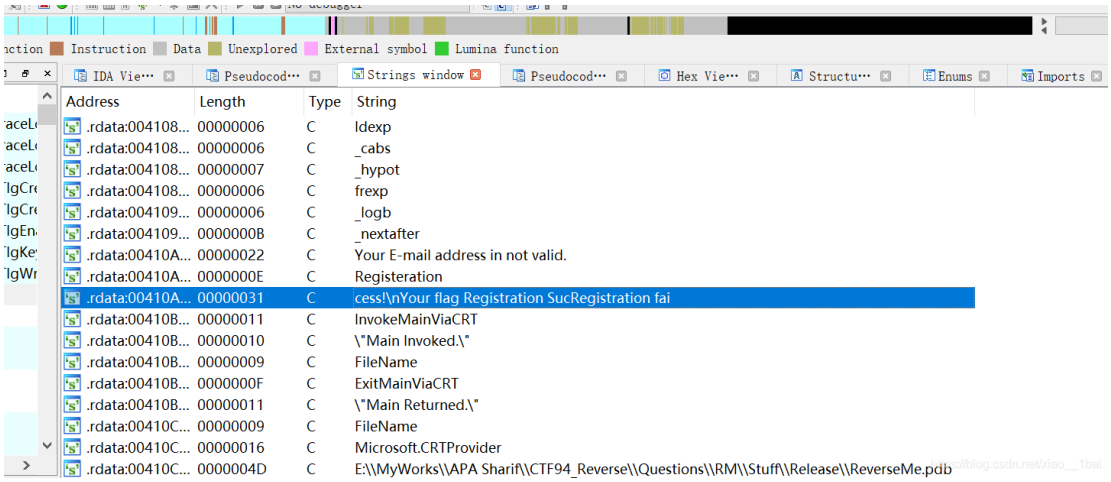
windows的32位程序，无壳，运行一下判断主要展示信息，看样子以为是逆向工程核心原理的例题，就是绕过注册条件的，结果后面发现不是：



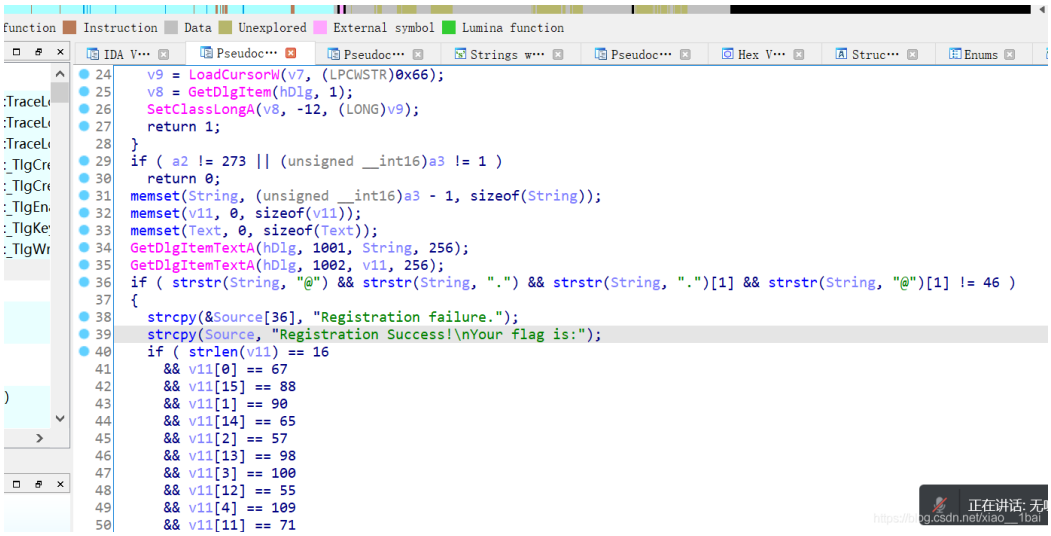
知道了一些特定字符串，和判断语句，信息够了，照例扔入IDA32中查看伪代码，有main函数看main函数，这里是Winmain函数：



跟踪，无果，照例下一步查看string窗口，锁定一开始展示字符串的位置：



找到了，双击跟踪，然后跟踪到引用它的函数：



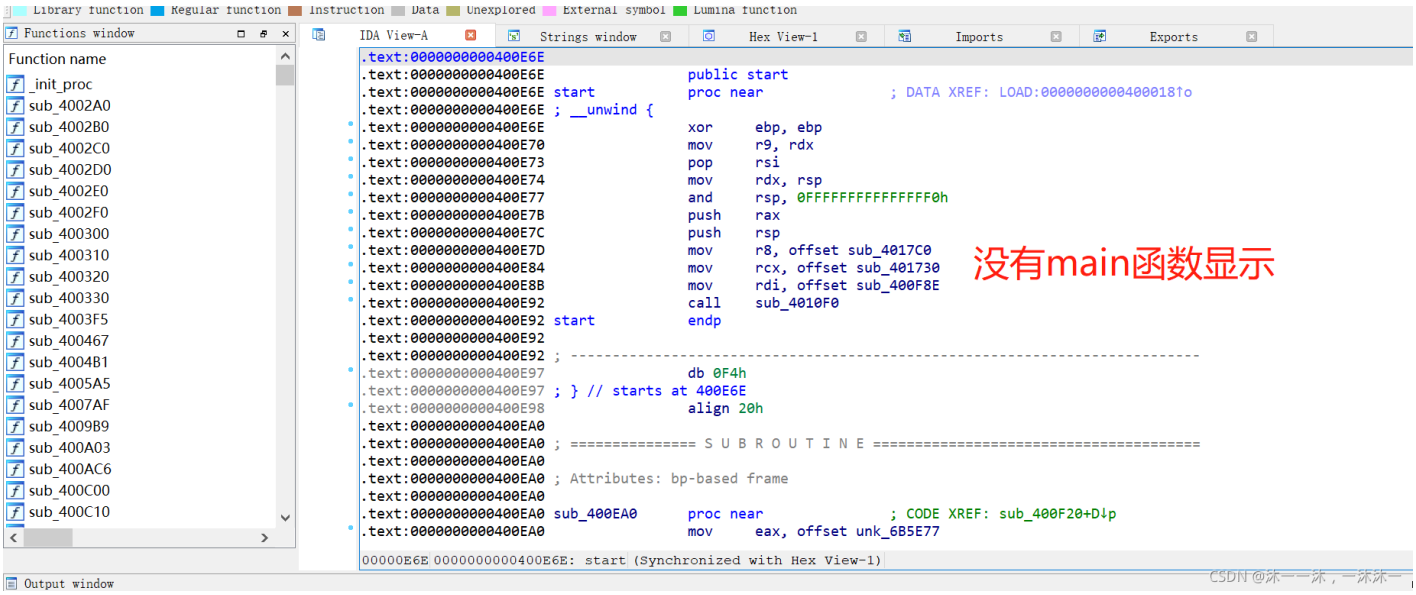
```
24    v9 = LoadCursorW(v7, (LPCWSTR)0x66);
25    v8 = GetDlgItem(hDlg, 1);
26    SetClassLongA(v8, -12, (LONG)v9);
27    return 1;
28  }
29  if ( a2 != 273 || (unsigned __int16)a3 != 1 )
30    return 0;
31  memset(String, (unsigned __int16)a3 - 1, sizeof(String));
32  memset(v11, 0, sizeof(v11));
33  memset(Text, 0, sizeof(Text));
34  GetDlgItemTextA(hDlg, 1001, String, 256);
35  GetDlgItemTextA(hDlg, 1002, v11, 256);
36  if ( strstr(String, "@") && strstr(String, ".") && strstr(String, ".")[1] && strstr(String, "@")[1] != 46 )
37  {
38    strcpy(&Source[36], "Registration failure.");
39    strcpy(Source, "Registration Success!\nYour flag is:");
40    if ( strlen(v11) == 16
41      && v11[0] == 67
42      && v11[15] == 88
43      && v11[1] == 90
44      && v11[14] == 65
45      && v11[2] == 57
46      && v11[13] == 98
47      && v11[3] == 100
48      && v11[12] == 55
49      && v11[4] == 109
50      && v11[11] == 71
```

答案很明显了，flag就是Registration Success!\nYour flag is:语句的下面一串字符，转字符后按序号排好即可：

```
38    strcpy(&Source[36], "Registration failure.");
39    strcpy(Source, "Registration Success!\nYour flag is:");
40    if ( strlen(v11) == 16
41      && v11[0] == 'C'
42      && v11[15] == 'X'
43      && v11[1] == 'Z'
44      && v11[14] == 'A'
45      && v11[2] == '9'
46      && v11[13] == 'b'
47      && v11[3] == 'd'
48      && v11[12] == '7'
49      && v11[4] == 'm'
50      && v11[11] == 'G'
51      && v11[5] == 'q'
52      && v11[10] == '9'
53      && v11[6] == '4'
54      && v11[9] == 'g'
55      && v11[7] == 'c'
56      && v11[8] == '8' )                    // CZ9dmq4c8g9G7bAX
57    {
58      strcpy_s(Text, 0x100u, Source);
```

攻防世界hackme：（可变参数混淆、随机抽取比较、取限制位数算法）

64位ELF文件无壳，照例扔入IDA64中查看伪代码信息，有main函数看main函数，结果没有main函数：




```
.text:0000000000400E6E
.text:0000000000400E6E                  public start
.text:0000000000400E6E start            proc near               ; DATA XREF: LOAD:0000000000400018↑o
.text:0000000000400E6E ; __unwind {
.text:0000000000400E6E                  xor     ebp, ebp
.text:0000000000400E70                  mov     r9, rdx
.text:0000000000400E73                  pop     rsi
.text:0000000000400E74                  mov     rdx, rsp
.text:0000000000400E77                  and     rsp, 0FFFFFFFFFFFFFFF0h
.text:0000000000400E7B                  push    rax
.text:0000000000400E7C                  push    rsp
.text:0000000000400E7D                  mov     r8, offset sub_4017C0
.text:0000000000400E84                  mov     rcx, offset sub_401730
.text:0000000000400E8B                  mov     rdi, offset sub_400F8E
.text:0000000000400E92                  call    sub_4010F0
.text:0000000000400E92 start            endp
.text:0000000000400E92
.text:0000000000400E92 ; ---------------------------------------------------------------------------
.text:0000000000400E97                  db 0F4h
.text:0000000000400E97 ; } // starts at 400E6E
.text:0000000000400E98                  align 20h
.text:0000000000400EA0
.text:0000000000400EA0 ; =============== S U B R O U T I N E =======================================
.text:0000000000400EA0
.text:0000000000400EA0 ; Attributes: bp-based frame
.text:0000000000400EA0
.text:0000000000400EA0 sub_400EA0       proc near               ; CODE XREF: sub_400F20+D↓p
.text:0000000000400EA0                  mov     eax, offset unk_6B5E77
```

没有main函数就运行程序收集显示信息：

找到两个字符串，直接在strings窗口双击跟踪，找到主要逻辑函数：

这里积累第一个经验：如上图红框所示

sub_407470((unsigned int)“Give me the password: “, a2, a3, a4, a5, a6, a2);

sub_4075A0((unsigned int)”%s”, (unsigned int)v16, v6, v7, v8, v9, v14);

人傻了，这么多个参数。双击跟踪进去不是嵌套就是复杂代码，查看反汇编，好像又没有调用这么多参数。以为是什么复杂的高端函数，开始怕了。结果才发现，这就是根据C语言函数可变参数的特性反汇编出来的，其实就是普通的输出和输入函数而已。果然还是自己经验太少，太菜了~。

```
.text:0000000000400F8F                mov     rbp, rsp
.text:0000000000400F92                sub     rsp, 0C0h
.text:0000000000400F99                mov     [rbp+var_B4], edi
.text:0000000000400F9F                mov     qword ptr [rbp+var_C0], rsi
.text:0000000000400FA6                mov     edi, offset aGiveMeThePassw ; "Give me the password: "
.text:0000000000400FAB                mov     eax, 0
.text:0000000000400FB0                call    sub_407470
.text:0000000000400FB5                lea     rax, [rbp+var_B0]
.text:0000000000400FBC                mov     rsi, rax
.text:0000000000400FBF                mov     edi, offset aS  ; "%s"
.text:0000000000400FC4                mov     eax, 0
.text:0000000000400FC9                call    sub_4075A0
.text:0000000000400FCE                mov     [rbp+var_4], 0
```

汇编语言中显示没有传入那么多参数，这只是简单输入输出函数而已。

跨过这个坎继续往下走，第一个红框v10范围是0~21，第二个红框数组又以v10作为下标，猜测是一个22的遍历数组操作，第三个红框是判断，所以逆向逻辑很简单。关键是最外面的v25的10次循环后面查了资料说是v10是一个随机数，范围也的确是0 ~ 21，但是不是顺序来取值的，结合外面v25的10循环就是随机从数组中抽取10个下标来比较，那我们直接顺序取整个下标操作也是一样的。

```
29  v26 = i == 22;
30  v25 = 10;
31  do
32  {
33    v10 = (int)sub_406D90() % 22;
34    v22 = v10;
35    v24 = 0;
36    v21 = byte_6B4270[v10];
37    v20 = v16[v10];
38    v19 = v10 + 1;
39    v23 = 0;
40    while ( v23 < v19 )
41    {
42      ++v23;
43      v24 = 1828812941 * v24 + 12345;
44    }
45    v18 = v24 ^ v20;
46    if ( v21 != ((unsigned __int8)v24 ^ v20) )
47      v26 = 0;
48    --v25;
49  }
50  while ( v25 );
51  if ( v26 )
52    v17 = sub_407470((unsigned int)"Congras\n", (unsigned int)v16, v24, v10, v11, v12, v15);
```

取0~21的随机数

v21是取byte_6B4270数组的v10随机成员

v20也是取v16数组的v10随机成员

v24是以随机成员v10为循环的一个实数

最后以v10随机数做基准取出的各个数进行比较，其实就是在相同的字符串中取随机但同样的位来比较，所以逆向是要顺序取。

根据前面回顾的逆向解题流程：

第一步确定Flag字符数量。

第二步找到已确定的字符串作为基点来反推falg字符。

第三步找出逻辑中与flag直接相关的部分，该部分可以正向爆破或者从尾到头的反向逻辑。

然后找到与flag没有直接关联的部分，该部分无需逆向逻辑，直接正向流程复现即可。

按照流程来即可写出逆向逻辑脚本：

key1=[95, 242, 94, 139, 78, 14, 163, 170, 199, 147, 129, 61, 95, 116, 163, 9, 145, 43, 73, 40, 147, 103]

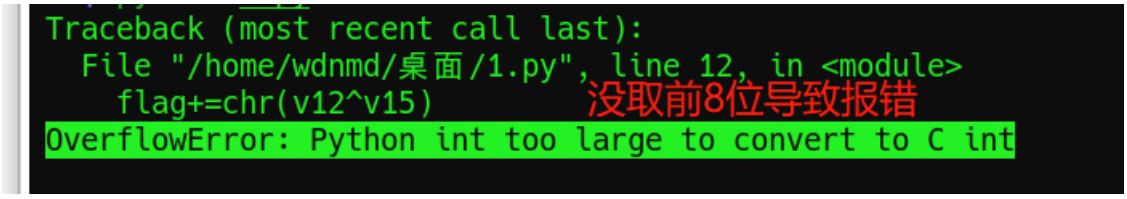
flag=""

for i in range(10):

```
for a in range(22):

v15=0

v12=key1[a]

v10=a+1

v14=0

for i in range(v10):

v14+=1

v15=1828812941*v15+12345

flag+=chr((v12^v15)&0xff)

print(flag)
```

最后这里积累第二个经验：

一开始我flag+=chr((v12^v15)&0xff)没加 &0xff，然后报错，我以为源程序中会有溢出导致的数重置，但是想起程序是64位的，不应该超范围啊。然后我看到源代码有__int8这个限制，这是取前8位啊，可是python中怎么取前8位呢？查了资料才发现有&0xff这种方法，因为&在Python中是逻辑与运算，所以与的时候就保留了v12 ^ v15的前8位，就达到取前8位的目的了，取前16，32位都可以套用这个方法。



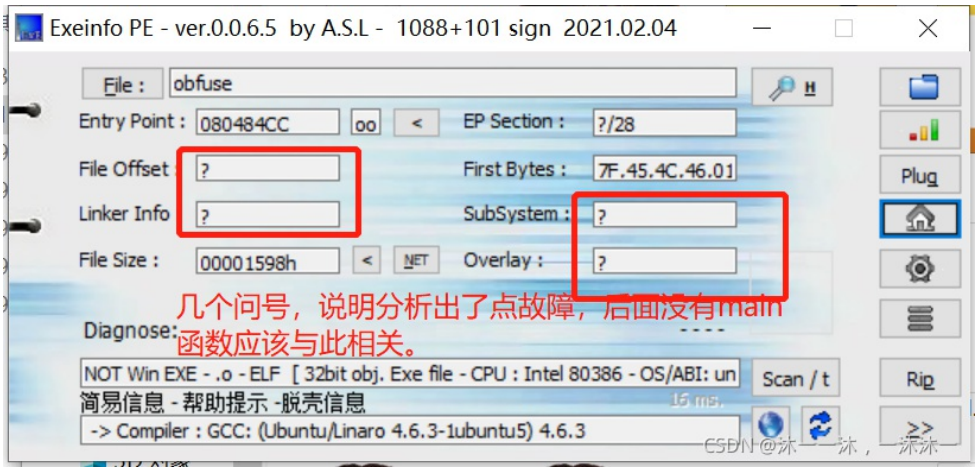
结果：(这里我一开始没理解是随机抽取10次，所以我照搬，结果循环了10次)

```
flag{d826e6926098ef46}flag{d826e6926098ef46}flag{d826e6926098ef46}flag{d826e6926098ef46}fl
ag{d826e6926098ef46}flag{d826e6926098ef46}flag{d826e6926098ef46}flag{d826e6926098ef46}flag
{d826e6926098ef46}flag{d826e6926098ef46}
```

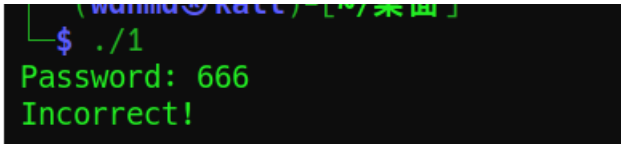
无伪代码分析汇编：

攻防世界之76号：（F7和F8交叉使用、函数逻辑封装、寄存器传参、switch正向代入推导）

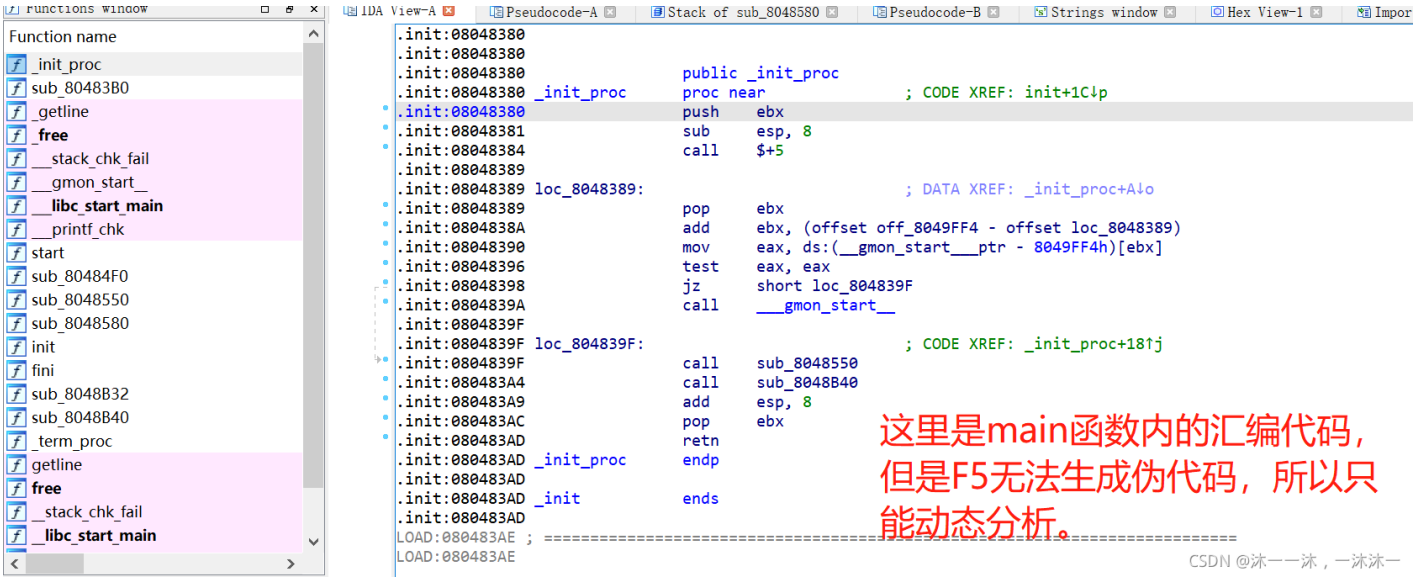
下载附件，解压，照例扔入exeinfope中查看信息，这里是32位ELF文件，无壳，这里几个?号可能就对应了IDA中没有main函数的原因把：



照例先运行一下程序，看一下主要回显信息：

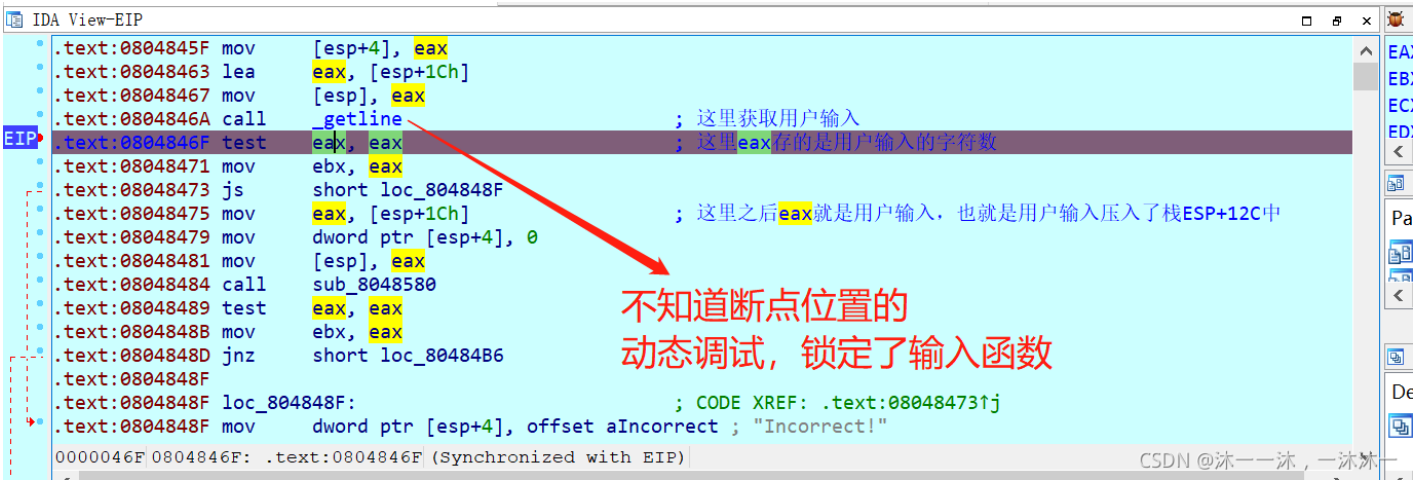


照例扔入IDA32中查看伪代码信息，没有main函数，在strings窗口跟踪一下password，然后就跳转到汇编代码中了，一开始没发现是main函数，按F5也生成不了伪代码，然后我就想到动调了。



(这里积累第一个经验)

只有汇编代码的动态调试中，我参考的是前面做过的serial-150，运行程序，因为不知道在哪里下断点，所以在输入的时候按暂停来下断，然后单步执行到从7F地址跳出到真实的40地址为止。

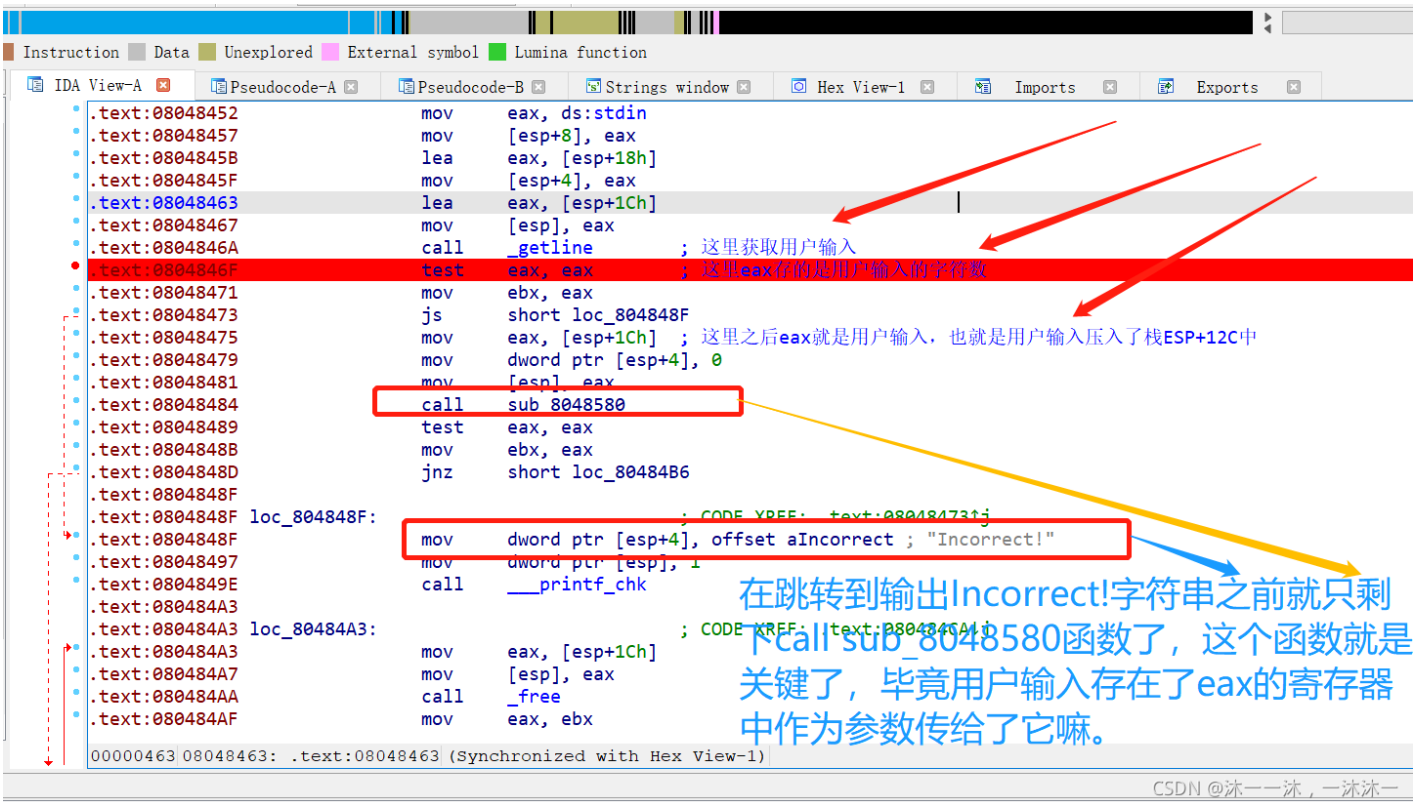


(这里积累第二个经验)

的确弹出到真实的执行地址了，因为一开始不知道这里是main函数的范围~。。。。额，现在知道了，所以可以直接在这里分析了。

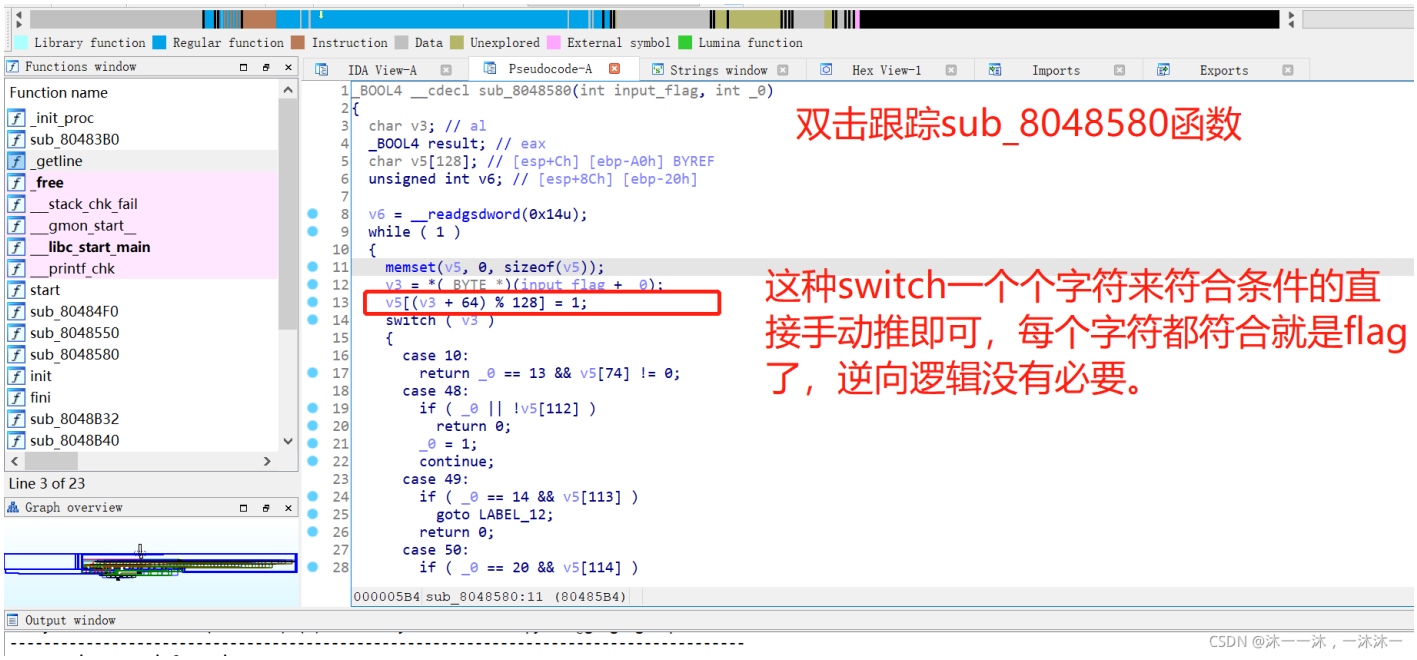
函数作用注释都写好了，_getline是获取用户输入，然后js short loc_804848F也没有跳转。eax什么的在动态中发现前后赋值为用户输入的字符数和用户输入的字符串。

然后在跳转到输出Incorrect!字符串之前就只剩下call sub_8048580函数了，这个函数就是关键了，毕竟用户输入存在了eax的寄存器中作为参数传给了它嘛。



(这里积累第三个经验)

双击跟踪sub_8048580函数，根据前面传入的参数重命名了一下形参，函数逻辑很简单，就是不断switch判断，其中_0形参是在判断中自己修改的，没有递增。



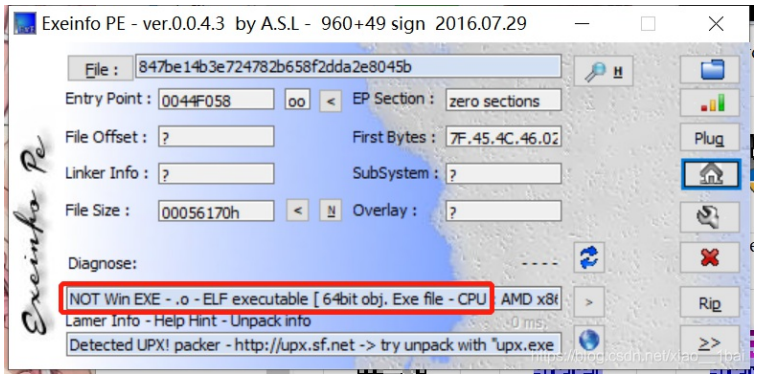
只能手动一个个去推了，注意v5的下标要对应，因为有一两个混淆项，最终flag：

flag{09vdf7wefijbk}

带壳题目类型

工具直接脱壳类型：

攻防世界simple-unpack脱壳：（工具脱壳）



显示说探测到UPX壳，由于第一次做带壳的题目，所以查到了以下资料：

UPX (the Ultimate Packer for eXecutables)是一款先进的可执行程序文件压缩器，压缩过的可执行文件体积缩小50%-70%，这样减少了磁盘占用空间、网络上传下载的时间和其它分布以及存储费用。 通过 UPX 压缩过的程序和程序库完全没有功能损失和压缩之前一样可正常地运行，对于支持的大多数格式没有运行时间或内存的不利后果。 UPX 支持许多不同的可执行文件格式 包含 Windows 95/98/ME/NT/2000/XP/CE 程序和动态链接库、DOS 程序、 Linux 可执行文件和核心。

UPX是一个压缩工具，好在今天准备看《逆向核心工程原理》这本书的压缩部分，原来这就是压缩，之前也学了一点PE文件格式，知道了一些文件资源的存放位置，那么下一步就是脱壳了。

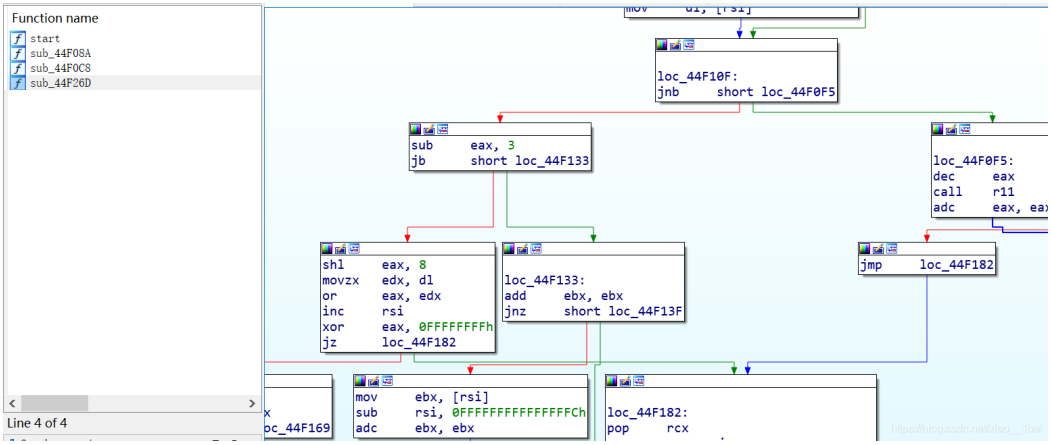
查到了kali中关于UPX的脱壳命令：

upx -d filename

脱完壳就可以直接IDA查看了，FLAG直接就显示出来了：

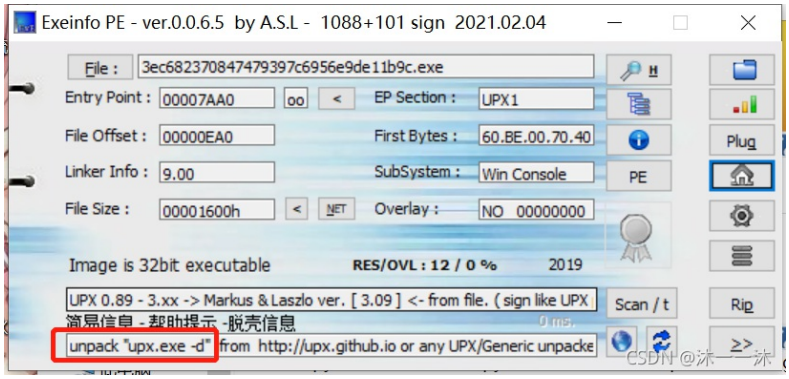
```
main proc near

s1= byte ptr -70h
var_8= qword ptr -8

; __unwind {
push    rbp
mov     rbp, rsp
sub     rsp, 70h
mov     rax, fs:28h
mov     [rbp+var_8], rax
xor     eax, eax
lea     rax, [rbp+s1]
mov     rsi, rax
mov     edi, offset a96s ; "%96s"
mov     eax, 0
call    __isoc99_scanf
lea     rax, [rbp+s1]
mov     esi, offset flag ; "flag{Upx_1s_n0t_a_d3liv3r_c0mp4ny}"
mov     rdi, rax        ; s1
call    _strcmp
test    eax, eax
jnz     short loc_4009FC
```

IDA等二进制分析器应该都需要完整的文件格式才能分析，压缩后(加壳)的文件由于并没有破坏可执行文件的格式规则，所以还是可以运行的，用IDA分析加壳后的文件就分析不出来了，如图：

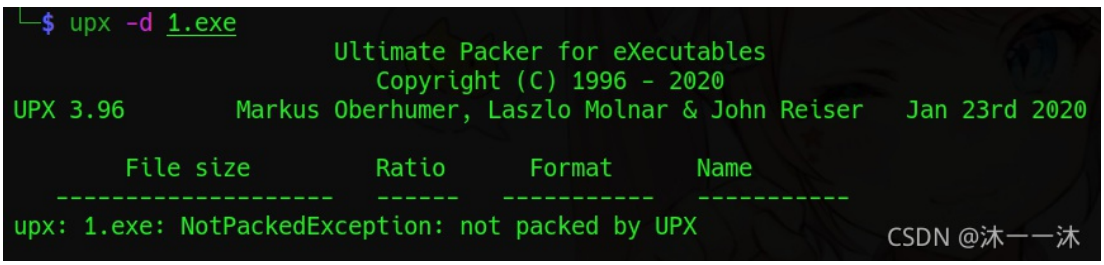


攻防世界Windows_Reverse1：（工具脱壳、不能直接运行、寄存器传参、地址差值+数组组合遍历字符串、字符ASCII码做索引、ASCII码表相关）

下载附件，照例扔入exeinfope中查看信息：



UPX壳，32位windows中，扔入我的kali中先用命令upx -d 文件名 脱壳先：



双击运行不了，查看不了起始信息，看了资料说：

UPX的壳，手动脱壳或者脱壳机脱壳，但发现脱完壳的程序在win7下打不开，即使是显示没壳（这里后来查到win7包括以上版本开启了ASLR(地址随机化)，winxp就没有，如果程序采用绝对地址，在win7和win10上就运行不了），直接IDA启动，IDA里不爆红就没事。

不管了,IDA分析伪代码：

```
1  int __cdecl main(int argc, const char **argv, const char **envp)
2  {
3    char v4; // [esp+4h] [ebp-804h] BYREF
4    char v5[1023]; // [esp+5h] [ebp-803h] BYREF
5    char input_flag; // [esp+404h] [ebp-404h] BYREF
6    char v7[1023]; // [esp+405h] [ebp-403h] BYREF
7
8    input_flag = 0;
9    memset(v7, 0, sizeof(v7));
10   v4 = 0;
11   memset(v5, 0, sizeof(v5));
12   printf("please input code:");
13   scanf("%s", &input_flag);
14   sub_401000(&input_flag);
15   if ( !strcmp(&v4, "DDCTF{reverseME}") )
16     printf("You've got it!!%s\n", &v4);
17   else
18     printf("Try again later.\n");
19   return 0;
20 }
```

自定义函数出入的是input_flag，但是比较的却是v4，这里设计寄存器传参操作。

代码一目了然，神奇的是红框框起来的地方。这里积累第一个经验：出入的是input_flag（v6改名而来），结果比较的是v4，一开始我以为是IDA出了错误，后来才发现题目考的就是我以前一直说得地址偏移间接操作。v4存入了寄存器中，寄存器再作为参数传入关键自定义函数中，IDA没有反汇编出寄存器参数，用的是寄存器操作。

input_flag点进去-00000404 var_404 db ?

v4点进去-00000804 var_804 db ?

这里积累第二个经验：

查看反汇编代码，前面有sub esp, 804h，所以esp+804h处可以说是基址EBP的地方，这里的[esp+82Ch+input_flag]和[esp+830h+v4]只是在esp+804h的EBP基础上加上中间代码的指令字节长度而已，本质就是取input_flag和v4参数。

```
.text:004010B7                 lea     eax, [esp+82Ch+input_flag]
.text:004010BE                 push    eax
.text:004010BF                 lea     ecx, [esp+830h+v4]
.text:004010C3                 call    sub_401000
.text:004010C8                 add     esp, 28h
```

flag和v4分别赋值给了eax和ecx

这里Input_flag和v4分别给了eax和ecx，我们查看sub_401000函数的反汇编代码和函数图都可以发现ECX（v4）被使用了：

```
.text:00401000 ; unsigned int __cdecl sub_401000(const char *input_flag)
.text:00401000 sub_401000      proc near               ; CODE XREF: _main+73↓p
.text:00401000
.text:00401000 input_flag      = dword ptr  4
.text:00401000
.text:00401000                 push    ecx
.text:00401001                 push    ebp
.text:00401002                 mov     ebp, [esp+8+input_flag]
.text:00401006                 push    esi
.text:00401007                 mov     eax, ebp
```

v4被使用了

所以我们分析逻辑代码：（地址差值+数组组合遍历字符串，单个字符ASCII码作为索引）

```
unsigned int __cdecl sub_401000(const char *input_flag)
{
_BYTE *v1; // ecx即外部v4，这里用v1来接受寄存器ecx的值，且值为0
unsigned int v2; // edi
unsigned int result; // eax
int v4; // ebx
v2 = 0;
result = strlen(input_flag);
if ( result )
{
v4 = input_flag - v1; //这里积累第三个经验：地址减地址取差值。这里v4是Input_flag和v1（主函数v4）的地址的差值，差值刚好在32，后面梳理完后发现byte_402FF8是ASCII码表，32后是可打印字符
do
{
*v1 = byte_402FF8[(char)v1[v4]]; //这里积累第四个经验：这里V1作为地址和v4作为数组v1[v4]执行的是v1+v4的操作，就是v4+v1=input_flag啊。因为数组a[b]本质就是在数组头地址a加上偏移量b来遍历数组的，所以这里是一种遍历input_flag的新操作，至于最外面的byte_402FF8[]数组框，应该这样理解，v1[v4]逐个取input_flag的单个字符，这个字符的ascii码作为偏移继续在byte_402FF8[]数组中寻址。(PS：这不是Python中list.index()函数可以用字符查找对应索引！)，最后ECX寄存器的v1接受了新的flag。
++v2;
++v1;
result = strlen(input_flag);
}
while ( v2 < result );
}
return result;
}
```

也附上别人博客的解释：（本是字符本身作为索引）

1：a1是通过压栈的方式传递的参数; v1是通过寄存器保存地址的方式传递的参数。

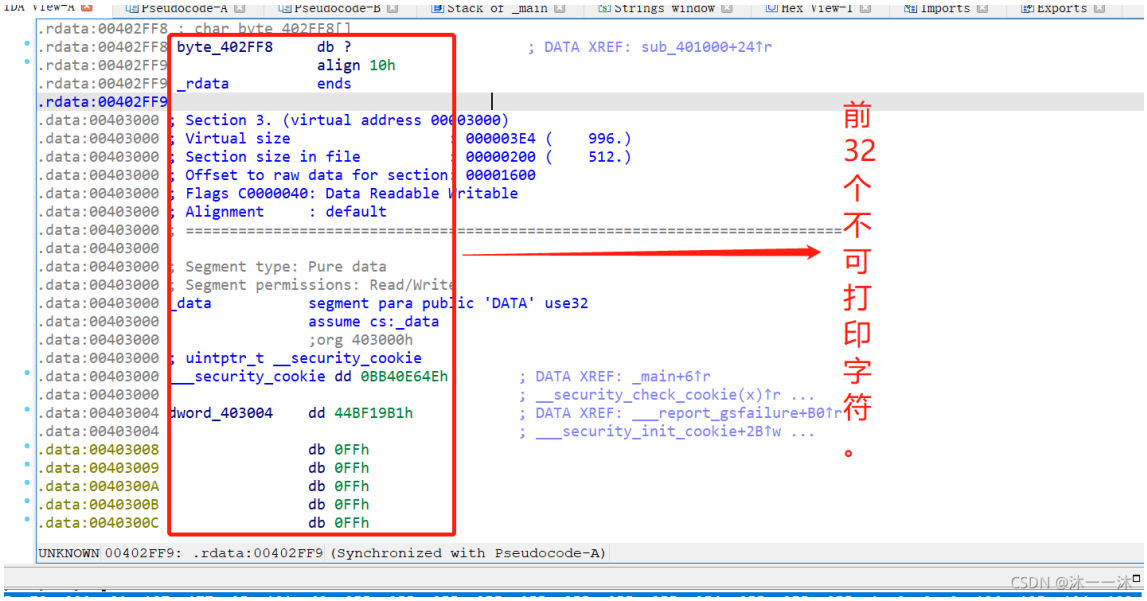
2：最令人迷惑的便是v1[v4]这个地方. v1是一个地址, v4是a1和v1两个地址间的差值. 地址的差值是怎么成为一个数组的索引的呢？

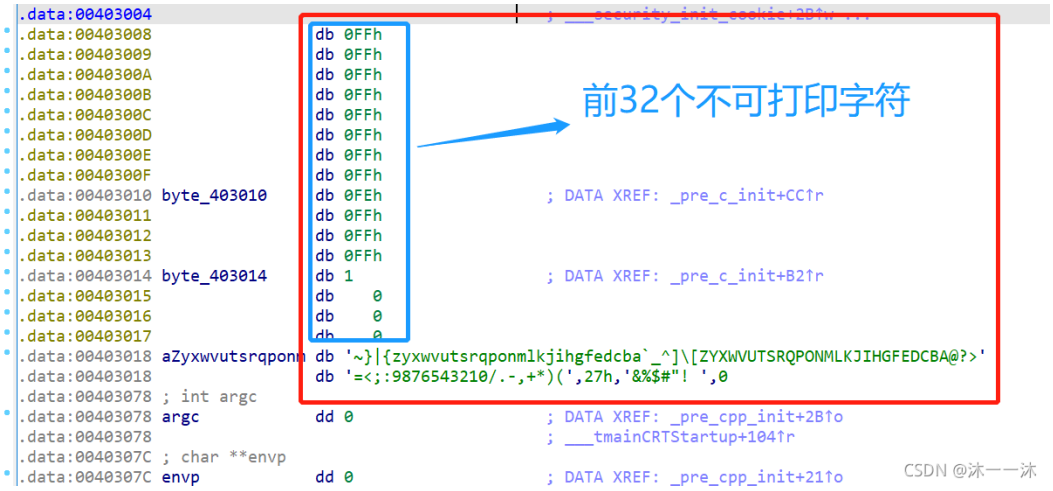
3：这里卡了我好长时间, 之后我突然意识到, v1[v4]和v1+v4是等价的, 而在循环刚开始的时候v1+v4等于a1, 随着v1的递增,v1[v4]也会遍历a1数组中的各个元素的地址。

4：而地址又怎么能作为数组的索引呢? 这里就是 IDA 背锅了, 换言之, 做题还是不能太依赖于反编译后的伪代码. 查看了反汇编代码后, 发现其实是将a1字符串中的字符本身作为byte_402FF8的索引, 取值后放入v1数组中。

双击跟踪byte_402FF8[]数组：

可以看到前面乱码的?，后面倒是有字符串，数组地址偏移从0x00402FF8~0x00403078。

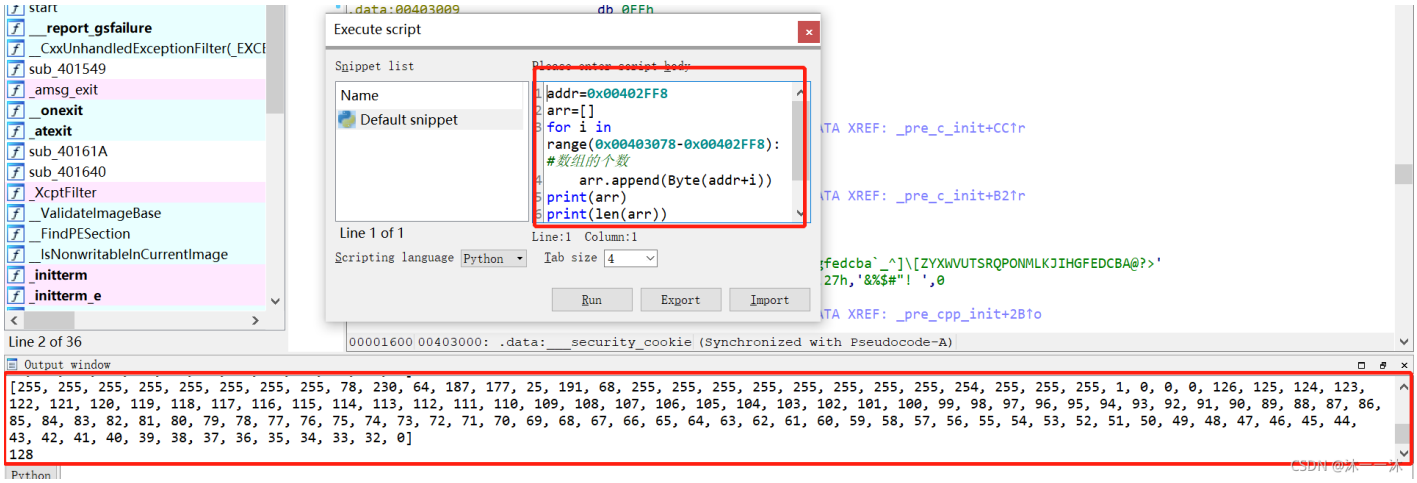




这里积累第5个经验：（ASCII码表可视字符范围）

ASCII编码表里的可视字符就得是32往后了, 所以, byte_402FF8里凡是位于32以前的数统统都是迷惑项. 不会被索引到的，而这里0x00402FF8~0x00403017刚好是32个字符。那么后面有字符串就可以解释通了，它们是连在一起的。

数组有了，逻辑有了，逆向逻辑很简单，先用IDA脚本打印0x00402FF8~0x00403078数组处的地址内容先：



复制数组内容写逆向脚本：

key1=[255, 255, 255, 255, 255, 255, 255, 255, 78, 230, 64, 187, 177, 25, 191, 68, 255, 255, 255, 255, 255, 255, 255, 255, 254, 255, 255, 255, 1, 0, 0, 0, 126, 125, 124, 123, 122, 121, 120, 119, 118, 117, 116, 115, 114, 113, 112, 111, 110, 109, 108, 107, 106, 105, 104, 103, 102, 101, 100, 99, 98, 97, 96, 95, 94, 93, 92, 91, 90, 89, 88, 87, 86, 85, 84, 83, 82, 81, 80, 79, 78, 77, 76, 75, 74, 73, 72, 71, 70, 69, 68, 67, 66, 65, 64, 63, 62, 61, 60, 59, 58, 57, 56, 55, 54, 53, 52, 51, 50, 49, 48, 47, 46, 45, 44, 43, 42, 41, 40, 39, 38, 37, 36, 35, 34, 33, 32, 0]

key2="DDCTF{reverseME}"

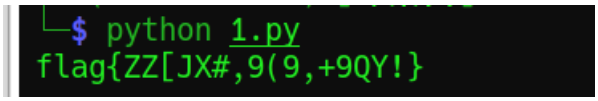
flag=""

for i in key2:

flag+=chr(key1[ord(i)]) //字符的ASCII码作为索引

print("flag{"+flag+"}")

结果：



攻防世界Replace：（工具脱壳、解题逆向模板、>> 和 % 运算符算法积累、正向爆破）

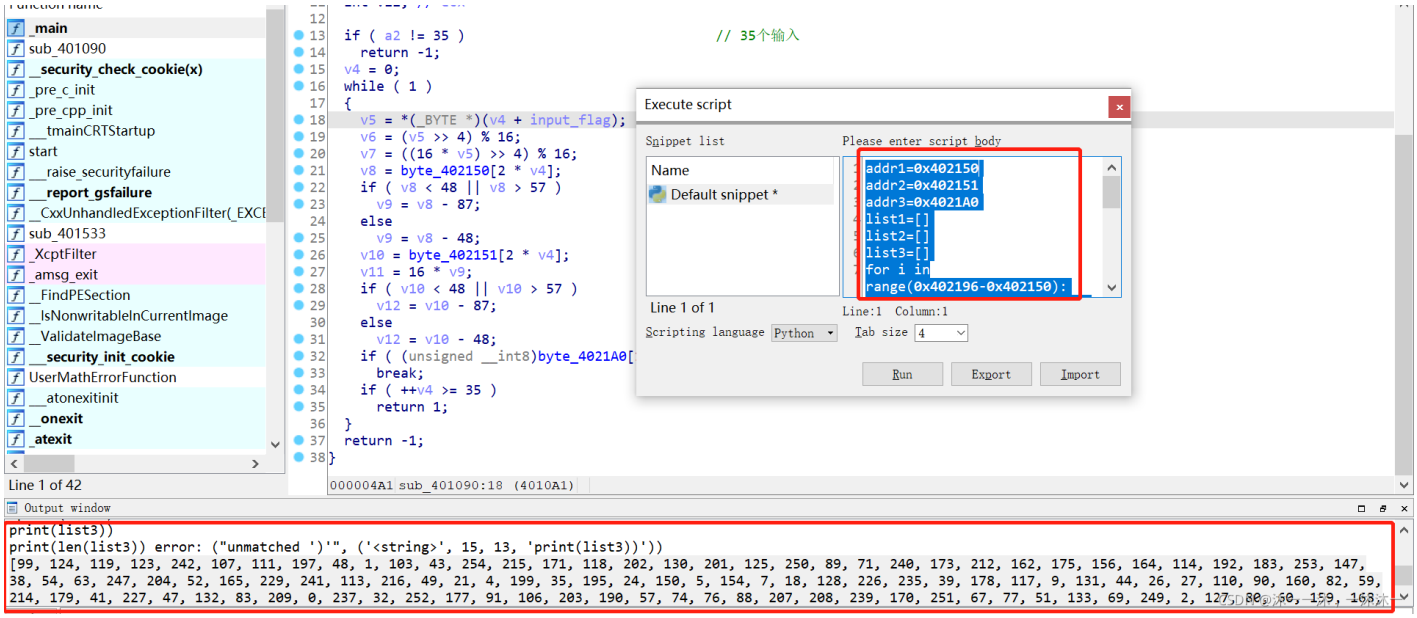
有壳，用Kali的upx -d脱壳，然后照例扔入IDA32中查看伪代码，有main函数看main函数：(脱壳后不能运行，因为伪代码信息足够，所以就不用修复了)

直接跟踪第二个红框自定义函数：

```
v4 = 0;
while ( 1 )
{
  v5 = *(_BYTE *)(v4 + input_flag);           // 取一个字符
  v6 = (v5 >> 4) % 16;
  v7 = ((16 * v5) >> 4) % 16;
  v8 = byte_402150[2 * v4];
  if ( v8 < 48 || v8 > 57 )
    v9 = v8 - 87;
  else
    v9 = v8 - 48;
  v10 = byte_402151[2 * v4];
  v11 = 16 * v9;
  if ( v10 < 48 || v10 > 57 )
    v12 = v10 - 87;
  else
    v12 = v10 - 48;
  if ( (unsigned __int8)byte_4021A0[16 * v6 + v7] != ((v11 + v12) ^ 25) )
    break;
  if ( ++v4 >= 35 )
    return 1;
}
return -1;
}
```

操作中涉及三个数组

主要逻辑在上面了，涉及三个数组，用IDA内嵌脚本dump下来，这里积累第一个经验：数组dump下载的时候dump到0字符结尾处，不要怕dump多，就怕dump少。



开始编写逆向逻辑脚本，这里积累第二个经验：主要回顾一下每一步的思路，给日后自己增添一些解题模板。

第一步确定Flag字符数量，第一个红框处得到flag数量是35。

第二步找到已确定的比较字符串作为基点来反推flag字符，如第二个红框处。

```
12
13  if ( a2 != 35 )                    flag数量          // 35个输入
14    return -1;
15  v4 = 0;
16  while ( 1 )
17  {
18    v5 = *(_BYTE *)(v4 + input_flag);                  // 取一个字符
19    v6 = (v5 >> 4) % 16;
20    v7 = ((16 * v5) >> 4) % 16;
21    v8 = byte_402150[2 * v4];
22    if ( v8 < 48 || v8 > 57 )
23      v9 = v8 - 87;
24    else
25      v9 = v8 - 48;
26    v10 = byte_402151[2 * v4];
27    v11 = 16 * v9;
28    if ( v10 < 48 || v10 > 57 )
29      v12 = v10 - 87;
30    else                                  已确定的比较字符串
31      v12 = v10 - 48;
32    if ( (unsigned __int8)byte_4021A0[16 * v6 + v7] != ((v11 + v12) ^ 25) )
33      break;
34    if ( ++v4 >= 35 )
35      return 1;
36  }
37  return -1;
38}
```


第三步找出逻辑中与flag直接相关的部分，该部分可以正向爆破或者从尾到头的反向逻辑，如第一个红框所示。然后找到与flag没有直接关联的部分，该部分无需逆向逻辑，直接正向流程复现即可，如第二个红框所示。

```
16  while ( 1 )
17  {
18    v5 = *(_BYTE *)(v4 + input_flag);          // 取一个字符
19    v6 = (v5 >> 4) % 16;
20    v7 = ((16 * v5) >> 4) % 16;
21    v8 = byte_402150[2 * v4];
22    if ( v8 < 48 || v8 > 57 )
23      v9 = v8 - 87;
24    else
25      v9 = v8 - 48;
26    v10 = byte_402151[2 * v4];
27    v11 = 16 * v9;
28    if ( v10 < 48 || v10 > 57 )
29      v12 = v10 - 87;
30    else
31      v12 = v10 - 48;
32    if ( (unsigned __int8)byte_4021A0[16 * v6 + v7] != ((v11 + v12) ^ 25) )
33      break;
34    if ( ++v4 >= 35 )
35      return 1;
36  }
37  return -1;
38}
```

与输入相关的逆向逻辑或正向爆破

与输入无关的正向流程复现。

梳理完这些之后就可以写脚本了：

脚本1，爆破。这里积累第三个经验：由于用了取余 % 运算，所以采用枚举正向爆破的方法，让flag中的每一个字符遍历常用的字符（ascii码表中32-126），带入加密算法，如果成功，就把这个flag存入。

list1=[50, 97, 52, 57, 102, 54, 57, 99, 51, 56, 51, 57, 53, 99, 100, 101, 57, 54, 100, 54, 100, 101, 57, 54, 100, 54, 102, 52, 101, 48, 50, 53, 52, 56, 52, 57, 53, 52, 100, 54, 49, 57, 53, 52, 52, 56, 100, 101, 102, 54, 101, 50, 100, 97, 100, 54, 55, 55, 56, 54, 101, 50, 49, 100, 53, 97, 100, 97, 101, 54]

list2=[97, 52, 57, 102, 54, 57, 99, 51, 56, 51, 57, 53, 99, 100, 101, 57, 54, 100, 54, 100, 101, 57, 54, 100, 54, 102, 52, 101, 48, 50, 53, 52, 56, 52, 57, 53, 52, 100, 54, 49, 57, 53, 52, 52, 56, 100, 101, 102, 54, 101, 50, 100, 97, 100, 54, 55, 55, 56, 54, 101, 50, 49, 100, 53, 97, 100, 97, 101, 54]

list3=[99, 124, 119, 123, 242, 107, 111, 197, 48, 1, 103, 43, 254, 215, 171, 118, 202, 130, 201, 125, 250, 89, 71, 240, 173, 212, 162, 175, 156, 164, 114, 192, 183, 253, 147, 38, 54, 63, 247, 204, 52, 165, 229, 241, 113, 216, 49, 21, 4, 199, 35, 195, 24, 150, 5, 154, 7, 18, 128, 226, 235, 39, 178, 117, 9, 131, 44, 26, 27, 110, 90, 160, 82, 59, 214, 179, 41, 227, 47, 132, 83, 209, 0, 237, 32, 252, 177, 91, 106, 203, 190, 57, 74, 76, 88, 207, 208, 239, 170, 251, 67, 77, 51, 133, 69, 249, 2, 127, 80, 60, 159, 168, 81, 163, 64, 143, 146, 157, 56, 245, 188, 182, 218, 33, 16, 255, 243, 210, 205, 12, 19, 236, 95, 151, 68, 23, 196, 167, 126, 61, 100, 93, 25, 115, 96, 129, 79, 220, 34, 42, 144, 136, 70, 238, 184, 20, 222, 94, 11, 219, 224, 50, 58, 10, 73, 6, 36, 92, 194, 211, 172, 98, 145, 149, 228, 121, 231, 200, 55, 109, 141, 213, 78, 169, 108, 86, 244, 234, 101, 122, 174, 8, 186, 120, 37, 46, 28, 166, 180, 198, 232, 221, 116, 31, 75, 189, 139, 138, 112, 62, 181, 102, 72, 3, 246, 14, 97, 53, 87, 185, 134, 193, 29, 158, 225, 248, 152, 17, 105, 217, 142, 148, 155, 30, 135, 233, 206, 85, 40, 223, 140, 161, 137, 13, 191, 230, 66, 104, 65, 153, 45, 15, 176, 84, 187, 22, 72, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

v4=0

flag=""

for i in range(35):

v8=list1[2*i]

if v8 < 48 or v8 > 57:

v9=v8 - 87

else:

v9=v8 - 48

v10=list2[2*i]

v11=16*v9

if v10 <48 or v10 > 57:

v12=v10-87

else:

v12=v10-48

v4=((v11+v12)^25) #这里前面都是与flag没有直接关联的正向流程复现

for a in range(32,127): #取余类型正向爆破，如果符合比较，就获取对应字符。

v5=a

v6=(v5>>4)%16 #逻辑都是一样的，因为是正向爆破

v7=((16*v5)>>4)%16

if list3[16*v6+v7] == v4:

flag+=chr(a)

print(flag)

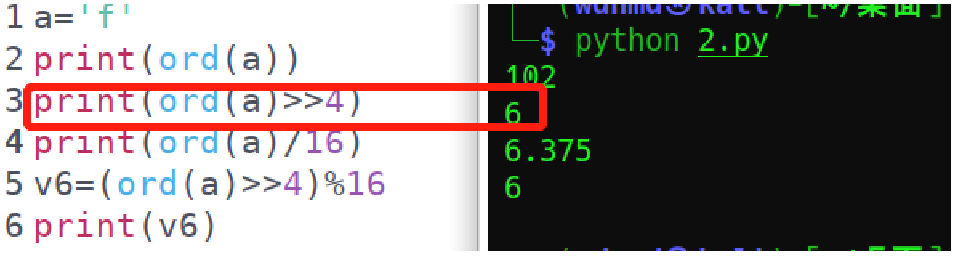
脚本2：这里积累第四个经验：>> 运算符其实是不带余数的除法 / 算法，单取整数部分。% 运算符其实是不带整数的求余运算，单取余数部分。

所以这里 v6 = (v5 >> 4) % 16 是除以16后的整数部分， v7 = ((16 * v5) >> 4) % 16 是乘16后除16再取16内的余数，也就是直接取16内的余数。一个取整数，一个取余数，所以他们的逆向算法就是16 * v6 + v7，就是最后数组byte_4021A0[16 * v6 + v7]的下标。

(既然v5 >> 4相当于v5除以16取整数部分，是不带余数的除法 / 算法。顺带说一下，v5 & 0xf 相当于v5除以16取余数部分，是完全的求余%算法，这里0xf是4位所以为16。)

所以flag就是由这些下标组成！

因此如果有些伪代码中前面的运算逻辑如果和后面的运算逻辑如果有相似之处，要试着判断他们是反向逻辑，因为出题者的意图往往如此。



list1=[50, 97, 52, 57, 102, 54, 57, 99, 51, 56, 51, 57, 53, 99, 100, 101, 57, 54, 100, 54, 100, 101, 57, 54, 100, 54, 102, 52, 101, 48, 50, 53, 52, 56, 52, 57, 53, 52, 100, 54, 49, 57, 53, 52, 52, 56, 100, 101, 102, 54, 101, 50, 100, 97, 100, 54, 55, 55, 56, 54, 101, 50, 49, 100, 53, 97, 100, 97, 101, 54]

list2=[97, 52, 57, 102, 54, 57, 99, 51, 56, 51, 57, 53, 99, 100, 101, 57, 54, 100, 54, 100, 101, 57, 54, 100, 54, 102, 52, 101, 48, 50, 53, 52, 56, 52, 57, 53, 52, 100, 54, 49, 57, 53, 52, 52, 56, 100, 101, 102, 54, 101, 50, 100, 97, 100, 54, 55, 55, 56, 54, 101, 50, 49, 100, 53, 97, 100, 97, 101, 54]

list3=[99, 124, 119, 123, 242, 107, 111, 197, 48, 1, 103, 43, 254, 215, 171, 118, 202, 130, 201, 125, 250, 89, 71, 240, 173, 212, 162, 175, 156, 164, 114, 192, 183, 253, 147, 38, 54, 63, 247, 204, 52, 165, 229, 241, 113, 216, 49, 21, 4, 199, 35, 195, 24, 150, 5, 154, 7, 18, 128, 226, 235, 39, 178, 117, 9, 131, 44, 26, 27, 110, 90, 160, 82, 59, 214, 179, 41, 227, 47, 132, 83, 209, 0, 237, 32, 252, 177, 91, 106, 203, 190, 57, 74, 76, 88, 207, 208, 239, 170, 251, 67, 77, 51, 133, 69, 249, 2, 127, 80, 60, 159, 168, 81, 163, 64, 143, 146, 157, 56, 245, 188, 182, 218, 33, 16, 255, 243, 210, 205, 12, 19, 236, 95, 151, 68, 23, 196, 167, 126, 61, 100, 93, 25, 115, 96, 129, 79, 220, 34, 42, 144, 136, 70, 238, 184, 20, 222, 94, 11, 219, 224, 50, 58, 10, 73, 6, 36, 92, 194, 211, 172, 98, 145, 149, 228, 121, 231, 200, 55, 109, 141, 213, 78, 169, 108, 86, 244, 234, 101, 122, 174, 8, 186, 120, 37, 46, 28, 166, 180, 198, 232, 221, 116, 31, 75, 189, 139, 138, 112, 62, 181, 102, 72, 3, 246, 14, 97, 53, 87, 185, 134, 193, 29, 158, 225, 248, 152, 17, 105, 217, 142, 148, 155, 30, 135, 233, 206, 85, 40, 223, 140, 161, 137, 13, 191, 230, 66, 104, 65, 153, 45, 15, 176, 84, 187, 22, 72, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

v4=0

flag=""

for i in range(35):

v8=list1[2*i]

if v8 < 48 or v8 > 57:

v9=v8 - 87

else:

v9=v8 - 48

v10=list2[2*i]

v11=16*v9

if v10 <48 or v10 > 57:

v12=v10-87

else:

v12=v10-48

v4=((v11+v12)^25) #这里前面都是与flag没有直接关联的正向流程复现

flag+=chr(list3.index(v4)) #这里知道逆向算法发现对应下标后，直接Index对要比较的字符v4取索引即可，flag就是索引中下标的内容。

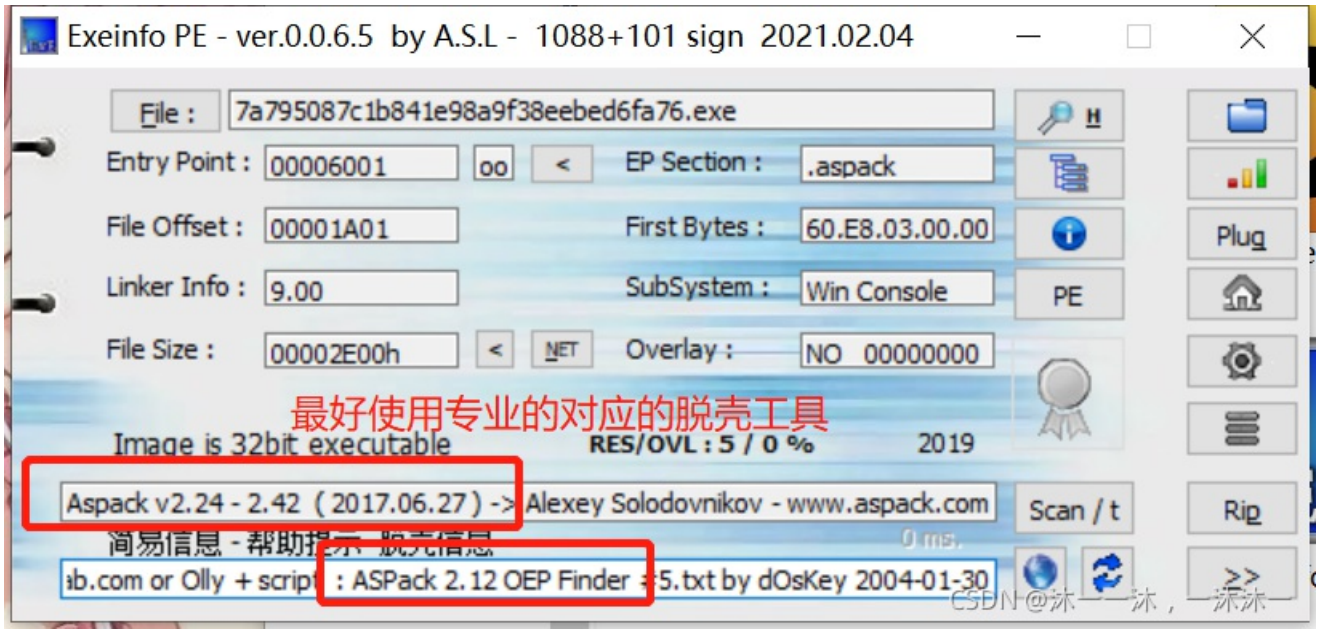
print(flag)

结果：

```
└─$ python 1.py
flag{Th1s_1s_Simple_Rep1ac3_Enc0d3}
```

攻防世界Windows_Reverse2：（专业脱壳工具、大小写字符转换算法、16进制转10进制算法、遍历字符加1算法积累、同地址变量、base64加密/解密算法、多层加密操作）

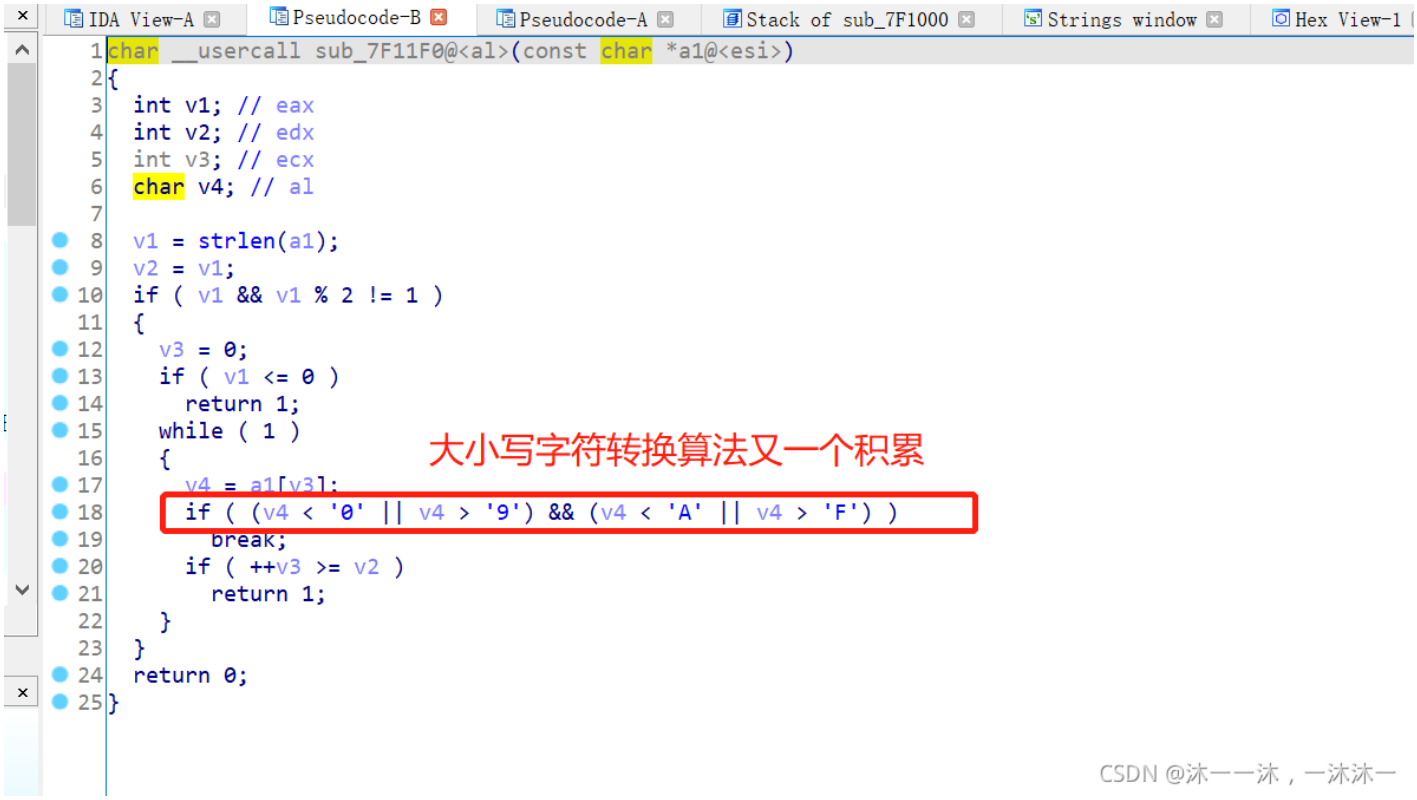
下载附件，照例扔入exeinfope中查看信息，发现ASPACK壳，也提示了使用ASPACK unpack脱壳工具脱壳：



用WASPACK脱壳工具脱壳后，运行不了，，，扔入IDA32中查看伪代码信息，有main函数看main函数：

(这里积累第一个经验)

可以发现三个关键部分，大概是三层逻辑修改。双击跟踪sub_7F11F0()函数，内容逻辑就是输入的数要在0~ 9 、 A~F之间，就是输入大写十六进制了，后面会有跟16进制相关的加密算法，大写是为了限制flag为大写，免得提交了小写。



(这里积累第二个经验)

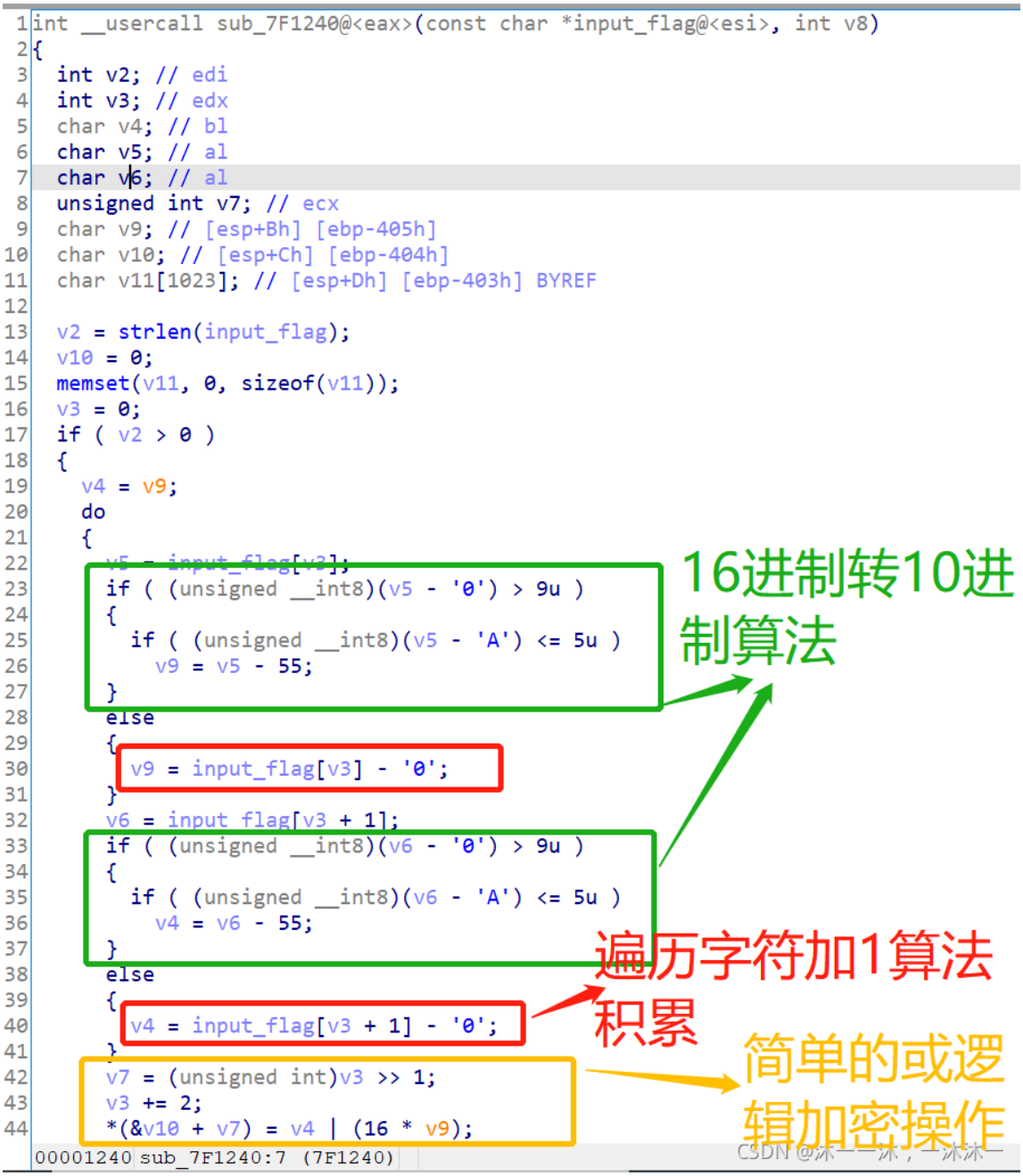
跟踪sub_7F1240(&v6, (int)&v8)函数，因为是自定义函数，所以有必要跟踪。

绿色的框中限定范围A~F，减去55之后就是变成了实数的10 ~15。红框中限定范围0 ~9减去字符'0'之后就变成了实数0 ~9。

所以就是0 ~ F分别对应实数的0 ~15之所以减得不同是因为ASCII表中0 ~ 9和A ~ F并不相连。

黄框中因为v3+=2，前面积累过 右移运算符 >> 是不带余数的除法，所以这里v7每次递增1。然后最后的*(&v10 + v7) = v4 | (16 * v9);就是把处理后的值赋值给v10地址处，异或的是后面逆向逻辑的时候好像考虑不到这段代码，而是直接用0~ F对0~15的转换来做的。

(后来感觉是用来对应base64基本字符数组的下标的。)

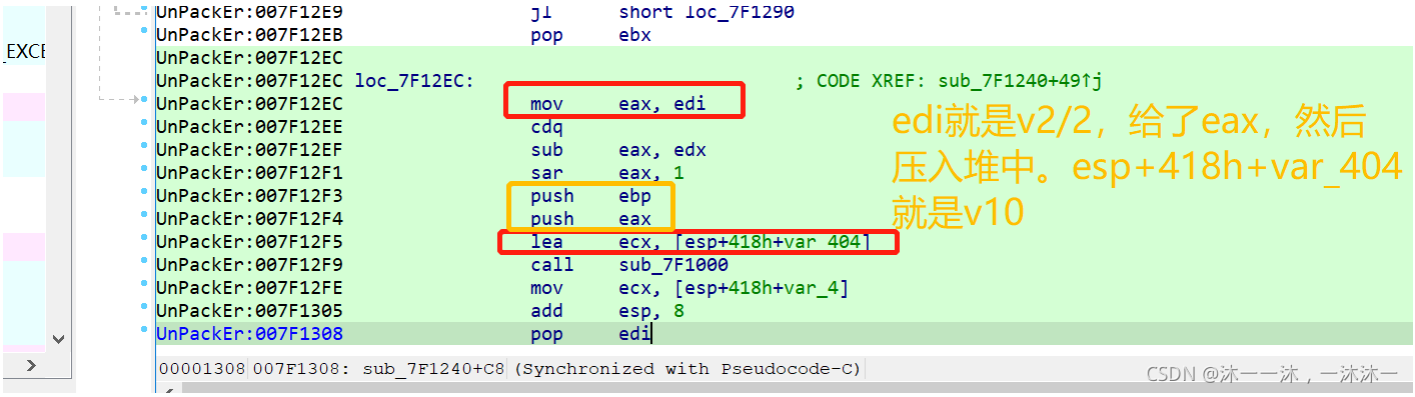



```
int __usercall sub_7F1240@<eax>(const char *input_flag@<esi>, int v8)
{
  int v2; // edi
  int v3; // edx
  char v4; // bl
  char v5; // al
  char v6; // al
  unsigned int v7; // ecx
  char v9; // [esp+Bh] [ebp-405h]
  char v10; // [esp+Ch] [ebp-404h]
  char v11[1023]; // [esp+Dh] [ebp-403h] BYREF

  v2 = strlen(input_flag);
  v10 = 0;
  memset(v11, 0, sizeof(v11));
  v3 = 0;
  if ( v2 > 0 )
  {
    v4 = v9;
    do
    {
      v5 = input_flag[v3];
      if ( (unsigned __int8)(v5 - '0') > 9u )
      {
        if ( (unsigned __int8)(v5 - 'A') <= 5u )
          v9 = v5 - 55;
      }
      else
      {
        v9 = input_flag[v3] - '0';
      }
      v6 = input_flag[v3 + 1];
      if ( (unsigned __int8)(v6 - '0') > 9u )
      {
        if ( (unsigned __int8)(v6 - 'A') <= 5u )
          v4 = v6 - 55;
      }
      else
      {
        v4 = input_flag[v3 + 1] - '0';
      }
      v7 = (unsigned int)v3 >> 1;
      v3 += 2;
      *(&v10 + v7) = v4 | (16 * v9);
```

(这里积累第三个经验)

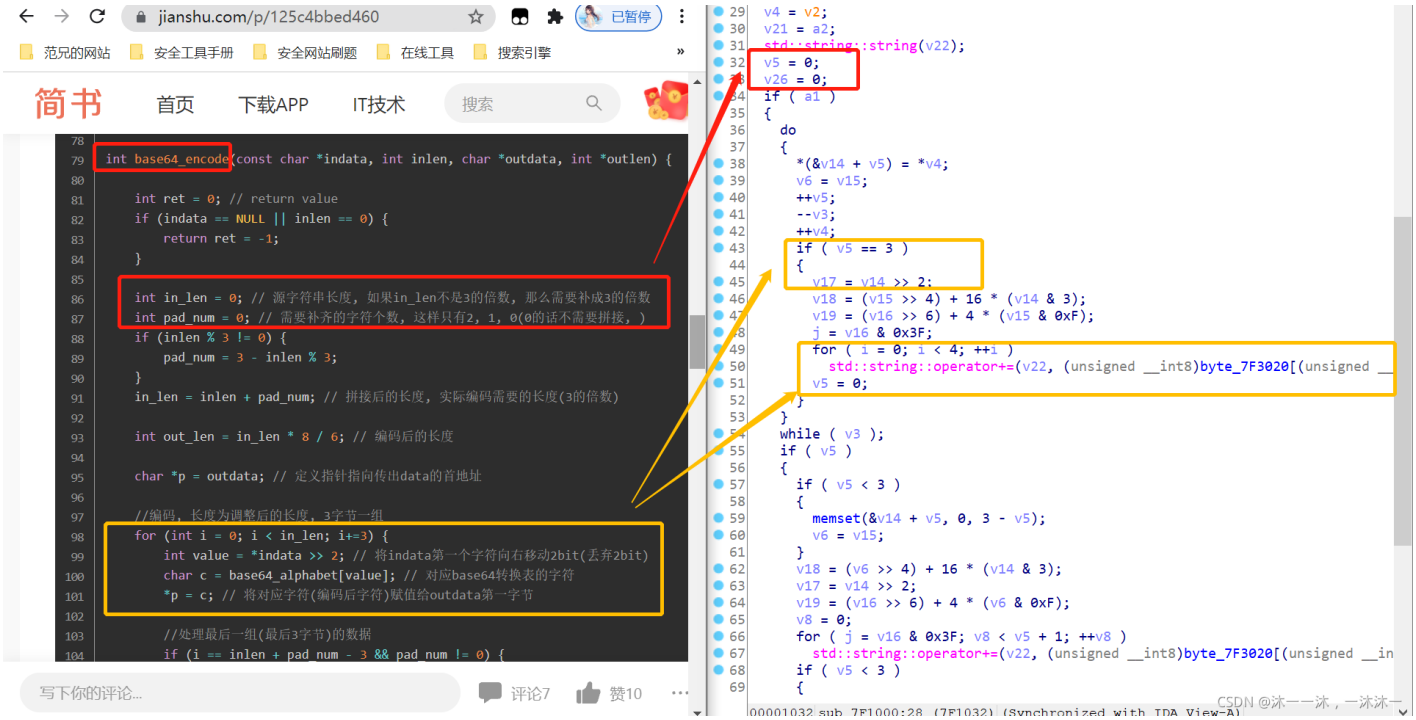
然后最神奇的就是最后面这个自定义函数，传入了v2/2，就是输入的flag长度的一半。然后再传入v8，关键是上面的变换都赋值给了v10地址处啊，如果在大量冗余代码中的确只看对输入flag直接操作的字符串，可是这里逻辑很简单，那么就是v10其实和v8是同一个地址！！！！

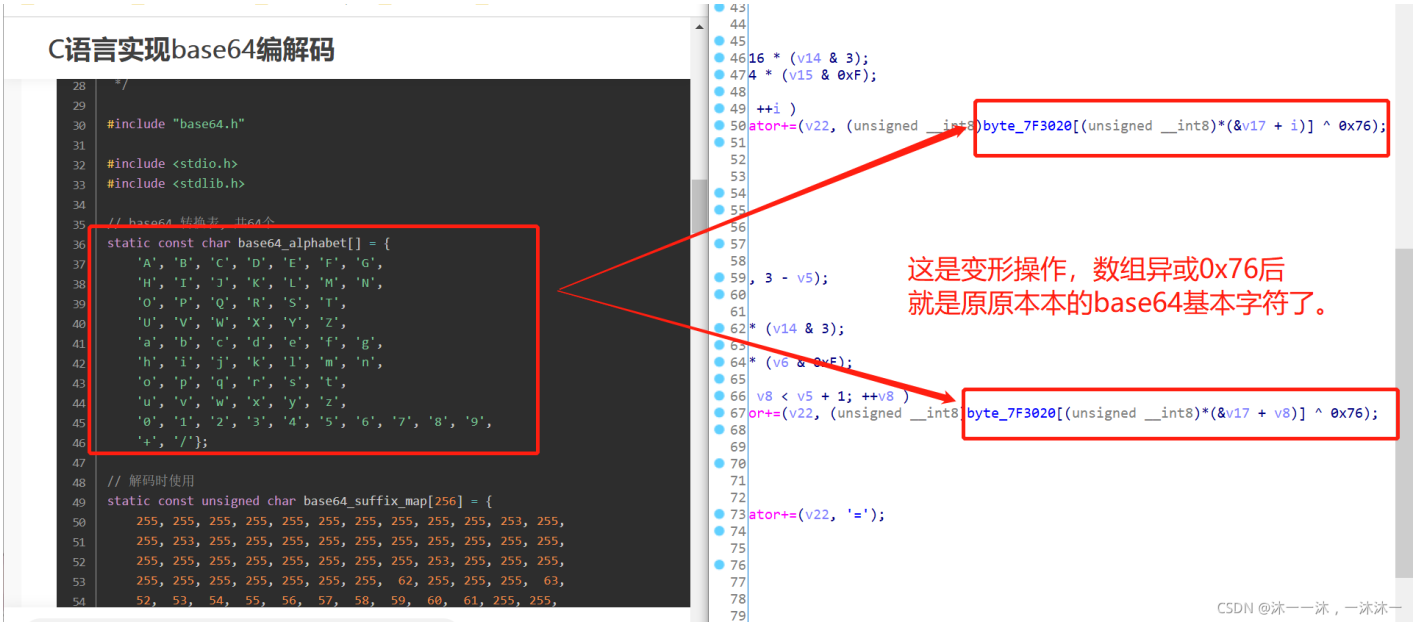
各自双击v8和v10查看堆栈就会发现真的是同一个地址！！！(本来我一开始看得IDA的确是同一个地址，可是现在不是了，看了反汇编代码倒是直接把v10作为参数压入堆中了。)



（这里积累第四个经验）

然后继续跟踪sub_7F1000(v2 / 2, (void *)v8)函数，一开始不知道，后来梳理了整个流程之后发现很多都是变量名不同，但是地址重叠的，很带干扰性。只能大体分析了。





(这里积累第5个经验)

附上官方的代码分析，这里也掌握一下C++一些函数的作用，如std::allocator和std::allocator<char>>::operator+=函数的作用：

std::basic_string<char,std::char_traits<char>,std::allocator<char>>::basic_string<char,std::char_traits<char>,st(&v22);//C++的构造函数

len = 0;

v26 = 0;

if (half_length)

{

do

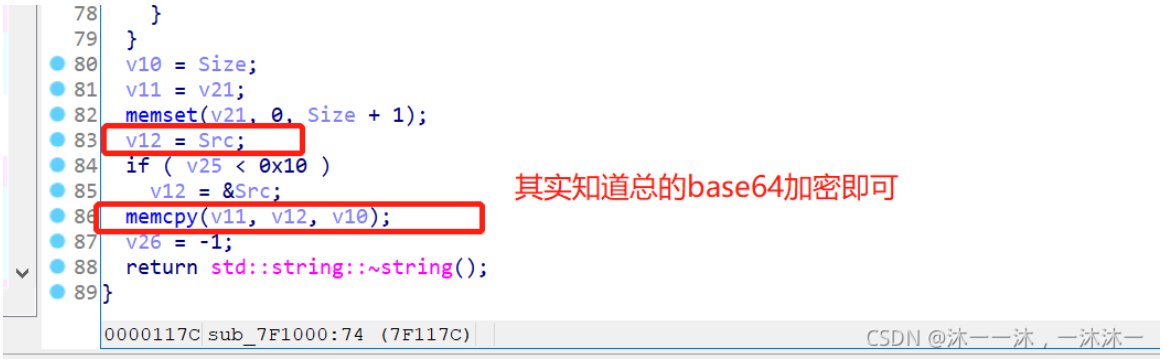
{

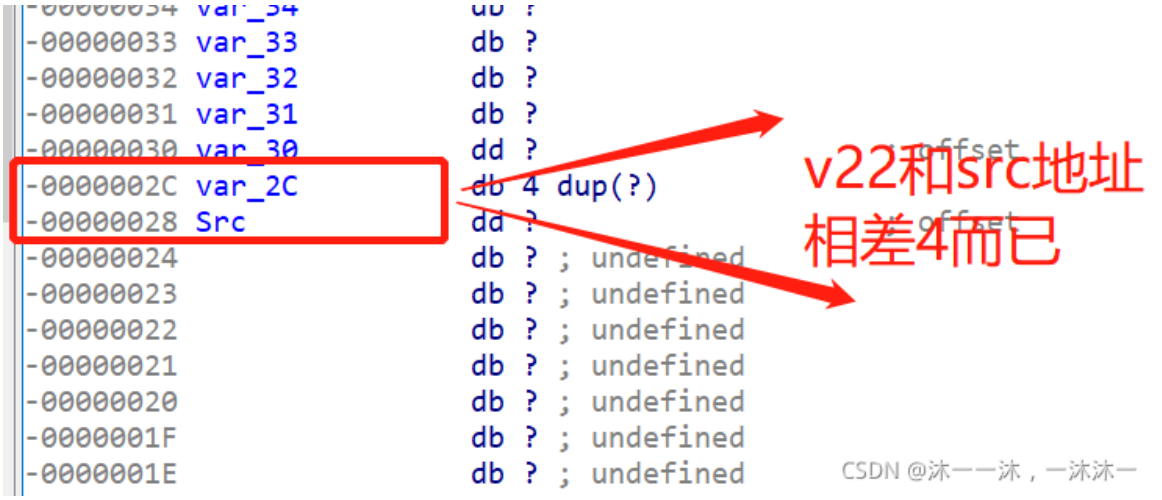
```
*(&str + len) = *v4;

str1_1 = str1;

++len;

--len_half;

++v4;

if ( len == 3 )

{

res0 = str >> 2;//这是熟悉的Base64加密算法，而且长度是3的倍数的情况下

res1 = (str1 >> 4) + 16 * (str & 3);

res2 = (str2 >> 6) + 4 * (str1 & 0xF);

res3 = str2 & 0x3F;

i = 0;

do

std::basic_string<char,std::char_traits<char>,std::allocator<char>>::operator+=(//这是C++的字符串运算符重载，把char转成string，方便直接字符叠加在后面。

&v22,

(word_1093020[*(&res0 + i++)] ^ 0x76));//从Base64表中（0x1093020）找到十进制下标所在的值异或0x76得到新值存到v22中，一次处理3个字符。

while ( i < 4 );

len = 0;

}

}

while ( len_half );

if ( len )

{

if ( len < 3 )//当长度不是3的倍数时，运算完，末尾加“=”填充，算法是一样的。

{

memset(&str + len, 0, 3 - len);

str1_1 = str1;

}

res1 = (str1_1 >> 4) + 16 * (str & 3);

res0 = str >> 2;
```

```
res2 = (str2 >> 6) + 4 * (str1_1 & 0xF);
k = 0;
for ( res3 = str2 & 0x3F; k < len + 1; ++k )
std::basic_string<char,std::char_traits<char>,std::allocator<char>>::operator+=(
&v22,
(word_1093020[*(&res0 + k)] ^ 0x76));
if ( len < 3 )
{
v9 = 3 - len;
do
{
std::basic_string<char,std::char_traits<char>,std::allocator<char>>::operator+=(&v22, '=');
--v9;
}
while ( v9 );
}
}
}
size = Size;
code_2 = code_1;
memset(code_1, 0, Size + 1);
src = Src;
if ( v25 < 0x10 )
src = &Src;
memcpy(code_2, src, size);//由栈分布可知src地址和v22相同，这是copy函数，把加密后的src存到code_2中再返回，也就是我们的v8啦
v26 = -1;
return std::basic_string<char,std::char_traits<char>,std::allocator<char>>::~basic_string<char,std::char_traits<char>,
();//析构函数
}
```

（这里积累第6个经验）

然后就是最后那一小段代码了，又是地址的重叠关联，前面加密的明明给了v22，最后这里却用了src的地址，看了堆栈发现地址只是相差4，虽然不是重叠，但是也应该有相通之处吧~

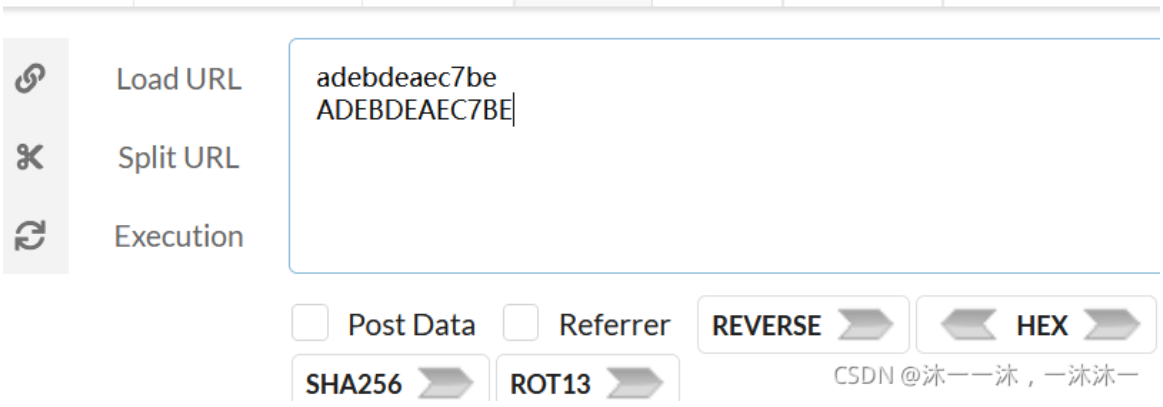




总的流程，输入的大写的十六进制----->转为十进制实数------>base64加密------->与aDdctfReverse明文相比较

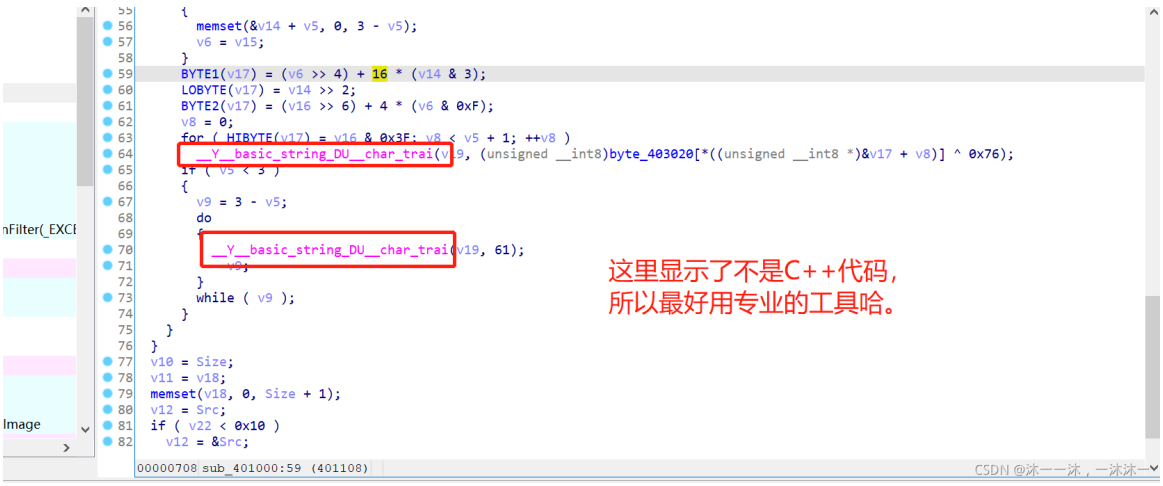
所以逆向逻辑就是aDdctfReverse来base64解密------>转大写十六进制。

结果：



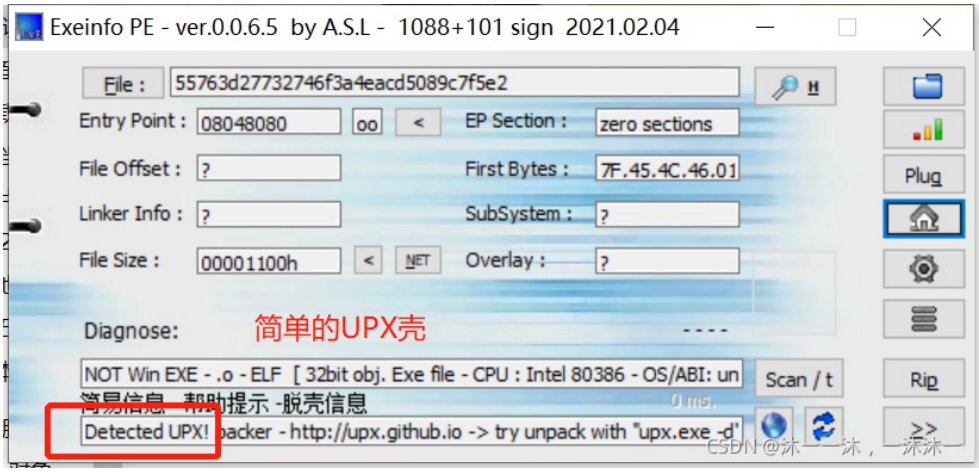
(这里积累第7个经验)

补充一个小插曲，用万能脱壳工具是脱壳是会乱码的，base64加密函数那里是显示有误的：

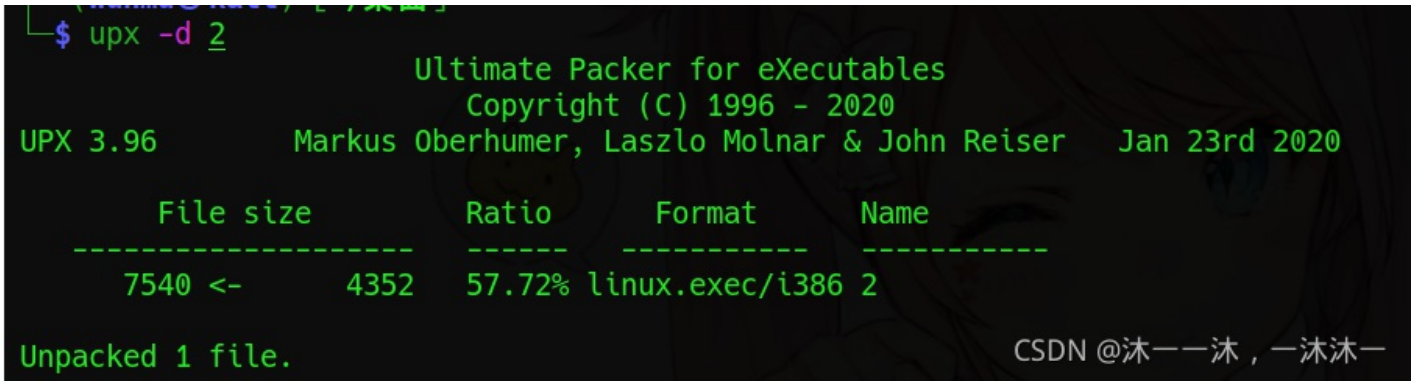


攻防世界easyre-153：（函数积累、线程操作积累、IDA伪代码生成优化）

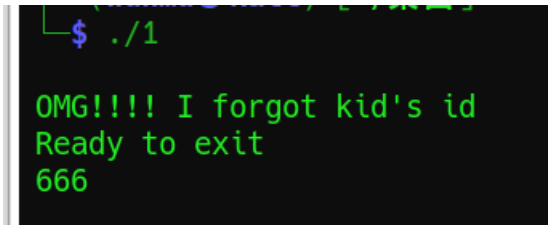
下载附件，照例扔入exeinfope中查看信息，发现有upx壳：



简单地扔入kali中用upx -d '文件名脱壳'：



照例运行一下程序查看主要回显信息



脱壳后照例扔入IDA32中查看伪代码，有main函数看main函数，如下图所示，代码量不多，主要是一些系统函数不懂，然后绿框那里就有个exit(0)，而输入的地方在后面。然后就懵了，exit(0)退出了怎么还能执行后面的__isoc99_scanf("%d", &v6);。

所以考察点就是这些系统函数v8 = __readgsdword(0x14u);、pipe(pipedes);、v5 = fork();read(pipedes[0], buf, 0x1Du);、了

一个进程在由 pipe()创建管道后，一般再fork一个子进程，然后通过管道实现父子进程间的通信：

```
  char buf[30]; // [esp+2Eh] [ebp-22h] BYREF
  unsigned int v8; // [esp+4Ch] [ebp-4h]

  v8 = __readgsdword(0x14u);
  pipe(pipedes);
  v5 = fork();
  if ( !v5 )
  {
    puts("\nOMG!!!! I forgot kid's id");
    write(pipedes[1], "6980087614356821435692875 3", 0x1Du);
    puts("Ready to exit     ");
    exit(0);
  }
  read(pipedes[0], buf, 0x1Du);
  __isoc99_scanf("%d", &v6);
  if ( v6 == v5 )
  {
    if ( (unsigned __int8)*(_DWORD *)((char *)lol + 3) == 204 )
    {
      puts(":D");
      exit(1);
    }
    printf("\nYou got the key\n ");
    lol(buf);
  }
  wait(0);
  return 0;
}
000007B7 main:26 (80487B7)
```

判断语句分割了两个一样的进程

子进程write写入字符串并退出

pipe管道的作用就是父子进程的通信

父进程read读取并执行后面代码

:hecks the segment permissions, class, and name

不难理解，flag就在lol函数中，跟踪lol函数，有意思的来了，它只有一个printf语句，但是反汇编代码中明显不止这一句代码：

```
19  read(pipedes[0], buf, 0x1Du);
20  __isoc99_scanf("%d", &v6);
21  if ( v6 == v5 )
22  {
23    if ( (unsigned __int8)*(_DWORD *)((char *)lol + 3) == 204 )
24    {
25      puts(":D");
26      exit(1);
27    }
28    printf("\nYou got the key\n ");
29    lol(buf);
30  }
31  wait(0);
32  return 0;
33}
```

前面输入的v6和pid值v5相等后应该要输出flag

所以flag应该就在lol函数中


```
1 int lol()
2 {
3   return printf("flag_is_not_here");
4 }
```

双击跟踪lol函数，只显示了一个printf语句

```
.text:080485F4 lol             proc near               ; CODE XREF: main+EE↓p
.text:080485F4                                         ; DATA XREF: main+AC↓o
.text:080485F4
.text:080485F4 var_13          = byte ptr -13h
.text:080485F4 var_12          = byte ptr -12h
.text:080485F4 var_11          = byte ptr -11h
.text:080485F4 var_10          = byte ptr -10h
.text:080485F4 var_F           = byte ptr -0Fh
.text:080485F4 var_E           = byte ptr -0Eh
.text:080485F4 var_D           = byte ptr -0Dh
.text:080485F4 var_C           = dword ptr -0Ch
.text:080485F4 arg_0           = dword ptr  8
.text:080485F4
.text:080485F4                 push    ebp
.text:080485F5                 mov     ebp, esp
.text:080485F7                 sub     esp, 28h
.text:080485FA                 mov     eax, [ebp+arg_0]
.text:080485FD                 add     eax, 1
.text:08048600                 movzx   eax, byte ptr [eax]
.text:08048603                 mov     edx, eax
.text:08048605                 mov     eax, [ebp+arg_0]
.text:08048608                 add     eax, 1
.text:0804860B                 movzx   eax, byte ptr [eax]
.text:0804860E                 lea     eax, [edx+eax]
.text:08048611                 mov     [ebp+var_13], al
.text:08048614                 mov     eax, [ebp+arg_0]
```

但是反汇编中明显不止printf这一条语句，也就说IDA没有把部分汇编语句反汇编成伪代码。

就因为这个IDA7.5版本的错误反汇编，所以找了大量资料才弄清了所以然：

（绿色是ida同步功能，显示第一条什么都没有的语句，竟然对应了这么多汇编，这明显就是没有反编译这一段，而且这些汇编明显是在作某种加密操作）

再往下翻一翻发现异常：

0x080486b0和0x080486b7处的汇编代码有明显联系

由于0x080486b0处对ebp+var_C的赋值，导致函数的固定跳转，这绿色的一段没有反编译很可能与此有关。

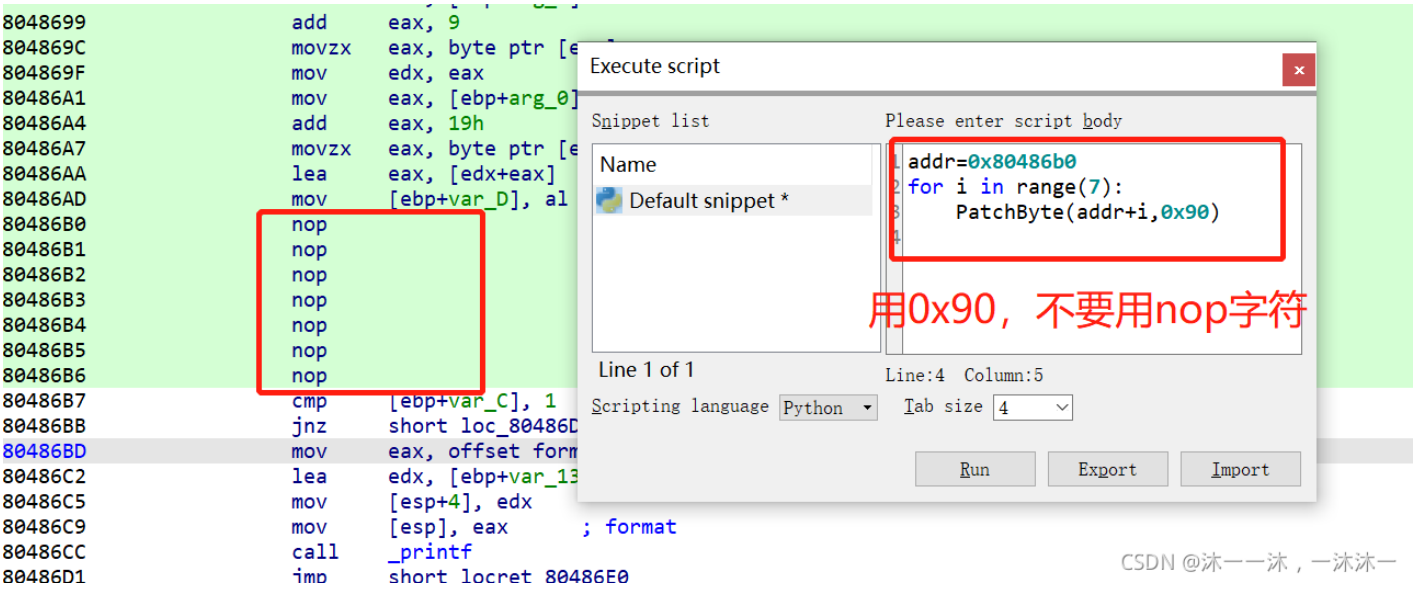
通过之前的学习，我知道在c语言的各种编辑器中，如果采取release版的输出，汇编代码都会因为O2优化而与源代码有较大差异，ida的反编译也会做类似操作。

在这里固定跳转会导致不可能运行到的代码干脆不进行反编译，这样相当于减少了工作量，优化了反编译时间。

但我寻思，这也不应该呀，不管最后固定跳转到下面的哪一个分支，上面这些数据处理的代码是必须运行的呀！但是仔细考究会发现，这上面的数据处理代码全在处理一两个变量。

然而如果固定跳转到80486d3的话，这操作压根和上面这处理半天的变量没关系，ida也很“聪明”地把没必要的数据处理给抛弃了。所以才有了我们刚开始分析看到的怪异函数内容。

那么综合以上分析，问题就在于这条mov指令，把它nop掉就可以正常F5反汇编了：



照着上面逻辑就可以输出flag了，因为这里考的是进程之间管道的简单通信，所以逻辑并不难：

key1="6980087614356821435692875 3"

flag=""

flag+=chr(2*ord(key1[1]))

flag+=chr(ord(key1[4])+ord(key1[5]))

flag+=chr(ord(key1[8])+ord(key1[9]))

flag+=chr(2*ord(key1[12]))

```
flag+=chr(ord(key1[18])+ord(key1[17]))

flag+=chr(ord(key1[10])+ord(key1[21]))

flag+=chr(ord(key1[9])+ord(key1[25]))

print("RCTF{"+flag+"}")
```

结果：

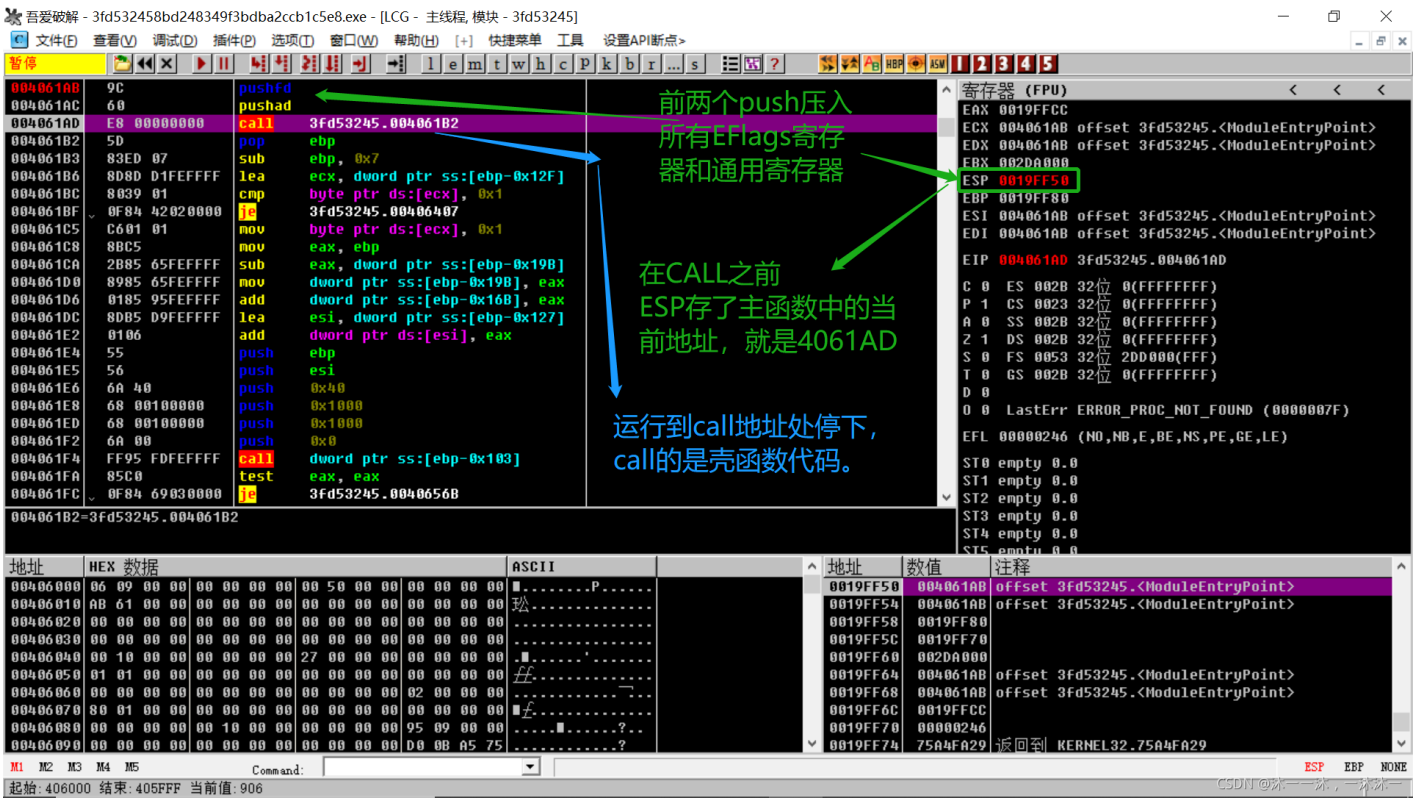
```
└─$ python 1.py
RCTF{rhelheg}
```

手动脱壳类型：

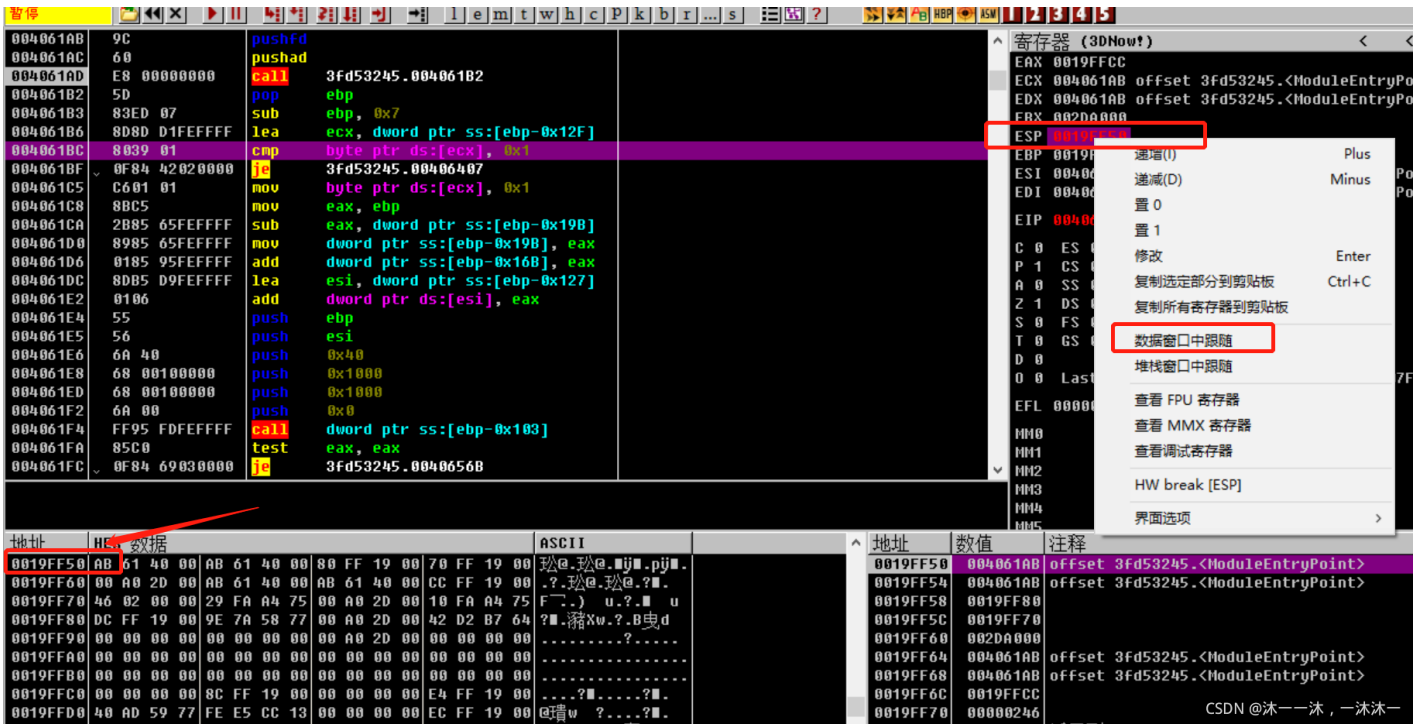
攻防世界crackme：（ESP脱壳定律、设立硬件访问断点、OD手动脱壳操作、工具脱壳、导入表修复）

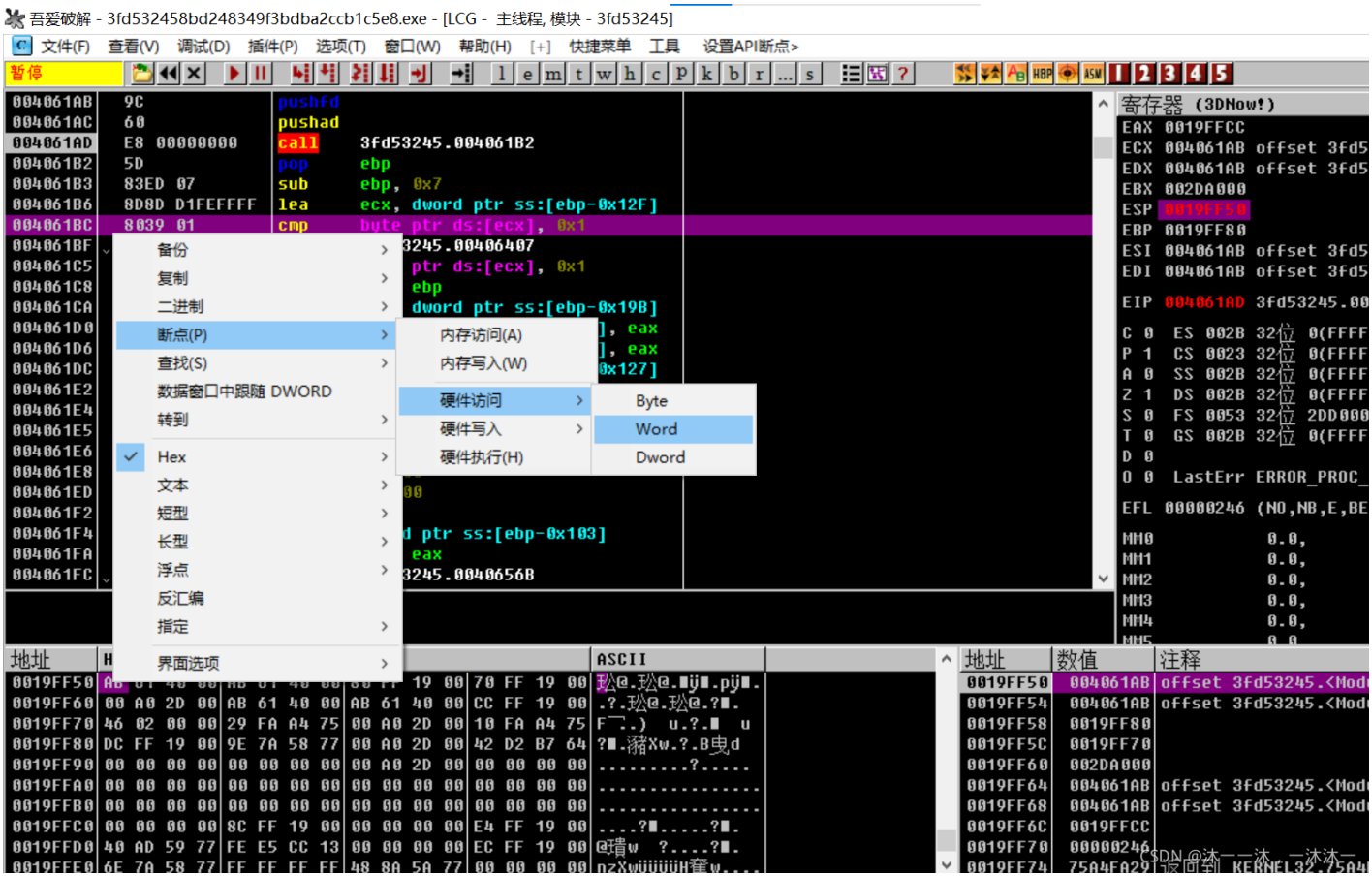
下载附件，照例扔入exeinfope中查看信息。nsPack壳，上网查了查说是北斗压缩壳(nsPack)，下面还提示用Quick unpack工具去壳：

利用ESP脱壳定律，手动脱壳，把程序扔如OD中，照着上面流程来一遍，这里面也可以看到pushfd和pushad是北斗壳的特征：

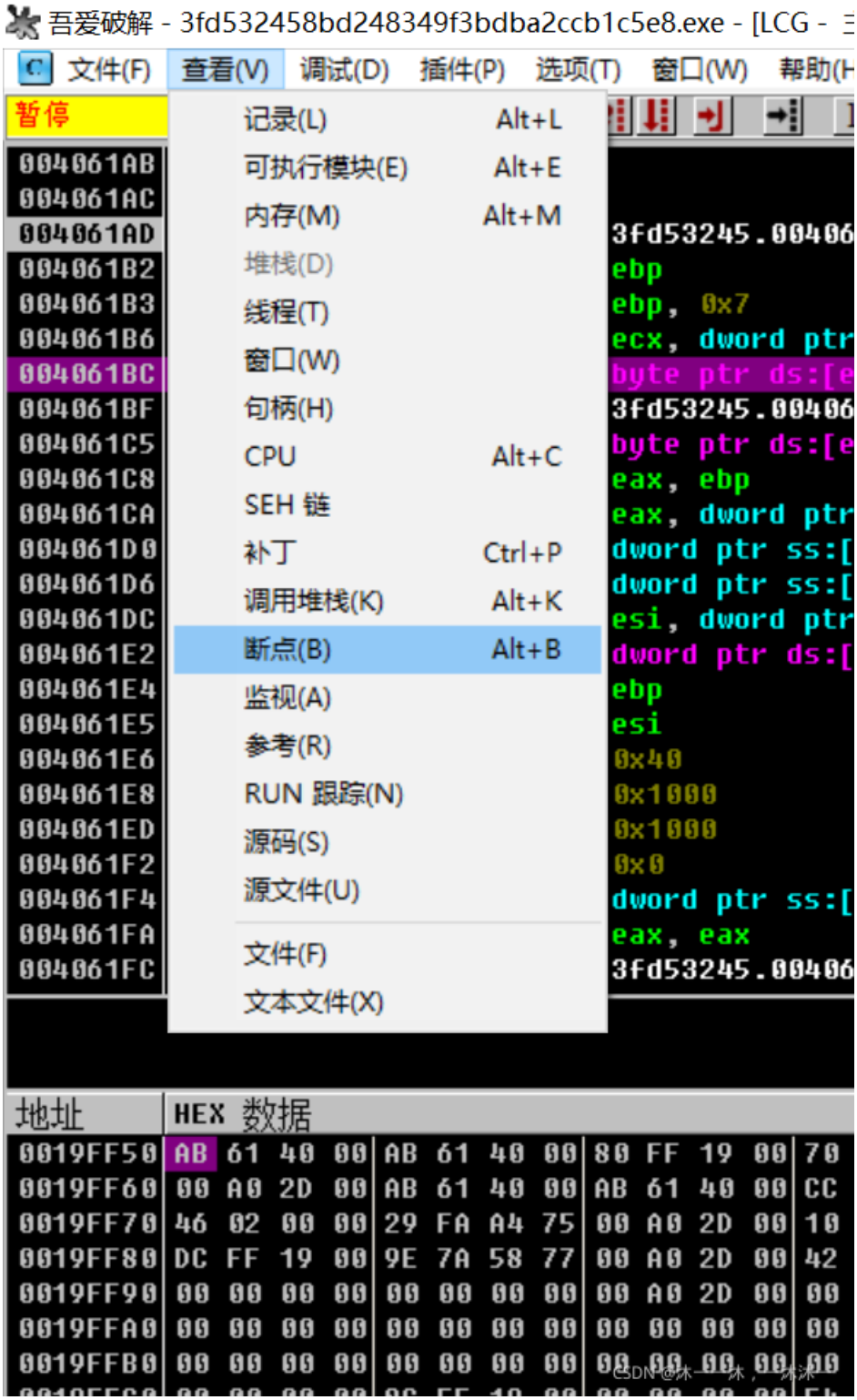


然后就是在ESP处数据窗口跟踪，再在数据窗口中右键设置word级或Dword级的硬件访问断点：





然后确认一下是否下了断点：



补充一下为什么是word级别，下面是《IDA权威指南中的解释》：（大小相关到断点的触发范围）

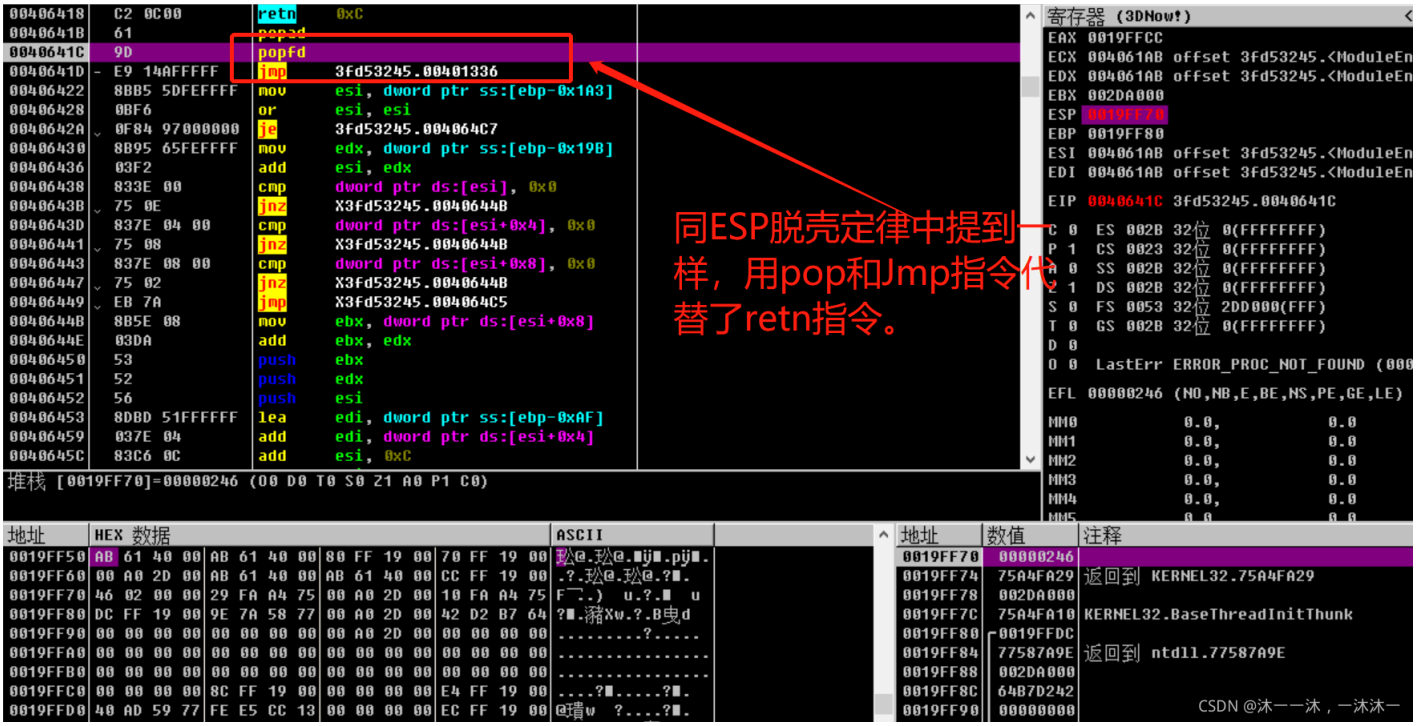
除了为硬件断点指定一种模式外，你还必须指定一个大小。执行类断点的大小必须为 1 字节，写入类或读取/写入类断点的大小可以设置为 1、2 或 4 字节。如果将大小设置为 2 字节，则断点的地址必须为字对齐（2 字节的整数倍）。同样，4 字节断点的地址必须为双字对齐（4 字节的整数倍）。硬件断点的大小和它的地址共同构成了触发这类断点的地址范围。下面举例说明。以在地址 0804C834h 处设置的一个 4 字节写入式断点为例，这个断点将由 1 字节写入 0804C837h、2 字节写入 0804C836h、4 字节写入 0804C832h 等操作触发。

(这里积累第四个经验)

点击运行程序，红框中的两行代码就是顶替了retn语句，这里标识壳函数调用完毕，要返回了。

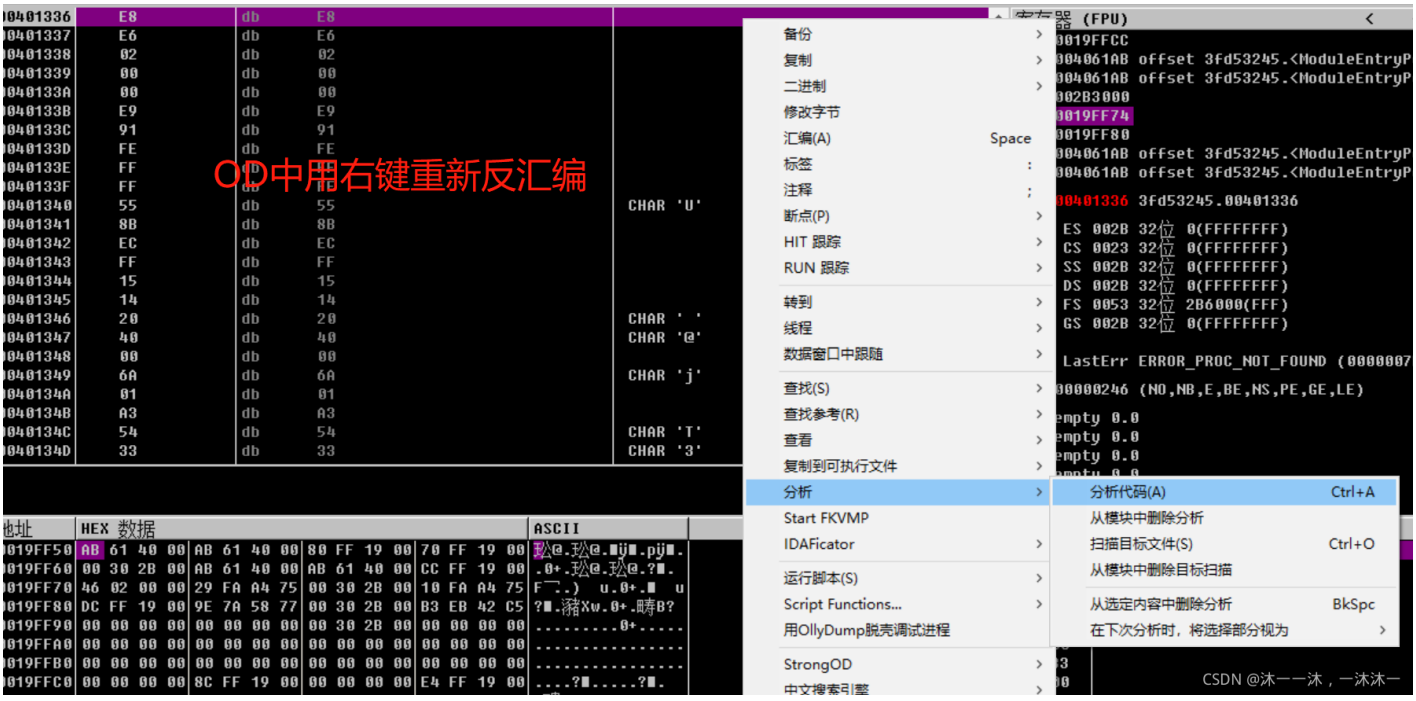
为什么停在这里？因为我们定下的是访问断点，停在这里是因为弹出当前的ESP 0019FF70栈内容后就要指向ESP 0019FF50了，就是对0019FF50栈地址处的一种访问，所以会断在这里。

(额。。。这里直接弹出到0019FF74去了，按理说应该是0019FF50的。我也不明白，可能是我理解有什么错误，或者这程序里面有什么高深的栈平衡技巧吧)

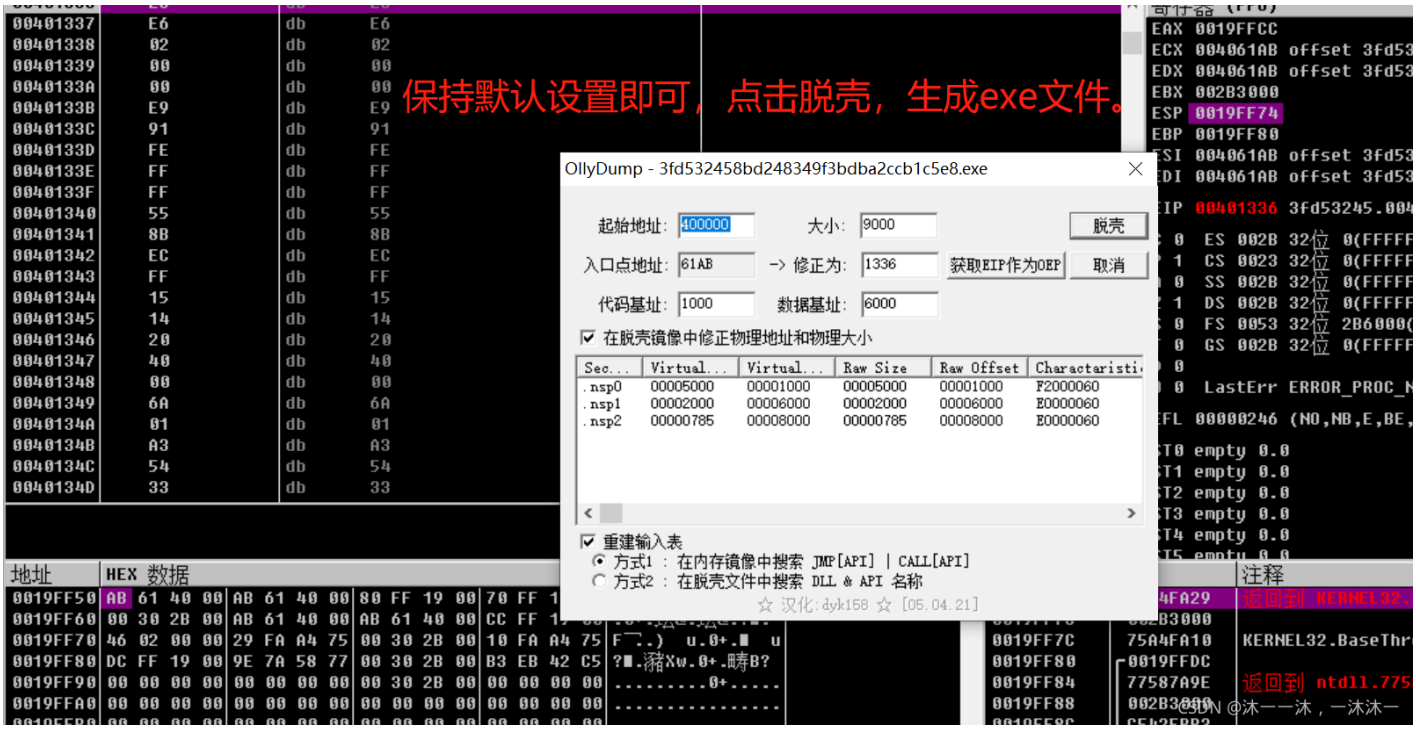


(这里积累第五个经验)

jmp跳出壳后就是解压缩后的代码了，右键分析当前代码把.text.段中数据重新反汇编成汇编代码，同IDA的热键C。



然后就是在这个位置处脱壳，用OD插件OllyDump来脱壳，生成新的可执行程序，然后IDA伪代码分析，有main函数看main函数：



3.exe

逻辑简单，因为考点是脱壳，所以直接附上脚本：

key1="this_is_not_flag"

key2=[18, 4, 8, 20, 36, 92, 74, 61, 86, 10, 16, 103, 0, 65, 0, 1, 70, 90, 68, 66, 110, 12, 68, 114, 12, 13, 64, 62, 75, 95, 2, 1, 76, 94, 91, 23, 110, 12, 22, 104, 91, 18]

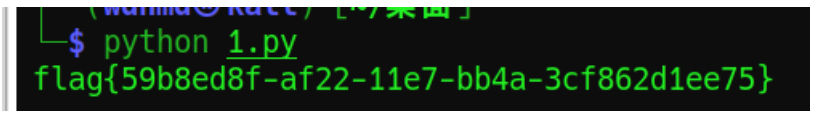
flag=""

for i in range(len(key2)):

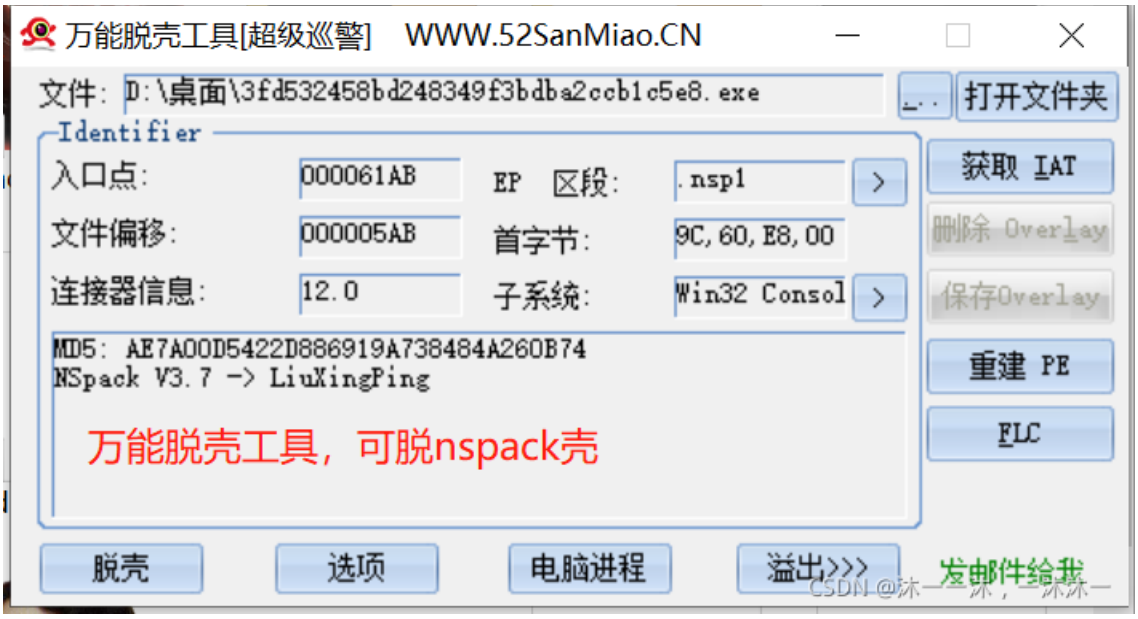
flag+=chr(key2[i]^ord(key1[i%16]))

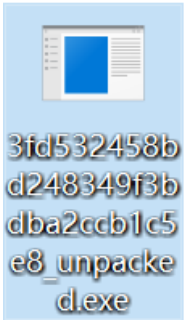
print(flag)

结果：

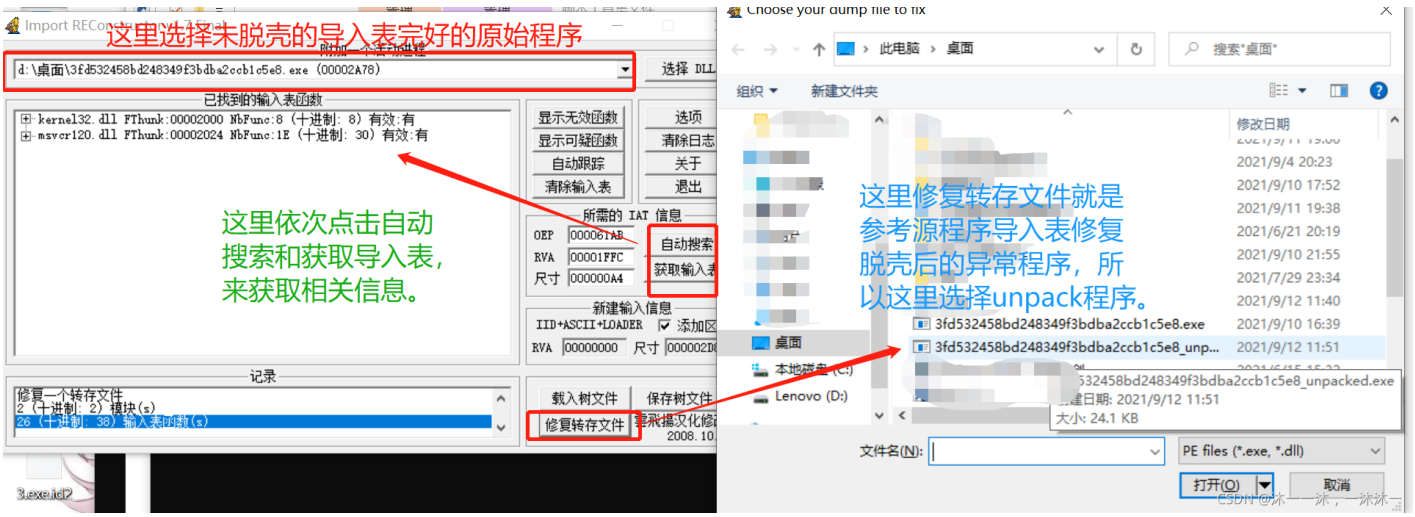


然后附上第二种方法，工具脱壳，使用万能脱壳工具，扔进去，脱壳，生成同名+unpack程序：





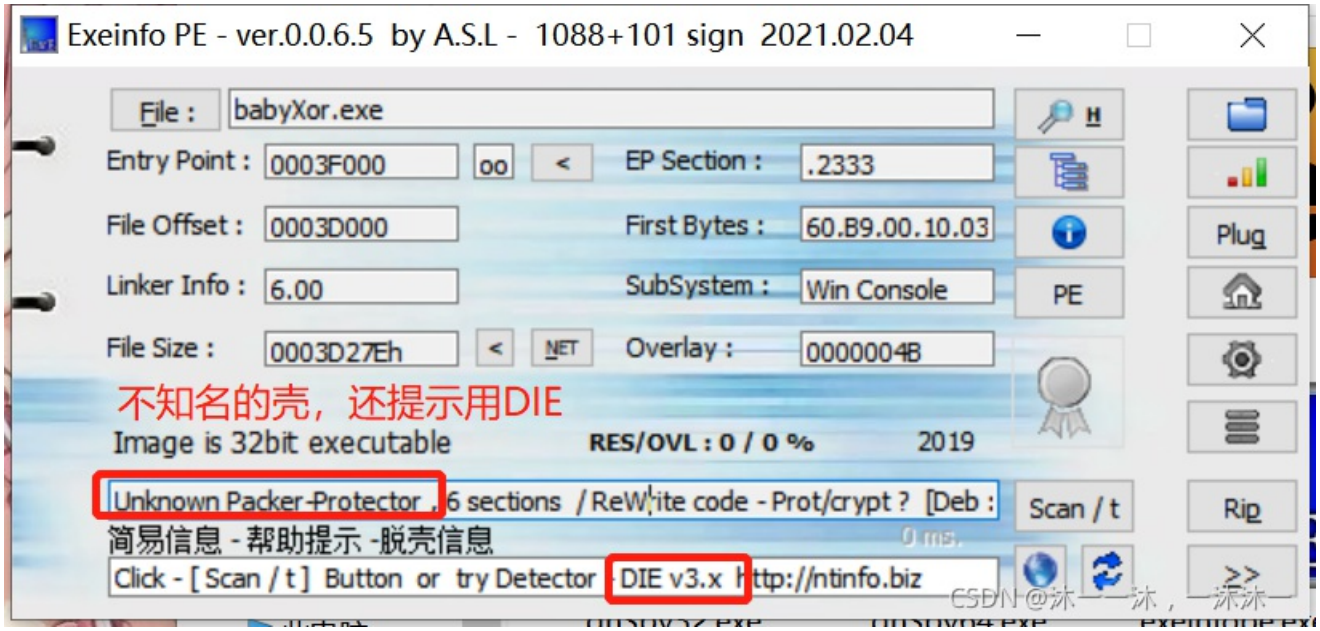
然后这个程序运行不了，因为导入表乱了，所以用importREC修复导入表。因为ImportREC是内存转储，所以要在运行程序中判断导入的库，首先双击运行前面万能脱壳工具脱壳后的程序：(生成同名的多个_的程序)

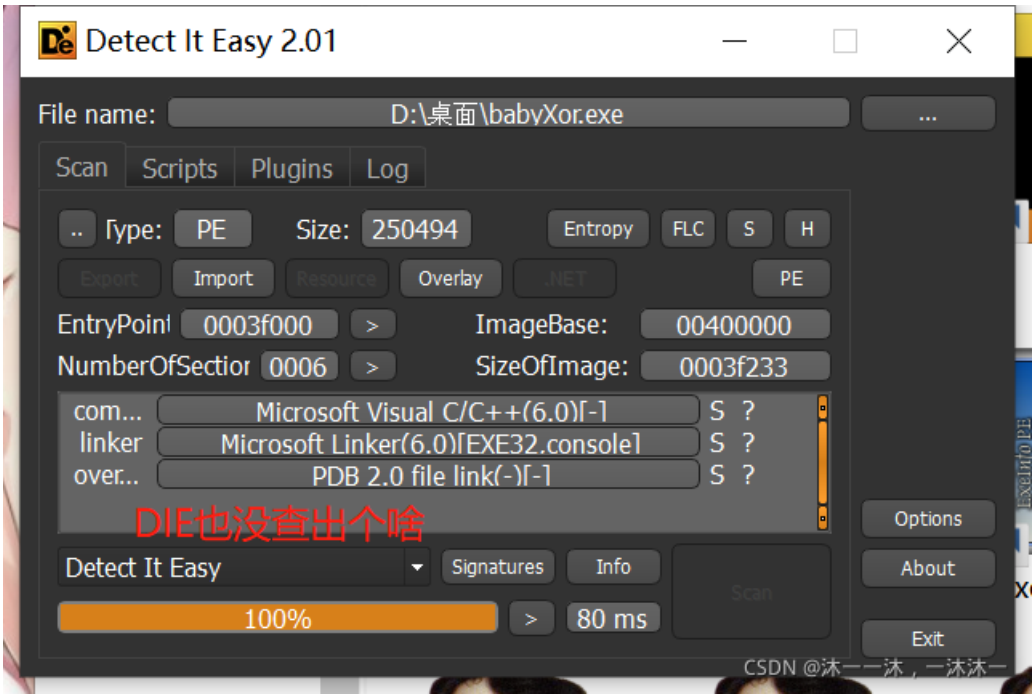


该程序就是正常的。

攻防世界BabyXor：（ESP脱壳定律、OD动态调试、OD手动脱壳操作、导入表修复、函数逻辑封装、函数积累、环境准备函数、不用输入类型、正向爆破、地址差值操作、for空执行循环遍历字符串）

下载附件，照例扔入exeinfope中查看信息，结果是不知名的壳，还叫我用DIE查看，结果也没分析出来：

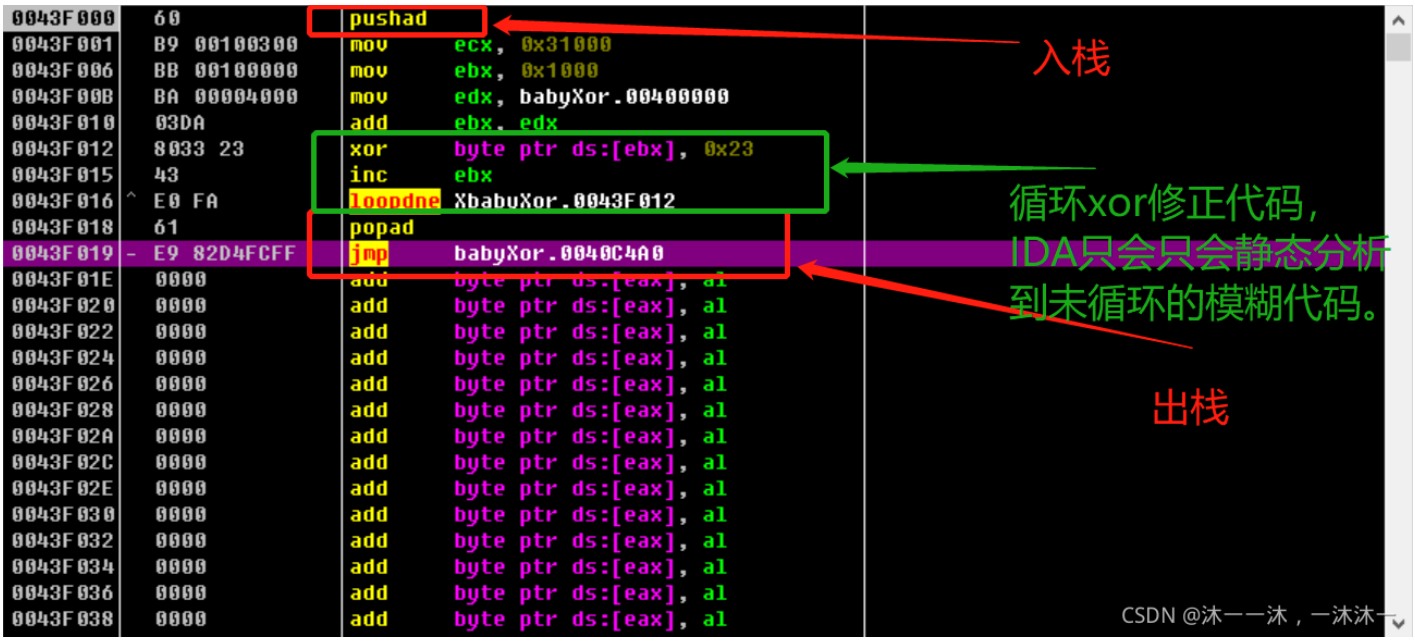




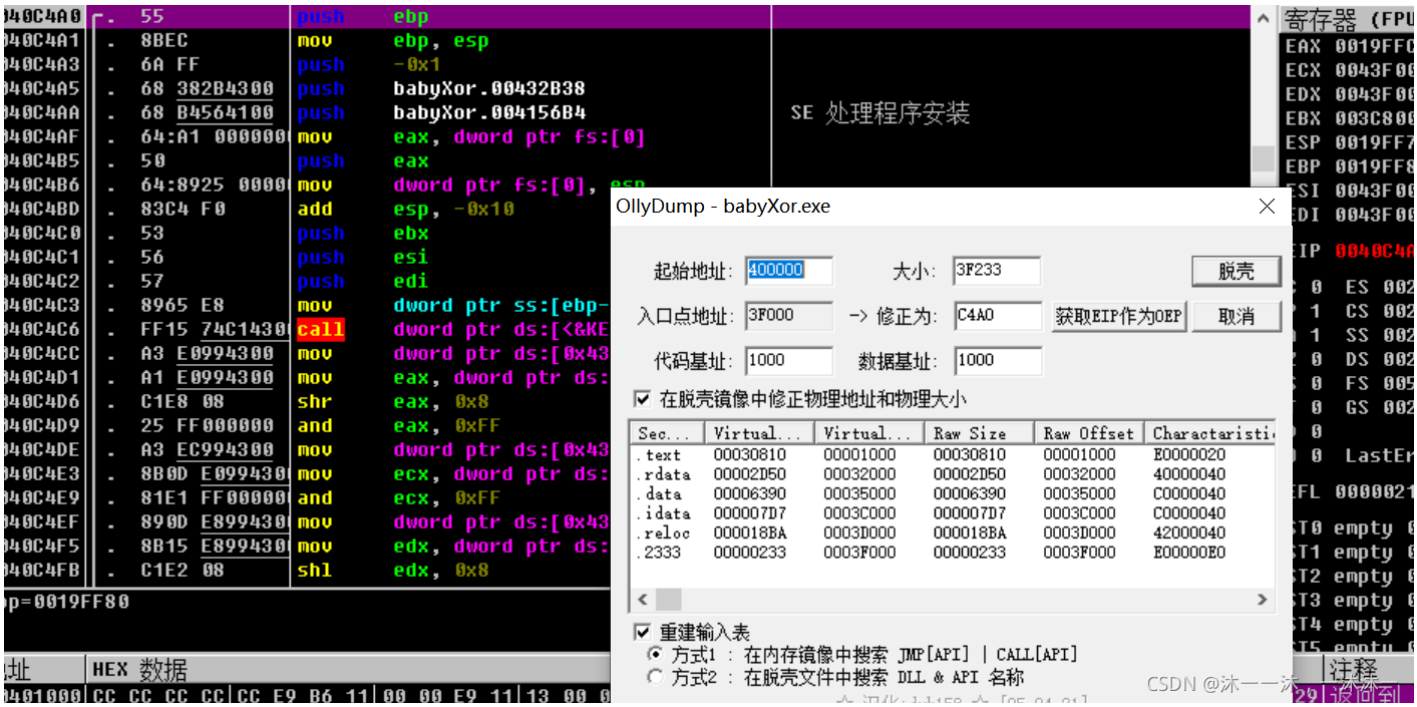
(这里积累第一个经验)

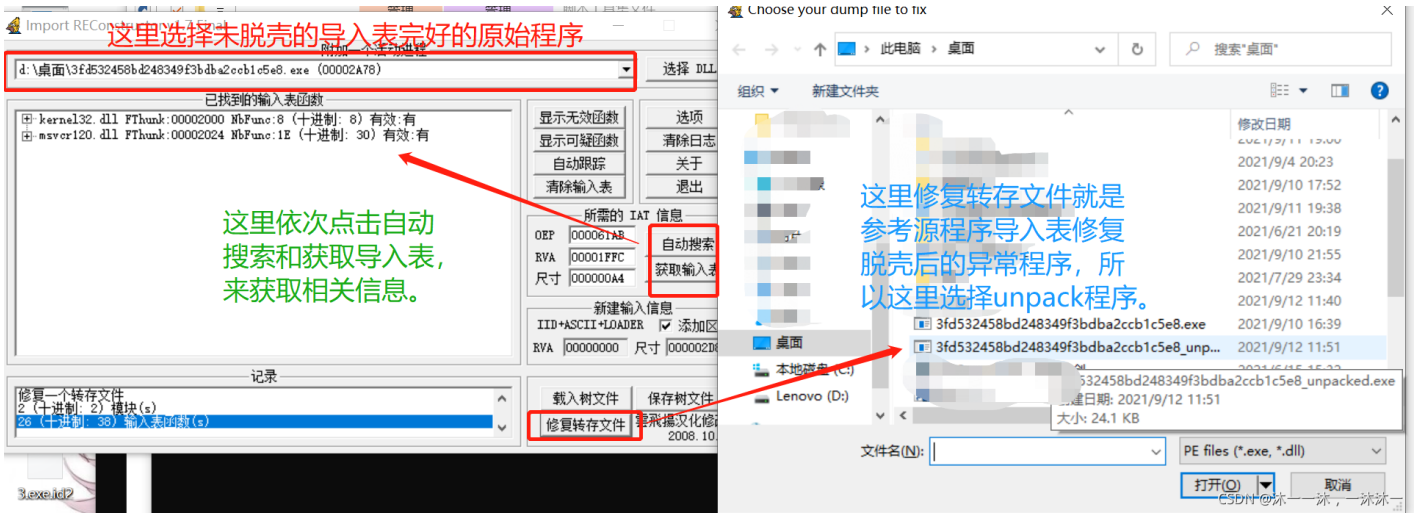
那没办法只能手动脱壳了，回顾以前的crackme题中积累的ESP脱壳定律，在这里也同样可以借鉴一二：

就是这里的pop和jmp代替retn，载入OD中我们可以发现开头处有相似的入栈出栈操作，只不过没有调用壳层函数，而是直接用代码解压缩而已：（就是直接把函数内容平铺出来了）



断点执行到jmp处右键分析代码然后点击插件处的脱壳即可，最后按照以前的教程用importRCE修复导入表，虽然还是运行不了~





不管了，照例扔入IDA32中查看伪代码信息，有Main函数看main函数：

```
int __cdecl main_0(int argc, const char **argv, const char **envp)
{
  size_t v3; // eax
  int v5; // [esp+50h] [ebp-1Ch]
  void *v6; // [esp+54h] [ebp-18h]
  const char *Src; // [esp+58h] [ebp-14h]
  int v8; // [esp+5Ch] [ebp-10h]

  sub_4010B4((int)&unk_4395F0, (char *)&byte_432020);
  sub_40107D(sub_40102D);
  if ( --File._cnt < 0 )
    _filbuf(&File);
  else
    ++File._ptr;
  v8 = sub_40108C((int)&unk_435DC0, 56);
  Src = (const char *)sub_401041((int)&unk_435DC0, (int)&dword_435DF8, 56u);
  v6 = malloc(0x64u);
  v3 = strlen(Src);
  memcpy(v6, Src, v3);
  v5 = sub_4010C3(&unk_435DC0, Src, &dword_435E30, 56);
  sub_40101E(v8, Src, v5);                      // flag1, flag2, flag3和flag4
  return 0;
}
```

6个自定义函数

(这里积累第二个经验)

一开始这全都是自定义函数的操作属实把我吓倒了，唉~ 总归还是底气不足啊，应该耐性下来一个个分析才对，不会太难的。

首先红框中的_filbuf(&File);在博客https://www.cnblogs.com/volcanol/archive/2011/06/09/2076907.html中有很好的诠释。

博客内容提取一下就是_filbuf(&File)是getc()的一个实现，就是读取输入。

getc() 在VC 6.0中有两个get()的定义， 一个是宏，一个是函数

宏的定义如下：

#define getc(_stream) (--(_stream)->_cnt >= 0 ? 0xff & *(_stream)->_ptr++ : _filbuf(_stream))

函数定义如下：

_CRTIMP int __cdecl getc(FILE *);


```
int __cdecl main_0(int argc, const char **argv, const char **envp)
{
  size_t v3; // eax
  int v5; // [esp+50h] [ebp-1Ch]
  void *v6; // [esp+54h] [ebp-18h]
  const char *Src; // [esp+58h] [ebp-14h]
  int v8; // [esp+5Ch] [ebp-10h]

  sub_4010B4((int)&unk_4395F0, (char *)&byte_432020);
  sub_40107D(sub_40102D);
  if ( --File._cnt < 0 )
    _filbuf(&File);
  else
    ++File._ptr;
  v8 = sub_40108C((int)&unk_435DC0, 56);
  Src = (const char *)sub_401041((int)&unk_435DC0, (int)&dword_435DF8, 56u);
  v6 = malloc(0x64u);
  v3 = strlen(Src);
  memcpy(v6, Src, v3);
  v5 = sub_4010C3(&unk_435DC0, Src, &dword_435E30, 56);
  sub_40101E(v8, Src, v5);                      // flag1, flag2, flag3和flag4
  return 0;
}
```

前面是读取输入的，那么关键就在后面了，耐心一点一个个点进去分析一下。第一个自定义函数sub_40108C是简单的移位和异或操作，赋值给v3[i]

```
 8
 9  sub_4010B4((int)&unk_4395F0, (char *)&byte_432020);
10  sub_40107D(sub_40102D);
11  if ( --File._cnt < 0 )
12    _filbuf(&File);
13  else
14    ++File._ptr;
15  v8 = sub_40108C((int)&unk_435DC0, 56);
16  Src = (const char *)sub_401041((int)&unk_435DC0, (int)&dword_435DF8, 56u);
17  v6 = malloc(0x64u);
18  v3 = strlen(Src);
19  memcpy(v6, Src, v3);
20  v5 = sub_4010C3(&unk_435DC0, Src, &dword_435E30, 56);
21  sub_40101E(v8, Src, v5);                      // flag1, flag2, flag3和flag4
22  return 0;
23}
```

用户输入

用户输入后第一个自定义函数，双击跟踪分析。


```
1 char *__cdecl sub_401190(int a1, unsigned int a2)
2 {
3   char *v3; // [esp+4Ch] [ebp-Ch]
4   int i; // [esp+54h] [ebp-4h]
5
6   v3 = (char *)malloc(a2 >> 2);
7   for ( i = 0; i < (int)(a2 >> 2); ++i )
8     sprintf(&v3[i], "%c", i ^ *(_DWORD *)(a1 + 4 * i));
9   return v3;
0}
```

简单的移位和异或操作，赋值给v3[i]，这种单向移位会丢失数，很难逆向，所以后面flag生成与输入无关也验证了这点。

第二个函数sub_401041还是移位和异或操作，赋值给Buffer[i]：

```
10  sub_40107D(sub_40102D);
11  if ( --File._cnt < 0 )
12    _filbuf(&File);
13  else
14    ++File._ptr;
15  v8 = sub_40108C((int)&unk_435DC0, 56);
16  Src = (const char *)sub_401041((int)&unk_435DC0, (int)&dword_435DF8, 56u);
17  v6 = malloc(0x64u);
18  v3 = strlen(Src);
19  memcpy(v6, Src, v3);
20  v5 = sub_4010C3(&unk_435DC0, Src, &dword_435E30, 56);
21  sub_40101E(v8, Src, v5);                      // flag1, flag2, flag3和flag4
22  return 0;
23}
```

输入后的第二个自定义函数，双击跟踪。


```
1 char *__cdecl sub_401240(int a1, int a2, size_t Size)
2 {
3   signed int i; // [esp+4Ch] [ebp-Ch]
4   char *Buffer; // [esp+50h] [ebp-8h]
5
6   Buffer = (char *)malloc(Size);
7   sprintf(Buffer, "%c", *(_DWORD *)a2);
8   for ( i = 1; i < (int)(Size >> 2); ++i )
9     sprintf(&Buffer[i], "%c", *(_DWORD *)(a1 + 4 * i) ^ *(_DWORD *)(a2 + 4 * i) ^ *(_DWORD *)(a1 + 4 * i - 4));
10  return Buffer;
11}
```

又是单向的移位异或，和前面的一样

第三个函数sub_4010C3还是简单的移位、异或、拼接操作，最终赋值给destination

```
    _filbuf(&File);
  else
    ++File._ptr;
  v8 = sub_40108C((int)&unk_435DC0, 56);
  Src = (const char *)sub_401041((int)&unk_435DC0, (int)&dword_435DF8, 56u);
  v6 = malloc(0x64u);
  v3 = strlen(Src);
  memcpy(v6, Src, v3);
  v5 = sub_4010C3(&unk_435DC0, Src, &dword_435E30, 56);
  sub_40101E(v8, Src, v5);                      // flag1, flag2, flag3和flag4
  return 0;
}
```

输入之后的第三个自定义函数

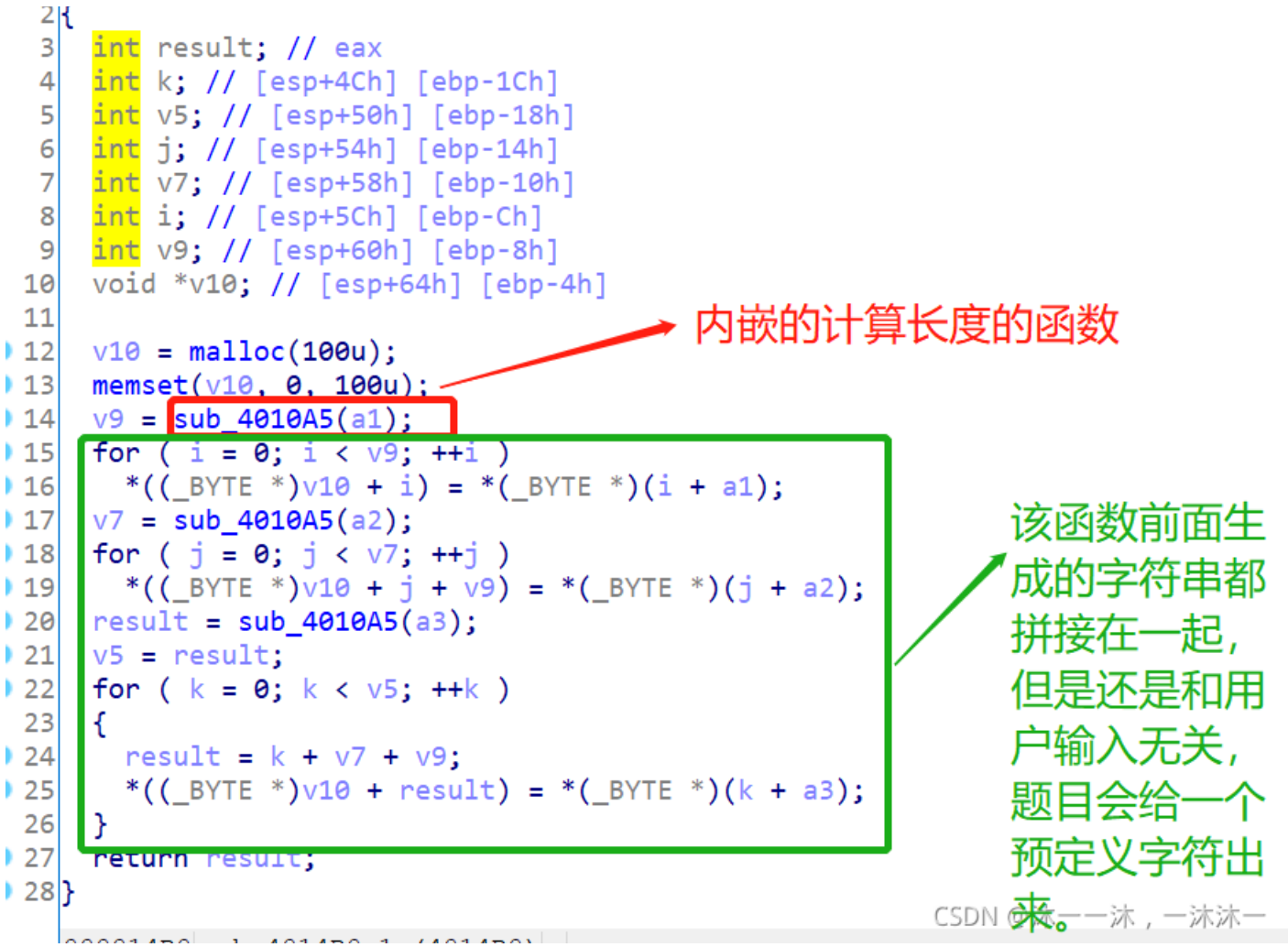

```
char *__cdecl sub_401320(int a1, int a2, int a3, unsigned int a4)
{
  int i; // [esp+4Ch] [ebp-10h]
  char *v6; // [esp+50h] [ebp-Ch]
  char *Source; // [esp+54h] [ebp-8h]

  Source = (char *)malloc(a4 - 1);
  v6 = (char *)malloc(4 * a4 - 1);
  for ( i = 0; i < (int)((a4 >> 2) - 1); ++i )
  {
    sprintf(&v6[i], "%c", *(_DWORD *)(a3 + 4 * i + 4) ^ *(char *)(i + a2));
    sprintf(&Source[i], "%c", i ^ v6[i]);
  }
  sprintf(&Destination, "%c", dword_435E30 ^ dword_435DF8);
  strcat(&Destination, Source);
  return &Destination;
}
```

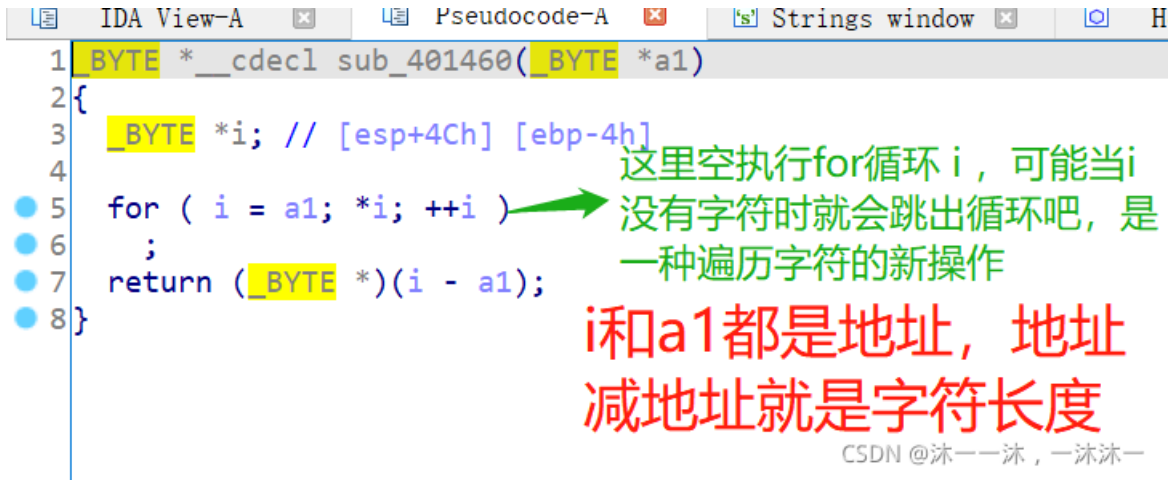
还是单向的移位，异或，拼接

看最后一个函数sub_40101E，内部还有个sub_4010A5函数，也双击跟踪看一下，发现是简单的计算长度strlen操作，那么这个函数就是把前面生成的字符串都拼接在一起的操作。

```
    _filbuf(&File);
  else
    ++File._ptr;
  v8 = sub_40108C((int)&unk_435DC0, 56);
  Src = (const char *)sub_401041((int)&unk_435DC0, (int)&dword_435DF8, 56u);
  v6 = malloc(0x64u);
  v3 = strlen(Src);
  memcpy(v6, Src, v3);
  v5 = sub_4010C3(&unk_435DC0, Src, &dword_435E30, 56);
  sub_40101E(v8, Src, v5);                      // flag1, flag2, flag3和flag4
  return 0;
}
```

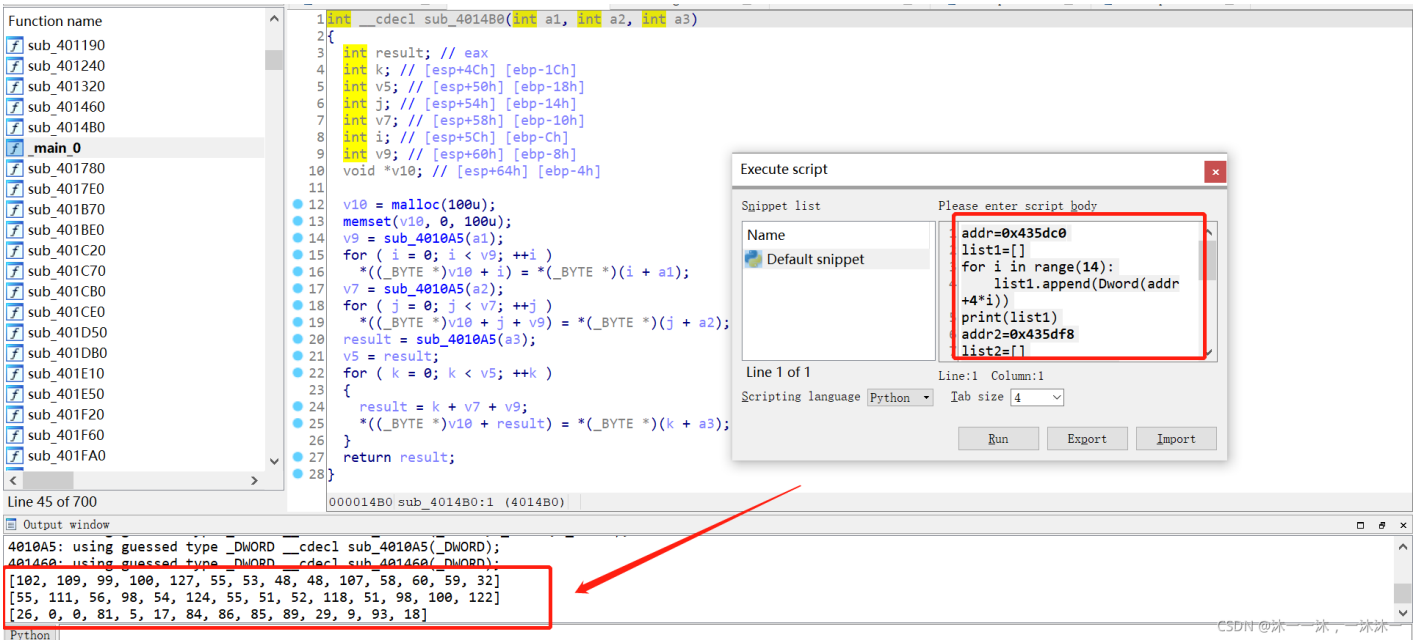
用户输入的最后一个自定义函数，双击跟踪分析。





很耐性的分析完了，好像和输入没有什么关系，那么应该是和输入无关的存储型flag类型了，先静态写代码生成。

首先打印所有用于操作的数组：



然后仿照逻辑写脚本：（正向逻辑复现）

key1=[102, 109, 99, 100, 127, 55, 53, 48, 48, 107, 58, 60, 59, 32]

flag1=""

for i in range(14):

flag1+=chr(i^key1[i])

#print(flag1)

key2=[55, 111, 56, 98, 54, 124, 55, 51, 52, 118, 51, 98, 100, 122]

flag2=""

flag2+=chr(key2[0]) #!!!!!!wocao

for i in range(1,14,1):

flag2+=chr(key1[i]^key2[i]^key1[i-1])

#print(flag2)

key3=[26, 0, 0, 81, 5, 17, 84, 86, 85, 89, 29, 9, 93, 18]

flag4=""

for i in range(13):

flag4+=chr(i^key3[i+1]^ord(flag2[i])) #前面flag2错了，所以这里也错了！！！！

flag3=""

flag3=chr(key2[0]^key3[0])

print(flag1+flag2+flag3+flag4)

（这里积累第三个经验）

写脚本中遇到一个坑就是第二个自定义函数flag2的sub_401041函数那里，Buffer[0]被赋了初始值啊，忽略的话对后面修改很大:

```
1 char *__cdecl sub_401240(int a1, int a2, size_t Size)
2 {
3   signed int i; // [esp+4Ch] [ebp-Ch]
4   char *Buffer; // [esp+50h] [ebp-8h]
5
6   Buffer = (char *)malloc(Size);
7   sprintf(Buffer, "%c", *(_DWORD *)a2);
8   for ( i = 1; i < (int)(Size >> 2); ++i )
9     sprintf(&Buffer[i], "%c", *(_DWORD *)(a1 + 4 * i) ^ *(_DWORD *)(a2 + 4 * i) ^ *(_DWORD *)(a1 + 4 * i - 4));
10  return Buffer;
11 }
```

这里给Buffer[0]赋了初始值啊！！！

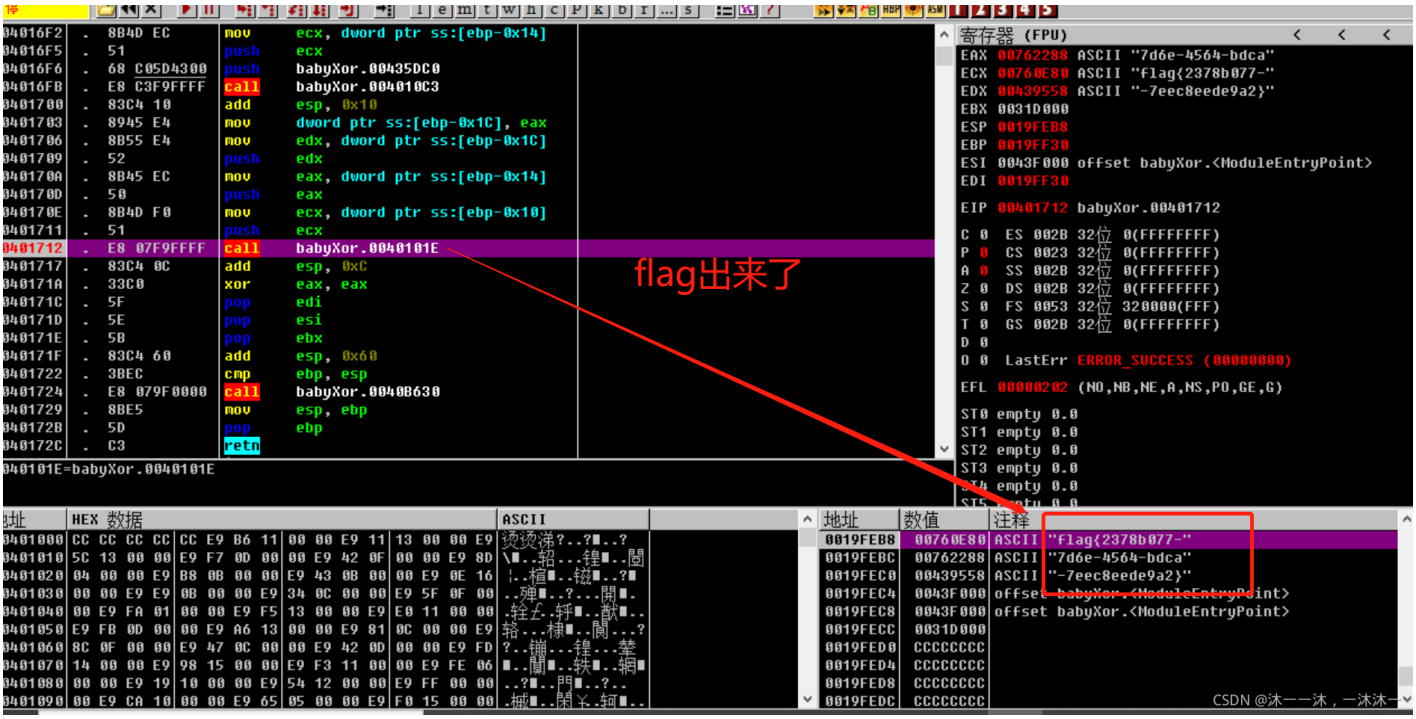
从1开始给到Buffer[1]，但是Buffer[0]在前面赋值啊！！！

结果：

```
flag{2378b077-7d6e-4564-bdca-7eec8eede9a2}
```

第二个方法，动态调试，在最后一个函数0x401712处下断点看寄存器即可：（可以动态调试，但是不能直接运行喔）

```
.text:00401703                 mov     [ebp+var_1C], eax
.text:00401706                 mov     edx, [ebp+var_1C]
.text:00401709                 push    edx
.text:0040170A                 mov     eax, [ebp+Src]
.text:0040170D                 push    eax
.text:0040170E                 mov     ecx, [ebp+var_10]
.text:00401711                 push    ecx
.text:00401712                 call    sub_40101E
.text:00401717                 add     esp, 0Ch
.text:0040171A                 xor     eax, eax
.text:0040171C                 pop     edi
.text:0040171D                 pop     esi
```

花指令题目类型（代码与数据混合）

自定义函数自修改：

攻防世界BABYRE：（函数名称暗示、IDA热键重新反汇编、IDA动态调试、栈地址连续小数组）

这题应该是我第一次接触的花指令题，怪兴奋的！目前我的理解是代码和数据混淆，让IDA误认为是数据而反汇编出错。

64位ELF文件，无壳，扔入IDA64位中查看伪代码：

```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3   char input_flag[24]; // [rsp+0h] [rbp-20h] BYREF
4   int v5; // [rsp+18h] [rbp-8h]
5   int i; // [rsp+1Ch] [rbp-4h]
6
7   for ( i = 0; i <= 181; ++i )
8     judge[i] ^= 0xCu;
9   printf("Please input flag:");
10  __isoc99_scanf("%20s", input_flag);
11  v5 = strlen(input_flag);
12  if ( v5 == 14 && !(*(unsigned int (__fastcall **)(char *))judge)(input_flag) )
13    puts("Right!");
14  else
15    puts("Wrong!");
16  return 0;
17 }
```

judge将会生成函数，所以judge数组也可接受参数。

这里犯下第一个错误：

由于是第一次接触花指令，对 (*(unsigned int (__fastcall **)(char *))judge)(input_flag)) 这一行代码我看了好久愣是没看懂，input_flag是我们输入的字符串，judge中文暗示是判断，根据以前做题这种中文类名字的确是暗示，可前面显示的judge[i] 这摆明是个数组啊，judge(input_flag) 不就变成了数组首地址加Input_flag了吗？还有这种玩法？？？

然后我就去看wp了(笑~)：

他们大概意思了judge是一个函数名，这个judge函数名的函数呢是通过前面逻辑用数据生成出来的。

想起C语言好像有通过字符串拼接生成新命令的技巧，突然就恍然大悟了。

所以这里 (*(unsigned int (__fastcall **)(char *))judge)(input_flag)) 应该这样分析，unsigned int 是judge函数的返回类型，(__fastcall **)是函数的调用约定，(char *)这个只是提取judge的数组头的一连串字符串做函数名而已。

然后看他们WP中显示judge是双击跟踪跟踪不了的，会报错，我的倒是跟踪得了，长这样：

```
.data:0000000000600B00 ; char judge[182]
.data:0000000000600B00 judge           db 59h                  ; CODE XREF: main+80↑p
.data:0000000000600B00                                         ; DATA XREF: main+16↑r ...
.data:0000000000600B01                 db  44h ; D
.data:0000000000600B02                 db  85h
.data:0000000000600B03                 db 0E9h
.data:0000000000600B04                 db  44h ; D
.data:0000000000600B05                 db  85h
.data:0000000000600B06                 db  71h ; q
.data:0000000000600B07                 db 0D4h
.data:0000000000600B08                 db 0CAh
.data:0000000000600B09                 db  49h ; I
.data:0000000000600B0A                 db 0ECh
.data:0000000000600B0B                 db  6Ah ; j
.data:0000000000600B0C                 db 0CAh
.data:0000000000600B0D                 db  49h ; I
.data:0000000000600B0E                 db 0EDh
.data:0000000000600B0F                 db  61h ; a
.data:0000000000600B10                 db 0CAh
.data:0000000000600B11                 db  49h ; I
.data:0000000000600B12                 db 0EEh
.data:0000000000600B13                 db  6Fh ; o
.data:0000000000600B14                 db 0CAh
.data:0000000000600B15                 db  49h ; I
.data:0000000000600B16                 db 0EFh
00000B00 0000000000600B00: .data:judge (Synchronized with Hex View-1)
```

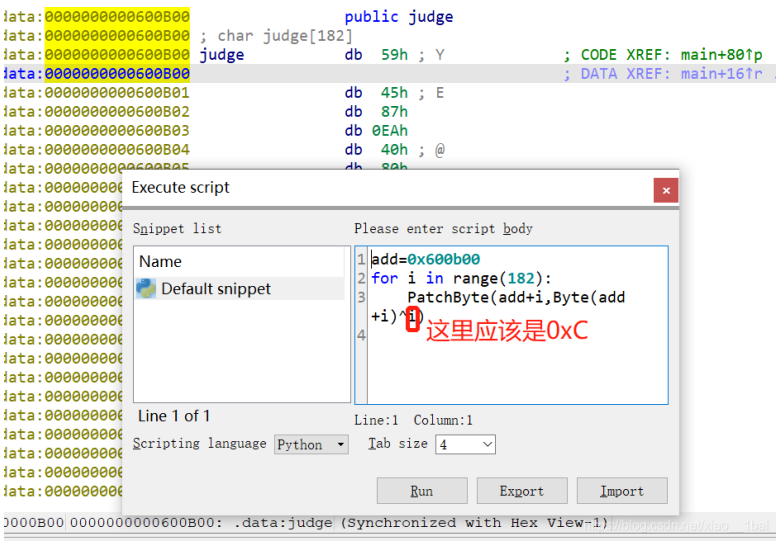
这里犯下第二个错误：

一开始这里也卡了我好一会，后来发现是我用了新版的IDA7.5,这里本来有个指针分析错误，IDA7.5直接把它当成数据去了，如果我用C（汇编），P（创建函数），在judge开头处按一下照样显示指针分析错误：(ps：不按C的话有时IDA会报错函数在指定地址具有未定义的指令/数据，您的请求已放入自动分析队列。按一下C可能是提醒它可以转成数据吧)

```
.data:0000000000600B00
.data:0000000000600B00 ; char judge[182]
.data:0000000000600B00                 public judge
.data:0000000000600B00 judge           proc far                ; CODE XREF: main+80↑p
.data:0000000000600B00                                         ; DATA XREF: main+16↑r ...
.data:0000000000600B00                 pop     rcx
.data:0000000000600B01                 test    ecx, r13d
.data:0000000000600B04                 test    [rcx-2Ch], r14d
.data:0000000000600B08                 retf    0EC49h
.data:0000000000600B08 judge           endp ; sp-analysis failed
.data:0000000000600B08
.data:0000000000600B08 ; ---------------------------------------------------------------------------
.data:0000000000600B0B                 db  6Ah ; j
.data:0000000000600B0C                 db 0CAh
.data:0000000000600B0D                 db  49h ; I
.data:0000000000600B0E                 db 0EDh
```


知道原理后我们可以开始做题了，第一种方法：直接嵌入脚本在让他把加密流程跑一遍得出真正judge代码逻辑，根据逻辑解密：

```
for ( i = 0; i <= 181; ++i )
  judge[i] ^= 0xCu;
```



然后C，P键扫描生成函数，如果有红色的行就用U设为未定义再C，P键即可：（C键是为了先转成数据，P键是在转成数据后创建函数）

```
.data:0000000000600B00 ; ---------------------------------------------------------------------------
.data:0000000000600B00
.data:0000000000600B00 ; char judge[182]
.data:0000000000600B00                 public judge
.data:0000000000600B00 judge:                          ; CODE XREF: main+80↑p
.data:0000000000600B00                                 ; DATA XREF: main+161↑r ...
.data:0000000000600B00                 pop     rcx
.data:0000000000600B01                 xchg    r13d, r10d
.data:0000000000600B04                 xor     byte ptr [rdi-2Dh], 0C2h
.data:0000000000600B09                 db      40h             ; PC/XT PPI port B bits:
.data:0000000000600B09                 out     61h, al         ; 0: Tmr 2 gate → OR 03H=spkr ON
.data:0000000000600B09                                         ; 1: Tmr 2 data ↗ AND 0fcH=spkr OFF
.data:0000000000600B09                                         ; 3: 1=read high switches
.data:0000000000600B09                                         ; 4: 0=enable RAM parity checking
.data:0000000000600B09                                         ; 5: 0=enable I/O channel check
.data:0000000000600B09                                         ; 6: 0=hold keyboard clock low
.data:0000000000600B09                                         ; 7: 0=enable kbrd
.data:0000000000600B0C                 mov     byte ptr [rbx+6Eh], 0DAh
.data:0000000000600B11                 pop     rax
.data:0000000000600B12                 cld
.data:0000000000600B13                 jl      short near ptr unk_600AF3
.data:0000000000600B15                 pop     rsp
.data:0000000000600B16                 stc
.data:0000000000600B17                 jg      short near ptr unk_600AEB
.data:0000000000600B19                 push    rax
.data:0000000000600B1A                 repne push 78F754D6h
00000B00 0000000000600B00: .data:judge (Synchronized with Hex View-1)
```

```
1  __int64 __fastcall judge(__int64 a1)
2  {
3    char v2[5]; // [rsp+8h] [rbp-20h] BYREF
4    char v3[9]; // [rsp+Dh] [rbp-1Bh] BYREF
5    int i; // [rsp+24h] [rbp-4h]
6
7    qmemcpy(v2, "fmcd", 4);
8    v2[4] = 127;
9    qmemcpy(v3, "k7d;V`;np", sizeof(v3));
10   for ( i = 0; i <= 13; ++i )
11     *(_BYTE *)(i + a1) ^= i;
12   for ( i = 0; i <= 13; ++i )
13   {
14     if ( *(_BYTE *)(i + a1) != v2[i] )
15       return 0LL;
16   }
17   return 1LL;
18 }
```

这里v2和v3在地址上是连在一起的，才有13的循环条件

函数逻辑很简单，flag是用户输入有关的生成型flag，对输入的简单的异或然后比较每个字符与预先给的字符串是否相等，等就返回1(true)，所以我们直接用预先给的字符串逆逻辑异或即可：

#key1="fmcd"

#key2=chr(0x7f)

#key3="k7d;V`;np"

#key4=key1+key2+key3

key="fmcd\x7Fk7d;V`;np"

flag=""

for i in range(14):

flag+=chr(ord(key[i])^i)

print(flag)

代码如上，犯下的第 3 个错误就是我一开始以为python会把\x7F判断成4个字符，结果是我多虑了，直接就是python自己会解析\x类型，太妙了，结果如图：

```
┌──(wdnmd㉿kali)-[~/桌面]
└─$ python 1.py
flag{n1c3_j0b}
```

第二种方法：IDA动态调试

直接远程调试，在第12行 if (v5 == 14 && (unsigned int)judge((__int64)s))处断下代码，断在这里的话IDA已经把judge解密完了，直接双击跟踪生成函数即可：

```
1  int __cdecl main(int argc, const char **argv, const char **envp)
2  {
3    char s[24]; // [rsp+0h] [rbp-20h] BYREF
4    int v5; // [rsp+18h] [rbp-8h]
5    int i; // [rsp+1Ch] [rbp-4h]
6
7    for ( i = 0; i <= 181; ++i )
8      *((_BYTE *)judge + i) ^= 0xCu;
9    printf("Please input flag:");
10   __isoc99_scanf("%20s", s);
11   v5 = strlen(s);
12   if ( v5 == 14 && (unsigned int)judge((__int64)s) )
13     puts("Right!");
14   else
15     puts("Wrong!");
16   return 0;
17 }
```



```
1  __int64 __fastcall judge(__int64 a1)
2  {
3    char v2[5]; // [rsp+8h] [rbp-20h] BYREF
4    char v3[9]; // [rsp+Dh] [rbp-1Bh] BYREF
5    int i; // [rsp+24h] [rbp-4h]
6
7    qmemcpy(v2, "fmcd", 4);
8    v2[4] = 127;
9    qmemcpy(v3, "k7d;V`;np", sizeof(v3));
10   for ( i = 0; i <= 13; ++i )
11     *(_BYTE *)(i + a1) ^= i;
12   for ( i = 0; i <= 13; ++i )
13   {
14     if ( *(_BYTE *)(i + a1) != v2[i] )
15       return 0LL;
16   }
17   return 1LL;
18 }
```


2021年10月广东强网杯，REVERSE的simplere：（迷宫结合、base64加密/解密算法、）

64位ELF文件，无壳，照例先运行一下程序，查看主要回显信息：

```
└─$ ./simple
your flag:666

nonono
```

照例扔入IDA中查看伪代码信息，有main函数看main函数：


```
__int64 __fastcall main(int a1, char **a2, char **a3)
{
  __int64 result; // rax
  char v4[33]; // [rsp+0h] [rbp-250h] BYREF
  char v5[15]; // [rsp+21h] [rbp-22Fh] BYREF
  char v6[256]; // [rsp+30h] [rbp-220h] BYREF
  char v7[256]; // [rsp+130h] [rbp-120h] BYREF
  void *src; // [rsp+230h] [rbp-20h]
  void *dest; // [rsp+238h] [rbp-18h]
  int v10; // [rsp+244h] [rbp-Ch]
  int j; // [rsp+248h] [rbp-8h]
  int i; // [rsp+24Ch] [rbp-4h]

  printf("your flag:");
  __isoc99_scanf("%s", v6);
  putchar(10);
  if ( v6[15] == '-' )
  {
    for ( i = 0; i <= 14; ++i )
      v5[i] = v6[i];
    for ( j = 0; j <= 17; ++j )
      v4[j] = v6[j + 16];
    v10 = sub_401192(v5);
    if ( v10 == 1 )
    {
      dest = mmap(0LL, 0x1000uLL, 7, 33, -1, 0LL);
      src = (void *)sub_40126F();
      memcpy(dest, src, 0x28CuLL);
      ((void (__fastcall *)(char *, char *, void *))dest)(v7, v4, &unk_404320);
      v10 = sub_401342(v7);
      if ( v10 )
        puts("congratulations!");
      else
        printf("nonono");
      result = 0LL;
    }
    else
    {
      printf("nonono");
      result = 0LL;
    }
  }
  else
  {
```

（这里积累第一个经验）

上图分析了前半部分，现在跟踪那个关键自定义函数v10 = sub_401192(v5);这是对前15个字符组成的迷宫进行操作。

点进去查看逻辑，是一个走迷宫，输入15步的wasd，要从A开始走，必须走到.上，最后必须走到B上。

根据11可以判断是二维数组，这也是我第一次学到迷宫的维数判断。因为一个走11步，另一个一步一步走，所以一维字符串最后得出二维迷宫，长11宽5的5*11型，s和w字符一下走11步就是上下移动，a和d一下走1步就是左右移动。也就是说走得多的就是行移动，走得少的就是列移动。

A********

.*...

...*.

*******...

******B

手扒得到ssddddwddddssas

```
__int64 __fastcall sub_401192(__int64 a1)
{
  int v1; // eax
  int i; // [rsp+Ch] [rbp-Ch]
  int v4; // [rsp+10h] [rbp-8h]
  int v5; // [rsp+14h] [rbp-4h]

  v5 = 1;
  v4 = 0;
  for ( i = 0; i <= 14; ++i )
  {
    v1 = *(unsigned __int8 *)(i + a1);
    if ( v1 == 'w' )
    {
      --v4;
    }
    else if ( *(unsigned __int8 *)(i + a1) <= (unsigned int)'w' )
    {
      if ( v1 == 's' )
      {
        ++v4;
      }
      else if ( *(unsigned __int8 *)(i + a1) <= (unsigned int)'s' )
      {
        if ( v1 == 'a' )
        {
          --v5;
        }
        else if ( v1 == 'd' )
        {
          ++v5;
        }
      }
    }
    if ( aAB[11 * v4 + v5] == 'B' )
      return 1LL;
    if ( aAB[11 * v4 + v5] != '.' )
      return 0LL;
  }
  return 0LL;
}
```

前面是对前15个字符的逐个字符比较

后面是对应的字符要满足条件才行，这里我是直接手扒的因为我比较菜，不会用算法来算。

```
// *A**********.***.....**.....***.*********..*********B**
```


然后上半部分条件就过了，开始分析后半部分：

```
  int i; // [rsp+24Ch] [rbp-4h]

  printf("your flag:");
  __isoc99_scanf("%s", v6);
  putchar(10);
  if ( v6[15] == '-' )
  {
    for ( i = 0; i <= 14; ++i )
      v5[i] = v6[i];
    for ( j = 0; j <= 17; ++j )
      v4[j] = v6[j + 16];
    v10 = sub_401192(v5);
    if ( v10 == 1 )
    {
      dest = mmap(0LL, 0x1000uLL, 7, 33, -1, 0LL);
      src = (void *)sub_40126F();
      memcpy(dest, src, 0x28CuLL);
      ((void (__fastcall *)(char *, char *, void *))dest)(v7, v4, &unk_404320);
      v10 = sub_401342(v7);
      if ( v10 )
        puts("congratulations!");
      else
        printf("nonono");
      result = 0LL;
    }
    else
    {
      printf("nonono");
      result = 0LL;
    }
  }
  else
  {
    printf("nonono");
    result = 0LL;
  }
  return result;
}
```

dest开辟空间，src关键自定义函数运算，然后复制给dest

又一个自定义函数

明显的自修改代码，可以动态调试，可以静态修补，后面的都是传入参数。

先跟踪分析src = (void *)sub_40126F();函数：

```
_BYTE *sub_40126F()
{
  _BYTE *v1; // [rsp+8h] [rbp-18h]
  int j; // [rsp+14h] [rbp-Ch]
  int i; // [rsp+18h] [rbp-8h]
  int v4; // [rsp+1Ch] [rbp-4h]

  v1 = malloc(0x200uLL);
  v4 = 0;
  for ( i = 0; i < 652; ++i )
  {
    if ( v4 == 64 )
      v4 = 0;
    v1[i] = byte_404080[i] ^ byte_404360[v4++];
  }
  for ( j = 0; j <= 63; ++j )
    byte_404320[j] ^= byte_404360[j];
  return v1;
}
```

两个原始数组异或，赋值给v1，v1再赋值给dest，这里的异或应该构成了dest的自修改函数了。

这里对byte_404320重新构造了，所以后面作为dest传入参数是修改过的。

（这里积累第二个经验）

这里的dest是在内存中的，这里静态修补我不太会，因为dest和src都是在栈中的所以直接用动态调试，用前面得出的前半部分搭配其它字符来运行ssddddwdddssas-666666666666666666：

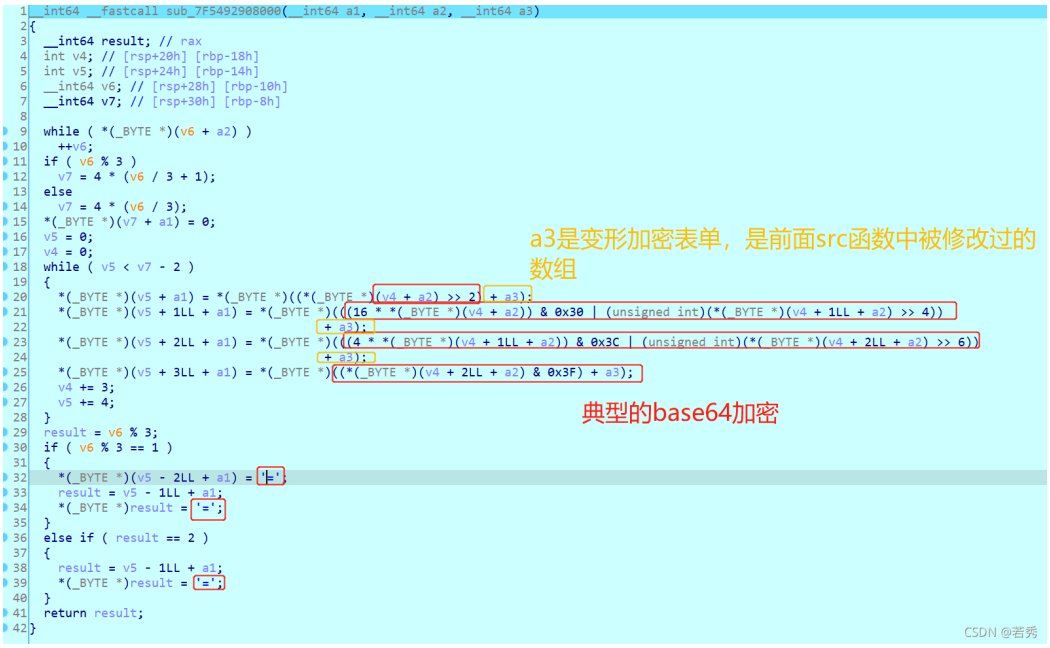
```
zero:00007F5492908000
zero:00007F5492908000 ; Attributes: bp-based frame
zero:00007F5492908000
zero:00007F5492908000 sub_7F5492908000 proc near
zero:00007F5492908000
zero:00007F5492908000 var_38= qword ptr -38h
zero:00007F5492908000 var_30= qword ptr -30h
zero:00007F5492908000 var_28= qword ptr -28h
zero:00007F5492908000 var_18= dword ptr -18h
zero:00007F5492908000 var_14= dword ptr -14h
zero:00007F5492908000 var_10= qword ptr -10h
zero:00007F5492908000 var_8= qword ptr -8
zero:00007F5492908000
zero:00007F5492908000 push    rbp
zero:00007F5492908001 mov     rbp, rsp
zero:00007F5492908004 mov     [rbp+var_28], rdi
zero:00007F5492908008 mov     [rbp+var_30], rsi
zero:00007F549290800C mov     [rbp+var_38], rdx
```

按P即可反汇编生成函数


```
zero:00007F5492908000
zero:00007F5492908000 ; Attributes: bp-based frame
zero:00007F5492908000
zero:00007F5492908000 sub_7F5492908000 proc near
zero:00007F5492908000
zero:00007F5492908000 var_38= qword ptr -38h
zero:00007F5492908000 var_30= qword ptr -30h
zero:00007F5492908000 var_28= qword ptr -28h
zero:00007F5492908000 var_18= dword ptr -18h
zero:00007F5492908000 var_14= dword ptr -14h
zero:00007F5492908000 var_10= qword ptr -10h
zero:00007F5492908000 var_8= qword ptr -8
zero:00007F5492908000
zero:00007F5492908000 push    rbp
zero:00007F5492908001 mov     rbp, rsp
zero:00007F5492908004 mov     [rbp+var_28], rdi
zero:00007F5492908008 mov     [rbp+var_30], rsi
zero:00007F549290800C mov     [rbp+var_38], rdx
```

按P即可反汇编生成函数

CSDN @若秀



所以这题就是base64变码加密了，继续跟踪最后的自定义函数v10 = sub_401342(v7);

密文也有了，可以写逻辑了，首先求出base64变形码表，从src = (void *)sub_40126F();处导出数组：

key2=[0x4D, 0x20, 0x07, 0x05, 0x43, 0x15, 0x7A, 0x73, 0x39, 0x01,

0x7F, 0x53, 0x66, 0x4E, 0x0D, 0x18, 0x60, 0x76, 0x75, 0x00,

0x58, 0x15, 0x00, 0x32, 0x68, 0x3F, 0x78, 0x7F, 0x7B, 0x64,

0x4E, 0x49, 0x0F, 0x2E, 0x3F, 0x0D, 0x0D, 0x0D, 0x64, 0x66,

0x61, 0x53, 0x06, 0x44, 0x34, 0x6E, 0x69, 0x2F, 0x20, 0x14,

0x37, 0x6A, 0x49, 0x55, 0x36, 0x37, 0x23, 0x23, 0x2A, 0x6B,

0x73, 0x06, 0x78, 0x0B]

key3=[0x3E, 0x7A, 0x40, 0x64, 0x27, 0x25, 0x48, 0x04, 0x4F, 0x63,

0x19, 0x60, 0x0B, 0x3A, 0x75, 0x5D, 0x11, 0x4E, 0x07, 0x44,

0x30, 0x4C, 0x4B, 0x06, 0x5F, 0x73, 0x0D, 0x1A, 0x38, 0x08,

0x34, 0x78, 0x45, 0x47, 0x58, 0x3B, 0x74, 0x7D, 0x2C, 0x2B,

0x4A, 0x3C, 0x29, 0x13, 0x01, 0x3F, 0x03, 0x61, 0x70, 0x52,

0x65, 0x09, 0x22, 0x00, 0x7F, 0x59, 0x6C, 0x77, 0x72, 0x3D,

0x32, 0x55, 0x41, 0x49]

for i in range(len(key3)):

key2[i]^=key3[i]

print(key2)

print(''.join(map(chr,key2))) #base64变表码

结果：

```
└─$ python 4.py
[115, 90, 71, 97, 100, 48, 50, 119, 118, 98, 102, 51, 109, 116, 120, 69, 113, 56, 114, 68,
104, 89, 75, 52, 55, 76, 117, 101, 67, 108, 122, 49, 74, 105, 103, 54, 121, 112, 72, 77, 43
, 111, 47, 87, 53, 81, 106, 78, 80, 70, 82, 99, 107, 85, 73, 110, 79, 84, 88, 86, 65, 83, 5
7, 66]
sZGad02wvbf3mtxEq8rDhYK47LueClz1Jig6ypHM+o/W5QjNPFRckUInOTXVAS9B
```

然后把变形码表替换传统base64加密码表，这里用的是我以前写过的base64python编码实现：（https://blog.csdn.net/xiao__1bai/article/details/120338971）

base64="sZGad02wvbf3mtxEq8rDhYK47LueClz1Jig6ypHM+o/W5QjNPFRckUInOTXVAS9B" #准备好base64的基表

def encryption(inputstring): #定义加密函数

ascii=['{:0>8}'.format(str(bin(ord(i))).replace('0b','')) for i in inputstring] #把每个输入字符保证8位一个，才能3*8变4*6。

#{:0>8}是右对齐8位然后左边补0,因为python是自己判断数据大小类型的，所以必须强制满足8位。bin转化二进制会带0b前缀，所以要用replace('0b','')去掉。

encrystr='' #while外的变量，返回base64加密后的字符串

equalnumber=0 #while外的变量，记录拆分后不足4的倍数时需要补齐的等号个数

while ascii:

subascii=ascii[:3] #用一个子列表subascii每次取输入的三位进行操作,前面操作后每位都是8位

while len(subascii)<3: #这里其实是最后一段截取中才会用上的，不满足3位时要单独取出，记录equalnumber数量用于后期补'='号，然后补齐8位的0免得干扰后面3*8拆分成4*6

equalnumber+=1 #计算要补‘=’的个数

subascii+=['0'*8] #补8个0来填充够3的倍数，这然后面就不会出错。

substring=''.join(subascii)#用substring合并subascii的3个8位，准备进行拆分操作

encrystringlist=[substring[x:x+6] for x in [0,6,12,18]] #开始进行3*8变4*6的拆分，每次拆分一组24位。

encrystringlist=[int(x,2) for x in encrystringlist] #把前面拆分的6位一组转成10进制，就不用进行位数补齐操作了，这是用来后面对应base64基表的下标。

if equalnumber:

encrystringlist=encrystringlist[0:4-equalnumber] #如果前面不足3字符补了0,比如2个8位字符16位，拆分后就要用3个6位共18位，所以有效位是4-equalnumber

encrystr+=''.join(base64[x] for x in encrystringlist) #这里encrystringlist已经在前面拆分成4*6且转换成10进制了，所以对应基表的下标。

ascii=ascii[3:] #每次向后取3个列表元素，对应while循环条件

encrystr+='='*equalnumber #因为前面encrystringlist[0:4-equalnumber]去掉了补0位，所以这里最后补齐'='号

return encrystr

def decryption(inputstring):

ascii=['{0:0>6}'.format(str(bin(base64.index(i))).replace('0b',''))for i in inputstring if i!='']#从加密字符中取除补位'='之外加密字符，即6位生成的base64基表下标的数，按6位一组排列，准备拆分

decrystr=''#准备while外的解密后的字符

equalnumber=inputstring.count('=')#这里计数补位的'='号的个数，后面不够8位时会根据'='号补加位数。

while ascii:

subascii=ascii[:4]#取加密字符的4个6位一组共24位准备拆分合并成3*8

substring=''.join(subascii)#先连成一串24位

if len(substring)%8!=0:

substring=substring[0:-1*equalnumber*2]

#截取到倒数第equalnumber*2个元素。对不足8位的组补位，因为加密时1个8位要来2个6位，两个'='号，截取到8位就是倒数第4位1*2*2。2个8位要3个6位，要一个'='号，截取到16位就是倒数第2位1*1*2。

decrystringlist=[substring[x:x+8] for x in [0,8,16]]#开始进行4*6变3*8的拆分，每次拆分4个6位一组24位。

decrystringlist=[int(x,2) for x in decrystringlist if x]#把前面拆分的8位一组转成10进制，用来对应十进制ASCII码，if x功能不清楚，但不可缺少，应该是要排除空格吧。

decrystr+=''.join([chr(x) for x in decrystringlist])#这里decrystringlist已经在前面拆分成3*8且转换成10进制了，现在转换成ASCII码。

ascii=ascii[4:]#每次向后取4个列表元素，对应while循环条件

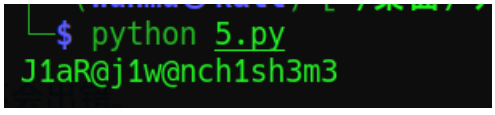
return decrystr

if __name__=="__main__":

#print(encryption('abcd'))

print(decryption('r60ihyZ/m4lseHt+m4t+mlkc'))

结果：



所以最终flag：

flag{ssddddwddddssas-J1aR@j1w@nch1sh3m3}

2021年9月广州羊城杯，REVERSE的RE-BabySmc：（函数积累、移位算法积累、IDA热键重新反汇编、

单层交叉引用查看、base64加密/解密算法、正向爆破）

下载附件，首先理解下SMC：

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。

而计算机病毒通常也会采用SMC技术动态修改内存中的可执行代码来达到变形或对代码加密的目的，从而躲过杀毒软件的查杀或者迷惑反病毒工作者对代码进行分析。

通常来说，SMC使用汇编去写会比较好，因为它涉及更改机器码，但SMC也可以直接通过C、C++来实现。

64位无壳，照例扔入IDA64中查看信息，有main函数看main函数：可以看出是乱码的，代码中也混杂有数据，这就是SMC了：

```
106    v42,
107    v43,
108    v44,
109    v45,
110    v46,
111    v47,
112    v48,
113    v49,
114    v50,
115    v51,
116    v52,
117    *((_QWORD *)&v52 + 1),
118    v53,
119    *((_QWORD *)&v53 + 1),
120    v54,
121    *((_QWORD *)&v54 + 1),
122    v55,
123    v56,
124    v57);
125  JUMPOUT(0x140001085i64);
126}
```

花指令SMC导致系统分析出错


```
.text:000000014000106B                mov     cs:lpAddress, rax
.text:0000000140001072                lea     rax, loc_140001D00
.text:0000000140001079                mov     cs:qword_14002AD88, rax
.text:0000000140001080                call    sub_140001E30
.text:0000000140001080 ; ---------------------------------------------------------------------------
.text:0000000140001085 byte_140001085  db 17h, 0D2h, 0DAh      ; DATA XREF: main:loc_140001064↑o
.text:0000000140001088                 dq 0AA4DD4ADAAE1D2D2h, 0F029AA4D53F2F029h, 52F2F029AA4DD3F2h
.text:0000000140001088                 dq 95CE90F2F229AA4Dh, 44A6FA52AACD79D0h, 0DA446EBA52AAD2DAh
.text:0000000140001088                 dq 0D2DA44D77A52AAD2h, 0AAD2DA449F3A52AAh, 0AAD2D2DAD0F3F65Ah
.text:0000000140001088                 dq 0AAD2D2DA50F3B65Ah, 0AAD2D2DAD1F3765Ah, 90D2D2DA51F3365Ah
.text:0000000140001088                 dq 90D2D2DAD6F376BEh, 0AAE155C5CE905C9Eh, 0AAE1C271AAE1D4ADh
.text:0000000140001088                 dq 0AA954CAADDCED46Ch, 90549E908A79D437h, 90D2D2429B955CCAh
.text:0000000140001088                 dq 8787878787878F27h, 90E56D90DC9E9087h, 0AD5C906D9E90649Eh
.text:0000000140001088                 dq 4BF8D2D28287F6AAh, 4BF8D2D2D2DA179Ch, 0FAA5DC90444BF8D4h
.text:0000000140001148                 db 0AAh, 0F6h, 44h, 9Ah, 2 dup(0D2h)
.text:0000000140001148 ; } // starts at 140001064
.text:000000014000114E ; __unwind { // __GSHandlerCheck
.text:000000014000114E byte_14000114E  db 90h                 ; DATA XREF: .pdata:000000014002D00C↓o
.text:000000014000114E                                        ; .pdata:000000014002D018↓o
.text:000000014000114F                 db 9Eh
.text:0000000140001150                 dq 93F3F19EB053F330h, 0D4C998D3F3B19EB0h, 0F2B727AAF044C998h
00000550 0000000140001150: main+150 (Synchronized with Hex View-1)
```

代码与数据混合

所以我们要判断它在哪里开始自修改，就是在哪里开始将这些数据转换成代码的，在这之前先了解一些SMC的基本知识和常用函数：

（出自https://zhuanlan.zhihu.com/p/66797526）

在 SMC 中将会对.text中某块内存的属性进行魔改，然后再进行下一步处理，例如进行解密。最后再执行。

VirtualProtect()

在 Windows 程序中使用了VirtualProtect()函数来改变虚拟内存区域的属性。

#include <Memoryapi.h>

BOOL VirtualProtect(

LPVOID lpAddress,

SIZE_T dwSize,

DWORD flNewProtect,

PDWORD lpflOldProtect

);

VirtualProtect()函数有4个参数，

lpAddress是要改变属性的内存起始地址，

dwSize是要改变属性的内存区域大小，

flAllocationType是内存新的属性类型，

lpflOldProtect内存原始属性类型保存地址。而flAllocationType部分值如下表。在 SMC 中常用的是 0x40。

Constant	value
PAGE_EXECUTE_READWRITE	0x40
PAGE_READONLY	0x02
PAGE_EXECUTE	0x10
PAGE_READWRITE	0x04
PAGE_EXECUTE_READ	0x20

mprotect()

在 Linux 程序中使用mprotect()函数来改变虚拟内存区域的属性。

#include <sys/mman.h>

int mprotect(void *addr, size_t len, int prot);

mprotect()系统调用修改起始位置为addr，长度为length字节的虚拟内存区域中分页上的保护。

addr取值必须为分页大小的整数倍，

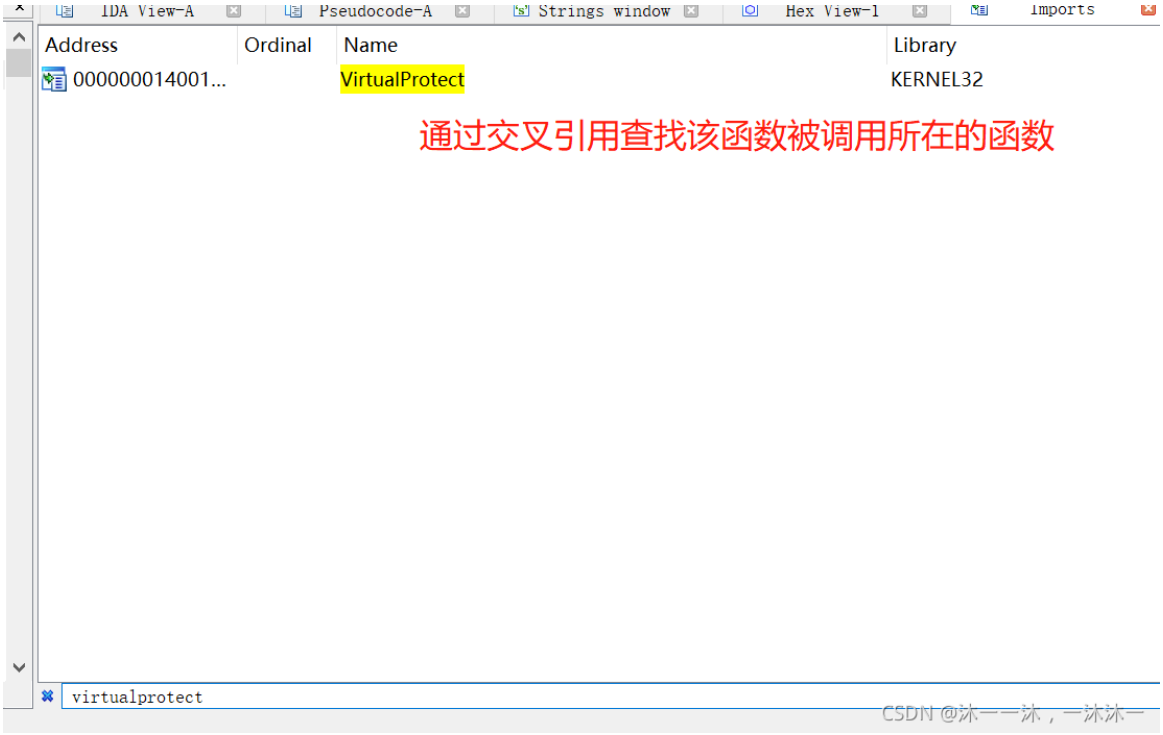
length会被向上舍入到系统分页大小的下一个整数倍。

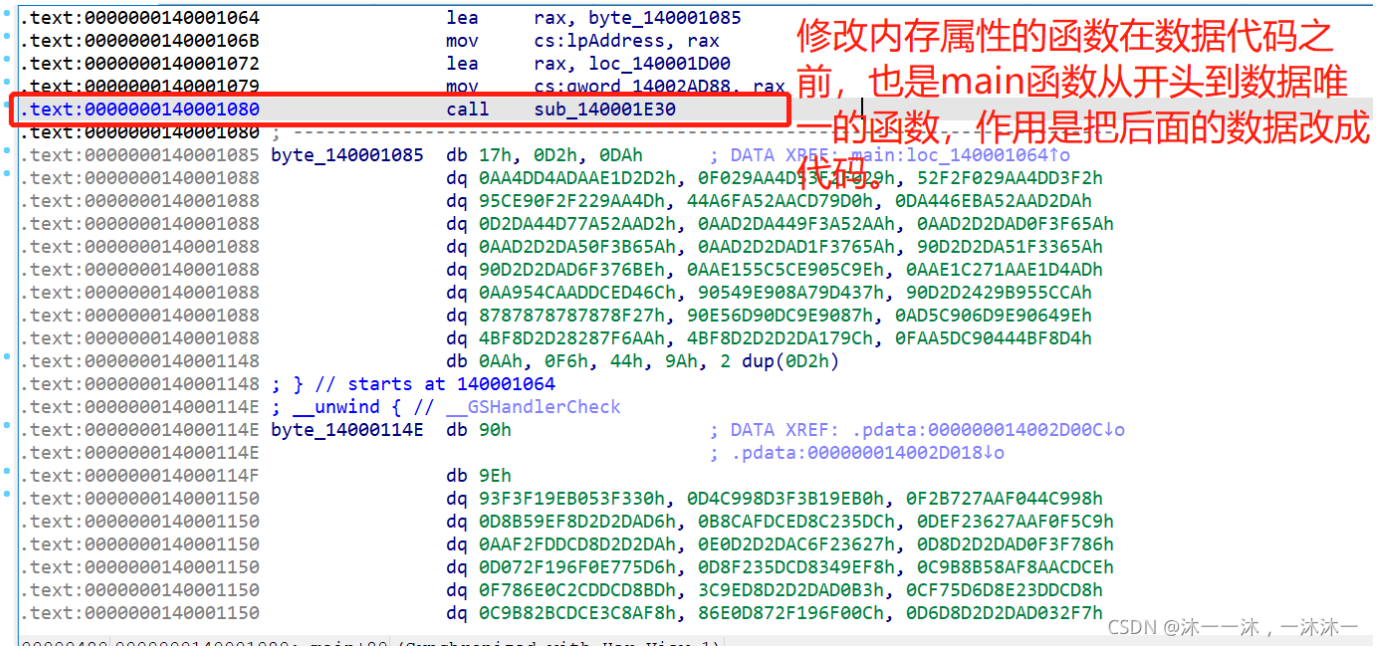
prot参数是一个位掩码。

prot一些属性如下：

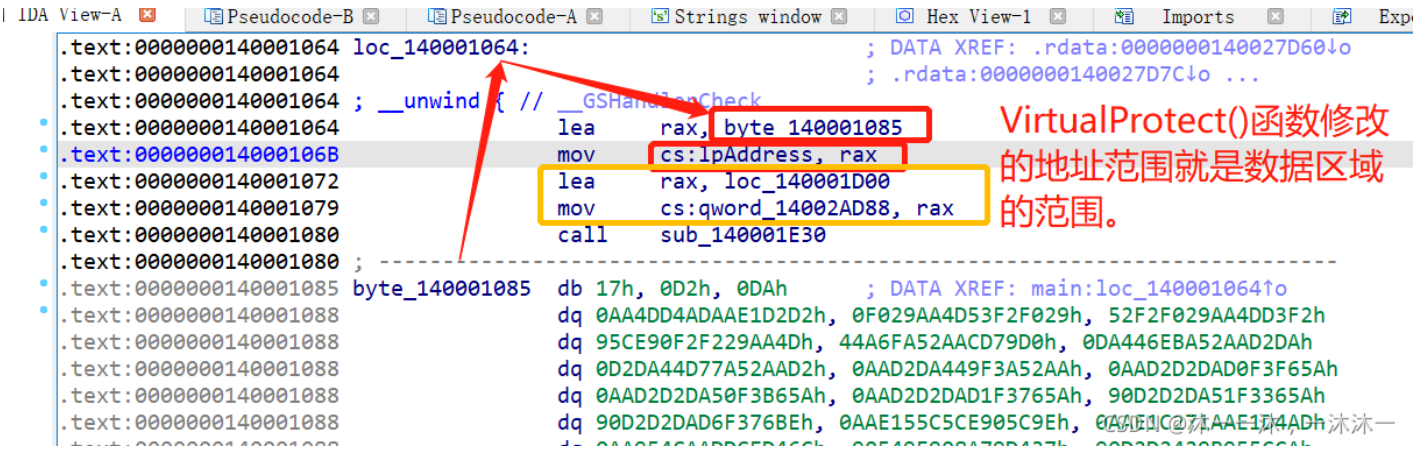
flags	description
PROT_READ	The memory can be read.
PROT_WRITE	The memory can be modified.
PROT_EXEC	The memory can be executed.

所以我们可以在IDA的import窗口找一下VirtualProtect()函数，经过不断的跟踪发现它是在数据代码前面，那就说得通了，因为代码中的数据要在执行前被正确修改成代码，所以自修改环节必须在代码中数据之前。





通过查看VirtualProtect()所在的sub_140001E30函数的伪代码，可以看到它的自修改逻辑。这里lpAddress和qword_14002AD88会在动态中动态赋值.text.代码段中数据的起始地址和末尾地址。逻辑也显而易见是__ROR1__(*v0, 3)循环右移3位后再异或0x5A。所以逆向逻辑就是异或0x5A后循环左移。



(这里积累第三个经验)

因为自修改的地方只有这里，所以我们有两种方法获取正常的代码。

1：动态调试：

在call sub_140001E30处下断点执行。然后注意一定要按下面顺序来按，首先是不能让IDA自己根据RIP生成代码，不然整个main函数结构就被拆分了，然后从选取从main函数头到第一个retn的代码段，因为IDA这时已经识别不了main函数块的结束部分了，然后选中Force强迫分析，并且不重新定义已存在的代码段，因为其它不用自修改的代码段都是正确的。

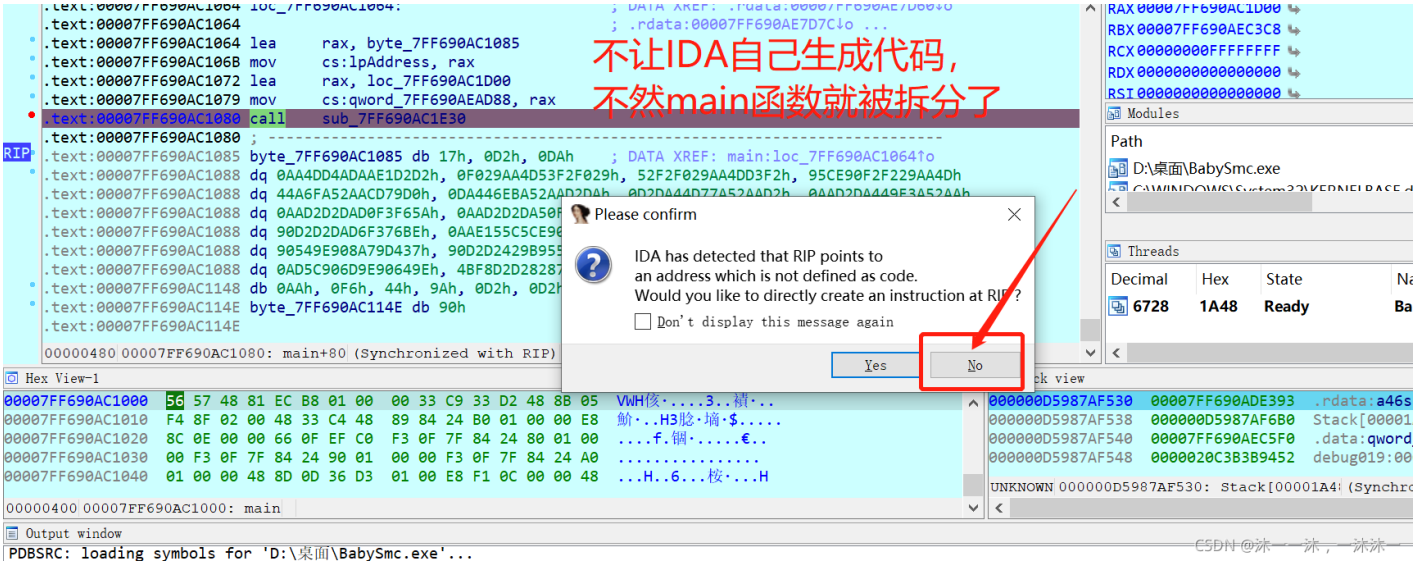
最后按F5就可以生成main函数的伪代码了。

```
.text:0000000140001063 ; } // starts at 140001000
.text:0000000140001064
.text:0000000140001064 loc_140001064:                          ; DATA XREF: .rdata:0000000140027D60↓o
.text:0000000140001064                                         ; .rdata:0000000140027D7C↓o ...
.text:0000000140001064 ; __unwind { // __GSHandlerCheck
.text:0000000140001064                 lea     rax, byte_140001085
.text:000000014000106B                 mov     cs:lpAddress, rax
.text:0000000140001072                 lea     rax, loc_140001D00
.text:0000000140001079                 mov     cs:qword_14002AD88, rax
.text:0000000140001080                 call    sub_140001E30
.text:0000000140001080 ; ---------------------------------------------------------------------------
.text:0000000140001085 byte_140001085  db 17h, 0D2h, 0DAh      ; DATA XREF: main:loc_140001064↑o
.text:0000000140001088                 dq 0AA4DD4ADAAE1D2D2h, 0F029AA4D53F2F029h, 52F2F029AA4DD3F2h
.text:0000000140001088                 dq 95CE90F2F229AA4Dh, 44A6FA52AACD79D0h, 0DA446EBA52AAD2DAh
.text:0000000140001088                 dq 0D2DA44D77A52AAD2h, 0AAD2DA449F3A52AAh, 0AAD2D2DAD0F3F65Ah
.text:0000000140001088                 dq 0AAD2D2DA50F3B65Ah, 0AAD2D2DAD1F3765Ah, 90D2D2DA51F3365Ah
.text:0000000140001088                 dq 90D2D2DAD6F376BEh, 0AAE155C5CE905C9Eh, 0AAE1C271AAE1D4ADh
.text:0000000140001088                 dq 0AA954CAADDCED46Ch, 90549E908A79D437h, 90D2D2429B955CCAh
.text:0000000140001088                 dq 8787878787878F27h, 90E56D90DC9E9087h, 0AD5C906D9E90649Eh
.text:0000000140001088                 dq 4BF8D2D28287F6AAh, 4BF8D2D2D2DA179Ch, 0FAA5DC90444BF8D4h
.text:0000000140001148                 db 0AAh, 0F6h, 44h, 9Ah, 2 dup(0D2h)
.text:0000000140001148 ; } // starts at 140001064
.text:000000014000114E ; __unwind { // __GSHandlerCheck
.text:000000014000114E byte_14000114E  db 90h                  ; DATA XREF: .pdata:000000014002D00C↓o
00000480 0000000140001080: main+80 (Synchronized with Hex View-1)
```

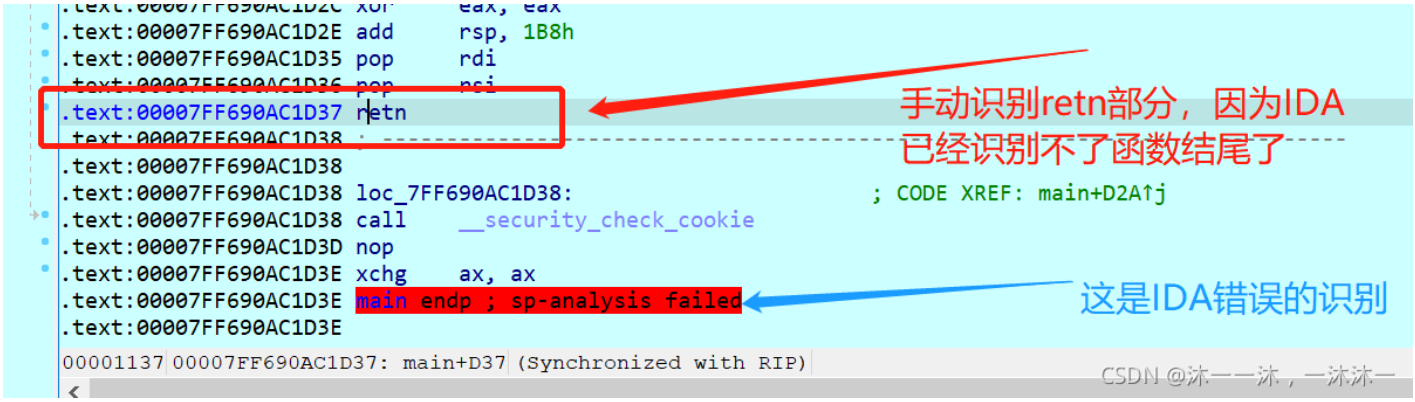
在内存修改处下断点

CSDN @沐一一沐，一沐沐一

不让IDA自己生成代码，
不然main函数就被拆分了



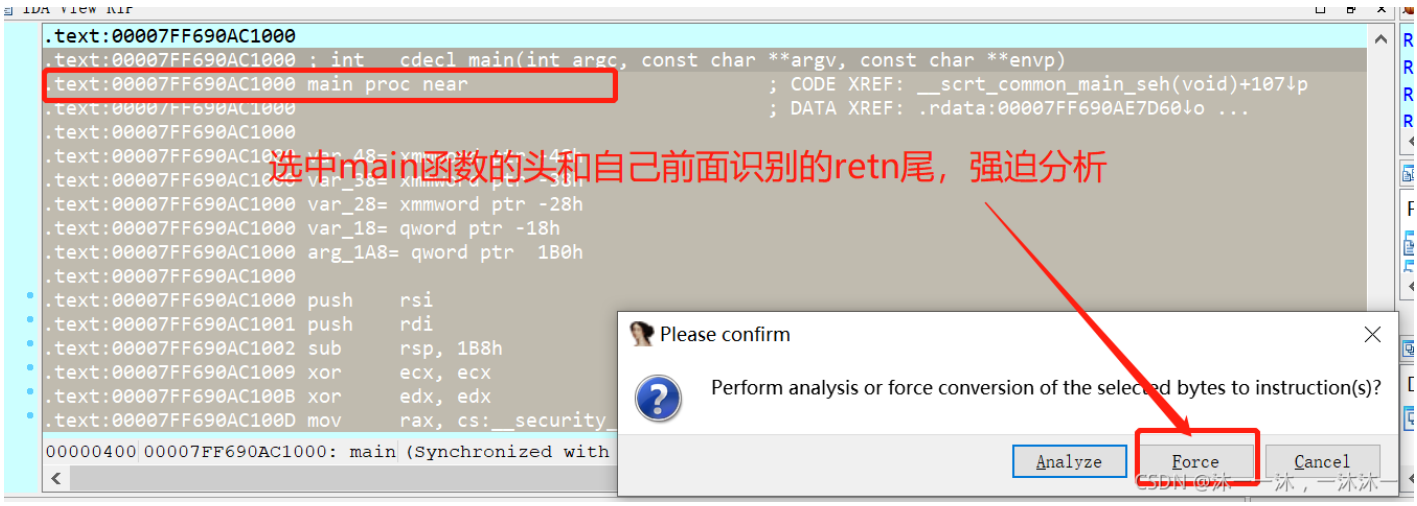
CSDN @沐一一沐，一沐沐一

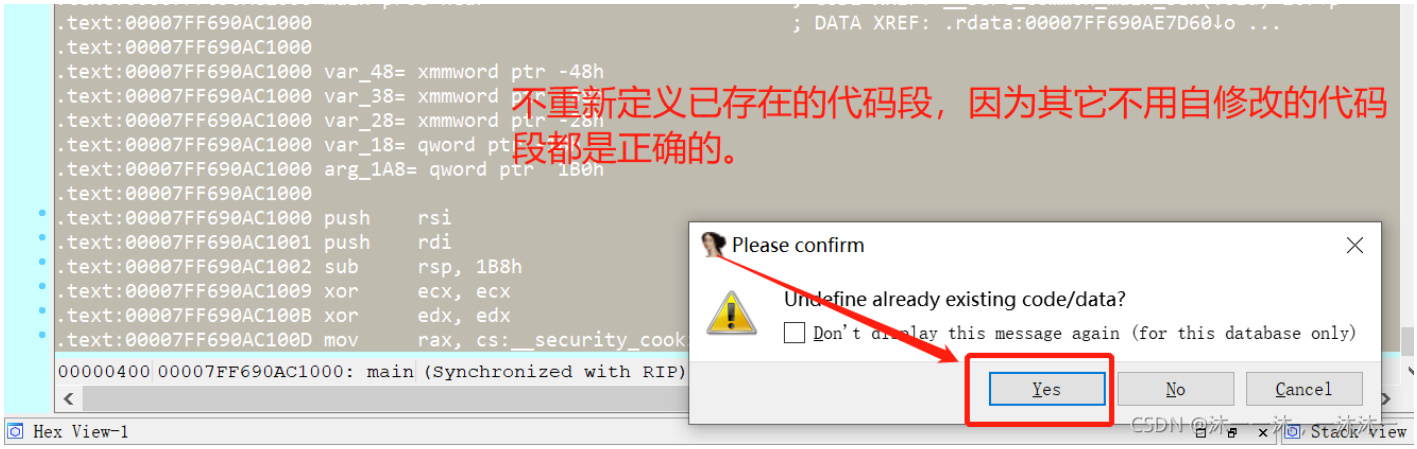


手动识别retn部分，因为IDA
已经识别不了函数结尾了

这是IDA错误的识别

CSDN @沐一一沐，一沐沐一






```
  __int128 v95[4]; // [rsp+140h] [rbp-88h]
  __m128i v96; // [rsp+180h] [rbp-48h] BYREF
  __int128 v97; // [rsp+190h] [rbp-38h]
  __int128 v98; // [rsp+1A0h] [rbp-28h]
  __int64 v99; // [rsp+1B0h] [rbp-18h]

  sub_7FF690AC1EB0(0i64, 0i64, envp);
  v96 = 0i64;
  v97 = 0i64;
  v98 = 0i64;
  sub_7FF690AC1D40("Input Your Flag : ");
  sub_7FF690AC1DC0("%46s", v96.m128i_i8);
  lpAddress = &loc_7FF690AC1085;
  qword_7FF690AEAD88 = (__int64)&loc_7FF690AC1D00;
  sub_7FF690AC1E30();
  v3 = 16i64;
  do
  {
    v93[v3 + 3] = 0i64;
    v93[v3 + 2] = 0i64;
    v93[v3 + 1] = 0i64;
    v93[v3] = 0i64;
    v3 -= 4i64;
  }
  while ( v3 * 16 );
  v95[0] = xmmword_7FF690ADE340;
  v95[1] = xmmword_7FF690ADE350;
  v95[2] = xmmword_7FF690ADE360;
  v95[3] = xmmword_7FF690ADE370;
  v5 = v4 + 0x80;
  v6 = v5 & 0xF;
  if ( !_BitScanForward((unsigned int *)&v8, (unsigned int)_mm_movemask_epi8(_mm_cmpeq_epi8((__m128i)0i64, v96)) >> v6) )
    v8 = sub_7FF690AC2340(v6, &v96.m128i_i8[v6]);
  v9 = v8;
  v10 = 0xAAAAAAAAAAAAAAABui64 * (unsigned __int128)(unsigned __int64)v8;
  v11 = *((_QWORD *)&v10 + 1) >> 1;
  if ( *((_QWORD *)&v10 + 1) >> 1 )
  {
    LODWORD(v12) = 0;
    LODWORD(v10) = 1;
    v14 = 0;
    v15 = 0;
    v16 = *((_QWORD *)&v10 + 1) >> 5;
    if ( *((_QWORD *)&v10 + 1) >> 5 )
```

反汇编且伪代码生成成功

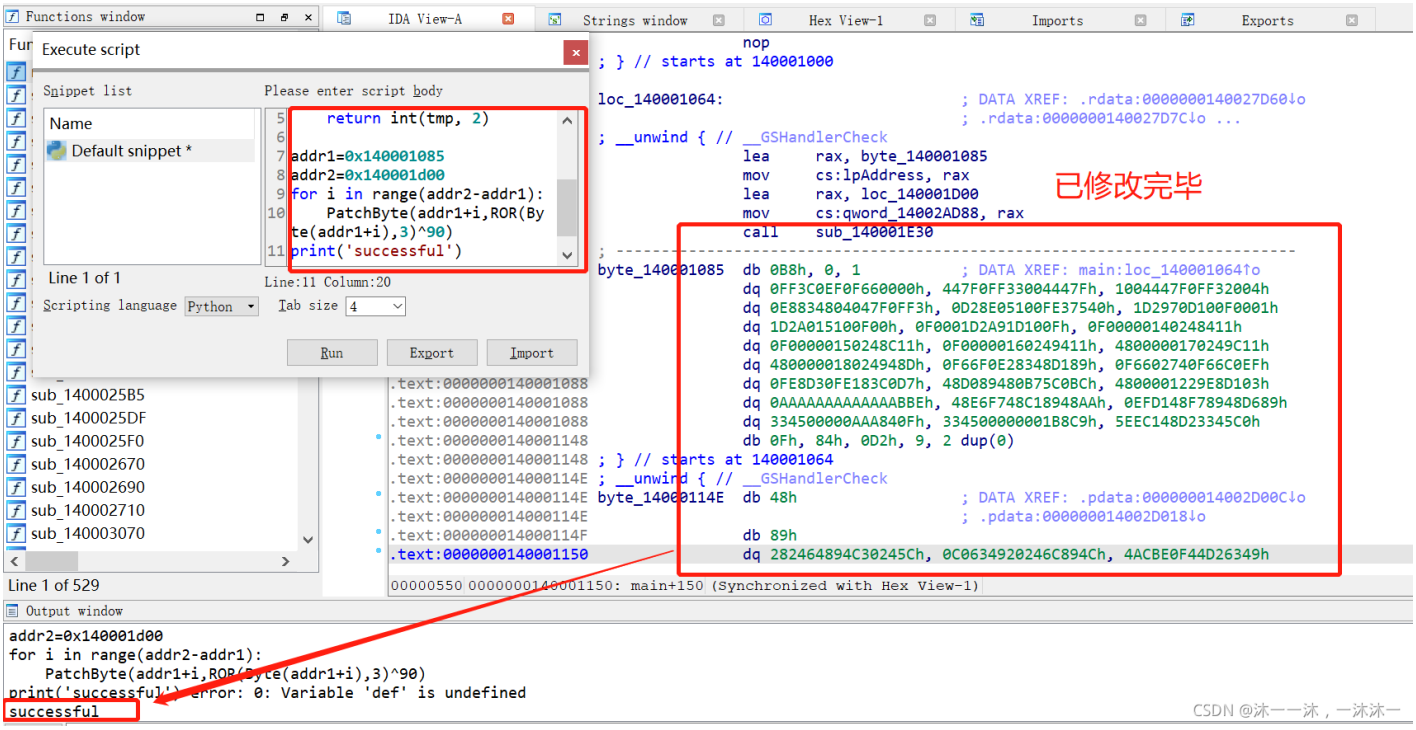
2：静态修补

前面说过自修改逻辑就是循环右移3位后再异或，那我们直接嵌入python脚本顺着它的逻辑修改代码也是可以的，首先附上python循环左右移动的代码：

```
def ROR(i,index):

tmp = bin(i)[2:].rjust(8,"0")

for _ in range(index):

tmp = tmp[-1] + tmp[:-1]

return int(tmp, 2)
```

```
addr1=0x140001085

addr2=0x140001d00

for i in range(addr2-addr1):

PatchByte(addr1+i,ROR(Byte(addr1+i),3)^90)

print('successful')
```

结果如图，然后照着前面方法强迫生成main函数伪代码即可：



然后就是分析函数主体代码量，代码量有点多，所以要挑重点看：

首先看最下面的判断逻辑。逻辑不难，因为出题重点在加密逻辑处，最后应该是加密后与v84字符串相等即可：

```
    v94[4 * v11] = 0;
  }
  v84 = "H>oQn6aqLr{DH6odhdm0dMe`MBo?lRglHtGPOdobDlknejmGI|ghDb<4";
  v85 = v94;
  while ( 1 )
  {
    v86 = (unsigned __int8)*v85 < (unsigned int)*v84;
    if ( *v85 != *v84 )
      break;
    if ( !*v85 )
      goto LABEL_21;
    v87 = v85[1];
    v86 = v87 < (unsigned int)v84[1];
    if ( v87 != v84[1] )
      break;
    v85 += 2;
    v84 += 2;
    if ( !v87 )
    {
LABEL_21:
      v88 = 0;
      goto LABEL_23;
    }
  }
  v88 = v86 ? -1 : 1;
LABEL_23:
  v89 = "No.\r\n";
  if ( !v88 )
    v89 = "Yes.\r\n";
  sub_140001D40(v89);
  sub_140006B38("pause");
  v90 = v99;
```

函数中用于最后比较的密文

比较成功就输出Yes

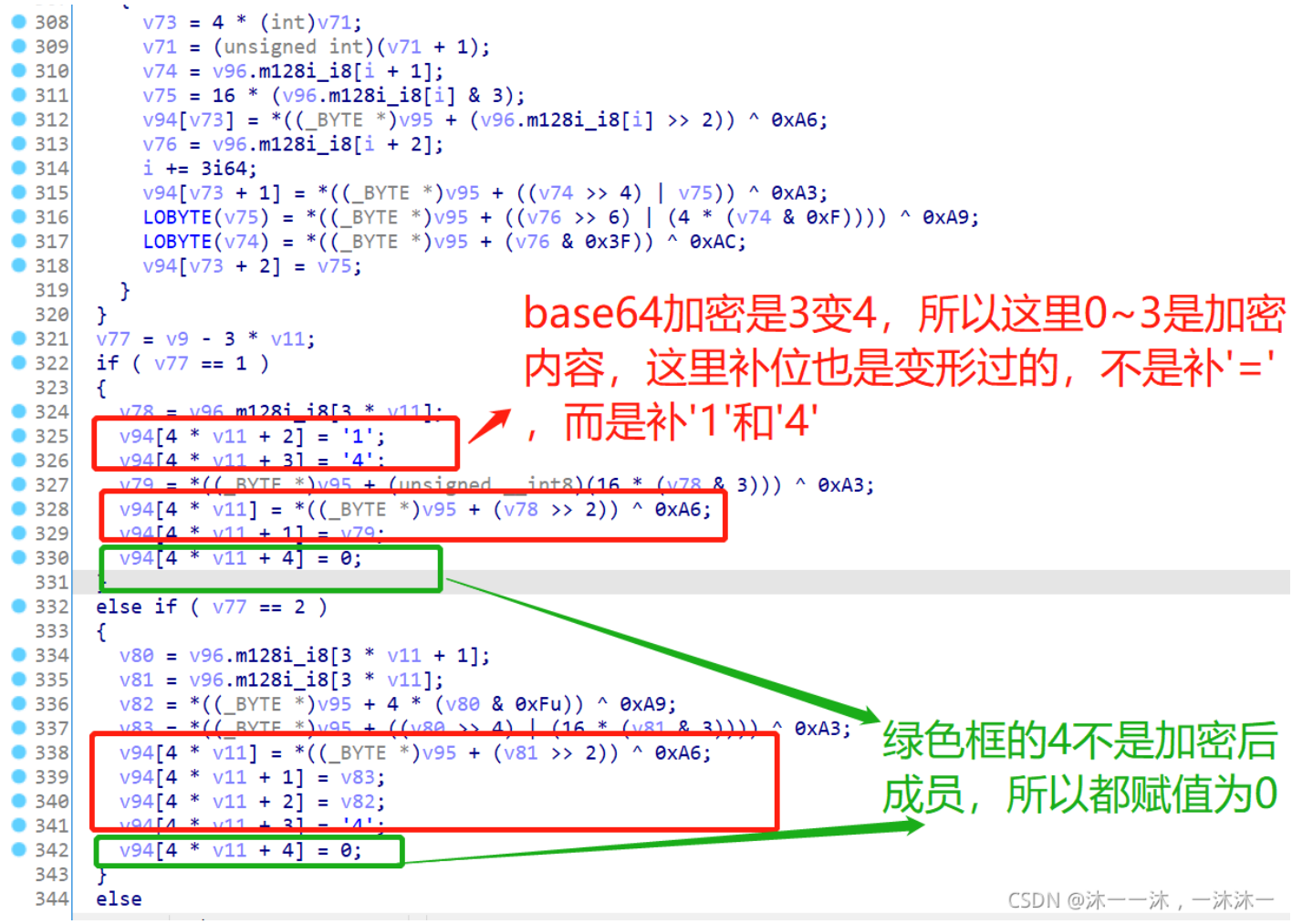
>>2 >>4 <<6 &0x3 &0xf &0x3f v77 == 1 v77 == 2看上去基本就是base64的加密3 * 8变4 * 6的实现了。

要说有不同就是这里的^0xA3 ^0xA6 ^0xA9 ^0xAc这些应该是出题着自己对base64加密的变形。

这里还要注意的是base64加密是3*8拆分成4*6的加密，所以红框中才是0~3才是加密后的数组，这个v94[4 * v11 + 4]并不属于加密数组的一部分。

关键就是这里是变形的base64加密，v77==1是指加密明文中最后一组不足3的倍数，而且只有1个。正常base64加密要补两个‘=’，但是这里补的是v94[4 * v11 + 2] = '1';和v94[4 * v11 + 3] = '4';

v77==2中补的是v94[4 * v11 + 3] = '4';结合比较的密文H>oQn6aqLr{DH6odhdm0dMe`MBo?lRglHtGPOdobDlknejmGl|ghDb<4可以知道最后的4是补上去的，解密的时候要单独抽出来，不然会造成干扰。



所以解密流程应该是这样，首先抽出H>oQn6aqLr{DH6odhdm0dMe`MBo?lRglHtGPOdobDlknejmGl|ghDb<4的4，然后进行异或操作来得到base64第一层base64加密后的密文：

key1="H>oQn6aqLr{DH6odhdm0dMe`MBo?lRglHtGPOdobDlknejmGl|ghDb<"#4

list1=[]

list2=[0xA6,0XA3,0XA9,0XAC]

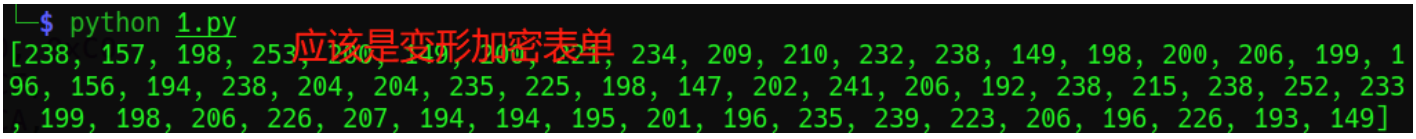
flag=""

for i in range(len(key1)):

list1.append(ord(key1[i])^list2[i%4])

print(list1)

结果中可以发现，这并不是base64基本的64个字符的ASCII码，所以可以判断这里是进行了base64的基本加密表单变换：

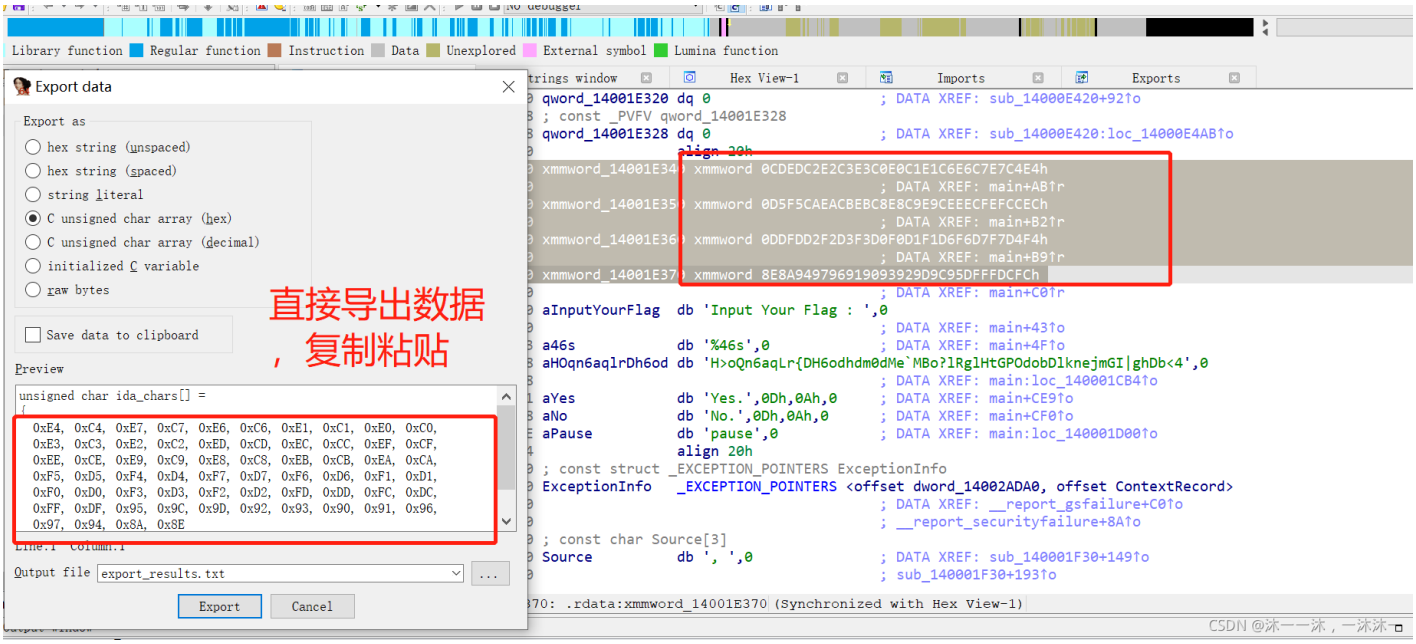


所以回到IDA找一下前面的表单，然后用shift+E dump下来(这个操作也是新学的，比我用IDA内嵌脚本快多了)

```
106  lpAddress = &loc_140001085;
107  qword_14002AD88 = (__int64)&loc_140001D00;
108  sub_140001E30();
109  v3 = 16i64;
110  do
111  {
112    v93[v3 + 3] = 0i64;
113    v93[v3 + 2] = 0i64;
114    v93[v3 + 1] = 0i64;
115    v93[v3] = 0i64;
116    v3 -= 4i64;
117  }
118  while ( v3 * 16 );
119  v95[0] = xmmword_14001E340;
120  v95[1] = xmmword_14001E350;
121  v95[2] = xmmword_14001E360;
122  v95[3] = xmmword_14001E370;
123  v5 = v4 + 0x80;
124  v6 = v5 & 0xF;
125  if ( !_BitScanForward((unsigned int *)&v8, (unsigned int)_mm_movemask_epi8(_mm_cmpeq_epi8((__m128i)0i6
126    v8 = sub_140002340(v6, &v96.m128i_i8[v6]);
127  v9 = v8;
128  v10 = 0xAAAAAAAAAAAAAAABui64 * (unsigned __int128)(unsigned __int64)v8;
129  v11 = *((_QWORD *)&v10 + 1) >> 1;
130  if ( *((_QWORD *)&v10 + 1) >> 1 )
131  {
132    LODWORD(v12) = 0;
```

i64是一个IDA类型，这里0i64就是0.

新的加密基本表单



然后至此我们就知道了总得逻辑了，新的base64加密基本字符表单---->加密明文并且多个异或操作------>最后一组不足一位且补了个'4'------>与最后密文比较

那么解密逻辑就是：最后密文单独抽出'4'并且异或操作得到一层密文------->根据新的加密表单进行base64逻辑的解密即可---->原base64解密中的'='号替换成'4'

下面是几种不同的解题方法：

第一种方法：

直接用原base64解密代码替换表单，'='操作替换成'4'，注意这里的'4'是后面加的，不在字符异或范围，所以要抽出来异或后再加进去。（能够单独抽出4再加进去是因为4的高位为0，单独操作根本不影响）

key1="H>oQn6aqLr{DH6odhdm0dMe`MBo?lRglHtGPOdobDlknejmGI|ghDb<"#4

list1=[]

```
list2=[0xA6,0XA3,0XA9,0XAC]
flag=""
for i in range(len(key1)):
list1.append(ord(key1[i])^list2[i%4])
print(list1)
for a in list1:
flag+=chr(a)
flag+='4'
list2=[0xE4, 0xC4, 0xE7, 0xC7, 0xE6, 0xC6, 0xE1, 0xC1, 0xE0, 0xC0,
0xE3, 0xC3, 0xE2, 0xC2, 0xED, 0xCD, 0xEC, 0xCC, 0xEF, 0xCF,
0xEE, 0xCE, 0xE9, 0xC9, 0xE8, 0xC8, 0xEB, 0xCB, 0xEA, 0xCA,
0xF5, 0xD5, 0xF4, 0xD4, 0xF7, 0xD7, 0xF6, 0xD6, 0xF1, 0xD1,
0xF0, 0xD0, 0xF3, 0xD3, 0xF2, 0xD2, 0xFD, 0xDD, 0xFC, 0xDC,
0xFF, 0xDF, 0x95, 0x9C, 0x9D, 0x92, 0x93, 0x90, 0x91, 0x96,
0x97, 0x94, 0x8A, 0x8E]
letters=""
for i in list2:
letters+=chr(i)
def decryption(inputString): #把传统base64解密的逻辑替换了加密基本表单元素，把'='替换成了'4'
# 对前面不是“=”的字节取索引，然后转换为2进制
asciiList = ['{:0>6}'.format(str(bin(letters.index(i))).replace('0b', ''))
for i in inputString if i != '4']
outputString = ''
#补齐“=”的个数
equalNumber = inputString.count('4')
while asciiList:
tempList = asciiList[:4]
#转换成2进制字符串
tempString = ''.join(tempList)
# 对没有8位2进制的字符串补够8位2进制
if len(tempString) % 8 != 0:
```

```
tempString = tempString[0:-1*equalNumber*2]

# 4个6字节的二进制 转换 为三个8字节的二进制

tempStringList = [tempString[x:x+8] for x in [0, 8, 16]]

# 二进制转为10进制

tempStringList = [int(x, 2) for x in tempStringList if x]

#连接成字符串

outputString += ''.join([chr(x) for x in tempStringList])

asciiList = asciiList[4:]

#print(output_str)

return outputString

print(decryption(flag))
```

第二种方法：正向爆破，也是借鉴修改了别人的脚本，但是我写了基本就是我的了~ (笑 ^ ~ ^)：

```
table=[0xE4, 0xC4, 0xE7, 0xC7, 0xE6, 0xC6, 0xE1, 0xC1, 0xE0, 0xC0,

0xE3, 0xC3, 0xE2, 0xC2, 0xED, 0xCD, 0xEC, 0xCC, 0xEF, 0xCF,

0xEE, 0xCE, 0xE9, 0xC9, 0xE8, 0xC8, 0xEB, 0xCB, 0xEA, 0xCA,

0xF5, 0xD5, 0xF4, 0xD4, 0xF7, 0xD7, 0xF6, 0xD6, 0xF1, 0xD1,

0xF0, 0xD0, 0xF3, 0xD3, 0xF2, 0xD2, 0xFD, 0xDD, 0xFC, 0xDC,

0xFF, 0xDF, 0x95, 0x9C, 0x9D, 0x92, 0x93, 0x90, 0x91, 0x96,

0x97, 0x94, 0x8A, 0x8E]

key1="H>oQn6aqLr{DH6odhdm0dMe`MBo?lRglHtGPOdobDlknejmGl|ghDb<4"

str_len=int(len(key1)/4-1) #最后一组4是补上去的，所以要特殊对待，这里str_len是13,/会得出小数，所以用int转换成整型。

flag=""

print(str_len)

for sub in range(str_len): #最外层循环是截取要比较的4位字符，放在最外面就才可以以4个为一组来比较。

for a in range(32,127):

for b in range(32,127): #完全的base64加密3*8变4*6的拆分

for c in range(32,127):

q=table[a >> 2] ^ 0xA6

w=table[((a & 0x3) << 4) | (b >> 4) ] ^ 0xA3

e=table[((b & 0xf) << 2) | (c >> 6) ] ^ 0xA9
```

r=table[c & 0x3f] ^ 0xAC

tempstr=chr(q)+chr(w)+chr(e)+chr(r)

if tempstr==key1[sub*4:(sub+1)*4]:

flag+=chr(a)

flag+=chr(b)

flag+=chr(c)

for i in range(32,127): #对待最后一组有补数的，因为只有一个补数，所以是2个明文的2*8对应3*6，剩下一个补位。

for j in range(32,127):

k=table[i >> 2] ^ 0xA6

l=table[((i & 0x3) << 4) | (j >> 4)] ^ 0xA3

m=table[(j & 0xf) <<2] ^ 0xA9 #从第三个开始的就不用写了，因为是0,不影响

tempstr=chr(k)+chr(l)+chr(m)+'4'

if tempstr=='Db<4':

flag+=chr(i)

flag+=chr(j)

print(flag)

第三种方法：下标对应法，这就要了解base64加密解密的本质了。

base64加密是3*8变4*6，获取的4个6位数对应着0~64内的范围，而base64的基本字符表ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/不过是0~64范围内对应的映射下标而已。

解密的时候也是用每个加密字符在0 ~64范围的数来拆分解密，关键就是这个解密时的6位数是怎么找的呢？是通过base64.index('加密字符')来找对应的0 ~ 64的下标。

所以加密字符表单的作用只是用来根据下标来映射6位数的0~64的范围而已，本质是0 ~64的下标。

那么我们就可以通过一一对应的方法来把题目中新的加密表单的下标来对应原生的base64加密下标了，因为base64在线解密工具只能通过base64.index('加密字符')来找0 ~ 64的下标。

脚本如下，注意抽出'4'来，并且在最后要把'='作为替换加上去，因为base64解密必须是4的倍数。

key1="H>oQn6aqLr{DH6odhdm0dMe`MBo?lRglHtGPOdobDlknejmGl|ghDb<"#4

list1=[]

list2=[0xA6,0XA3,0XA9,0XAC]

flag=""

for i in range(len(key1)):

list1.append(ord(key1[i])^list2[i%4])

```
print(list1)

for a in list1:

flag+=chr(a)

print(flag)

list2=[0xE4, 0xC4, 0xE7, 0xC7, 0xE6, 0xC6, 0xE1, 0xC1, 0xE0, 0xC0,

0xE3, 0xC3, 0xE2, 0xC2, 0xED, 0xCD, 0xEC, 0xCC, 0xEF, 0xCF,

0xEE, 0xCE, 0xE9, 0xC9, 0xE8, 0xC8, 0xEB, 0xCB, 0xEA, 0xCA,

0xF5, 0xD5, 0xF4, 0xD4, 0xF7, 0xD7, 0xF6, 0xD6, 0xF1, 0xD1,

0xF0, 0xD0, 0xF3, 0xD3, 0xF2, 0xD2, 0xFD, 0xDD, 0xFC, 0xDC,

0xFF, 0xDF, 0x95, 0x9C, 0x9D, 0x92, 0x93, 0x90, 0x91, 0x96,

0x97, 0x94, 0x8A, 0x8E]

base64="ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"

secret=""

for i in list1:

k=list2.index(i)

secret+=base64[k]

secret+='='

print(secret)
```

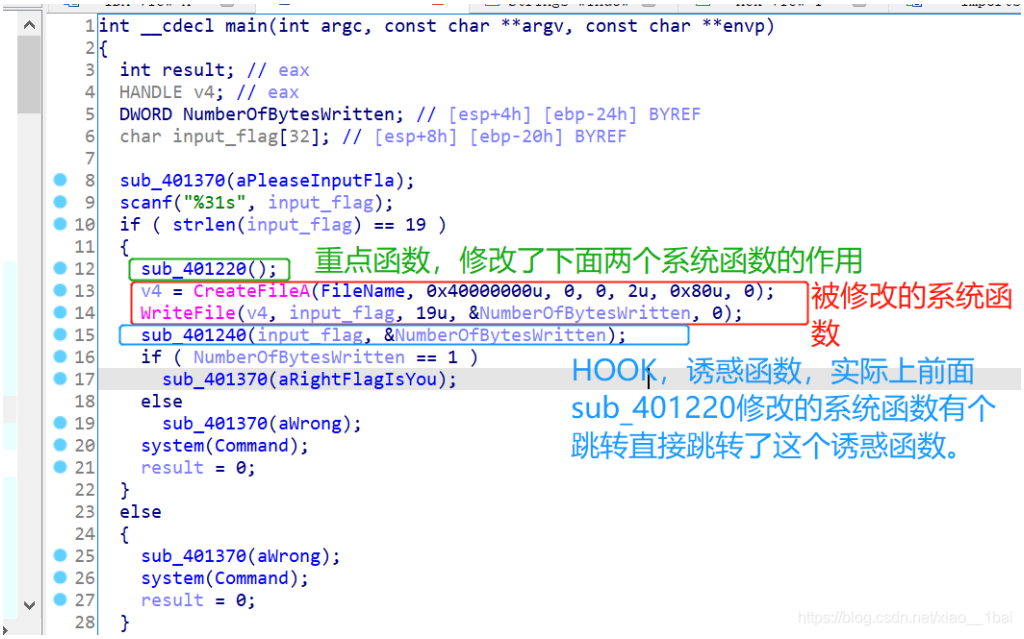
系统函数函数自修改：（HOOK，通常两次修改系统函数，一次改成自定义机器码，一次改回正常）

Hook 技术又叫做钩子函数，在系统没有调用该函数之前，钩子程序就先捕获该消息，钩子函数先得到控制权，这时钩子函数既可以加工处理（改变）该函数的执行行为，还可以强制结束消息的传递。简单来说，就是把系统的程序拉出来变成我们自己执行代码片段。

攻防世界EASYHOOK：（非预期行为、函数积累、手动机器码）

这是我第一次遇到HOOK类型的题目，我很是兴奋，我也花了相当长的时间才把该HOOK流程全部弄懂，希望能给日后更多经验。

直接入重点，打开IDA查看main函数，具体分析我写在代码里：




```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  int result; // eax
  HANDLE v4; // eax
  DWORD NumberOfBytesWritten; // [esp+4h] [ebp-24h] BYREF
  char input_flag[32]; // [esp+8h] [ebp-20h] BYREF

  sub_401370(aPleaseInputFla);
  scanf("%31s", input_flag);
  if ( strlen(input_flag) == 19 )
  {
    sub_401220();
    v4 = CreateFileA(FileName, 0x40000000u, 0, 0, 2u, 0x80u, 0);
    WriteFile(v4, input_flag, 19u, &NumberOfBytesWritten, 0);
    sub_401240(input_flag, &NumberOfBytesWritten);
    if ( NumberOfBytesWritten == 1 )
      sub_401370(aRightFlagIsYou);
    else
      sub_401370(aWrong);
    system(Command);
    result = 0;
  }
  else
  {
    sub_401370(aWrong);
    system(Command);
    result = 0;
  }
```

sub_401370(aPleaseInputFla);

scanf("%31s", input_flag);

if (strlen(input_flag) == 19)

{

sub_401220(); //未知名函数，一开始当然不会注意

v4 = CreateFileA(FileName, 0x40000000u, 0, 0, 2u, 0x80u, 0); //创建文件函数

WriteFile(v4, input_flag, 19u, &NumberOfBytesWritten, 0); //写入函数，这些系统函数通常不是重点，如果影响理解代码的话直接查API即可， WriteFile(句柄, 写入字符串, 写入字节, 指向写入直接的指针, 0)

sub_401240(input_flag, &NumberOfBytesWritten);

if (NumberOfBytesWritten == 1) //判断函数

sub_401370(aRightFlagIsYou);

else

sub_401370(aWrong);

system(Command);

result = 0;

}

else

{

sub_401370(aWrong);

system(Command);

result = 0;

}

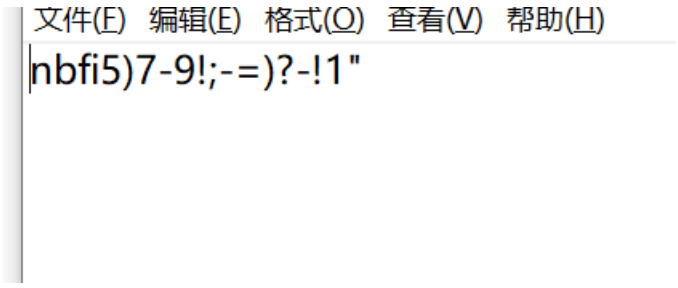
return result;

}

好了，疑惑开始，首先当然直接跟踪sub_401240函数：然后就犯下第一个错误：(被HOOK了)：

查看该函数代码让我疑惑的事情发生了，v4[a1 - v4 + result] == v4[result] 这条神仙代码无解啊，a1是我输入的字符串地址，作为下标运算就算了，result还是从0开始的，完全走不通啊！(PS：这题还算好了，给了我一个走不通的HOOK，要是这个HOOK还是走得通的话就更花时间了！)

然后我又看了一下生成的Your_input文件，？？？生成的不是我写入的：



然后就去查资料了，期间学到了很多东西，我们先用静态调试细致分析：首先这一系列非预期行为基本说明了存在其他操作，那前面非系统函数就只剩下sub_401220()了，跟踪加代码分析：

int sub_401220()

{

HMODULE v0; // eax

DWORD v2; // eax

v2 = GetCurrentProcessId(); //系统函数，获取进程ID，就是当前程序的ID

hProcess = OpenProcess(0x1F0FFFu, 0, v2); //系统函数，返回现有进程对象的句柄。

v0 = LoadLibraryA(LibFileName); //系统函数，将指定的可执行模块映射到调用进程的地址空间，这里LibFileName双击跟踪存入的是kernel32.dll模块，就是导入了它

*(_DWORD *)WriteFile_0 = GetProcAddress(v0, ProcName); //系统函数，返回指定的导出动态链接库（DLL）函数的地址。 ProcName存放的是WriteFile函数名，也就是导入WriteFile函数。

lpAddress = *(LPVOID *)WriteFile_0; //获取WriteFile函数地址

if (!*(_DWORD *)WriteFile_0) //无用函数，因为kernel32.dll模块的WriteFile函数一定存在

return sub_401370(&unk_40A044); //sub_401370函数类型puts函数，是通过主函数分析得到的，&unk_40A044处的字符串是获取原API入口地址出错，不用管他。

unk_40C9B4 = *(_DWORD *)lpAddress; //这里获取WriteFile函数的地址

*((_BYTE *)&unk_40C9B4 + 4) = *((_BYTE *)lpAddress + 4); //这里地址后四位也保持一样，不知道有什么用

byte_40C9BC = 0351; //这里连同第二句是我犯下的第二个错误：这里转十六进制不是E9，是JMP的机器码指令，而不一开始并没有机器码的相关知识。

dword_40C9BD = (char *)sub_401080 - (char *)lpAddress - 5; //这里是一个偏移地址，而且还是满足HOOK的连续地址。之所以这样写是因为汇编语言JMP address被编译后会变成机器指令码，E9 偏移地址，偏移地址=目标地址-当前地址-5(jmp和其后四位地址共占5个字节)。所以前面直接用E9，这里直接用偏移地址就省去编译生成机器码那一步。这也是HOOK的原型。

return sub_4010D0(); //返回一个函数，继续跟踪，因为前面并没有什么修改程序的行为，只是创造了一个JMP指令而已。

}

跟踪sub_4010D0()函数：(这里替换了WriteFile函数的5字节为JMP跳转指令)

BOOL sub_4010D0()

{

DWORD v1; // [esp+4h] [ebp-8h] BYREF

DWORD flOldProtect; // [esp+8h] [ebp-4h] BYREF

v1 = 0;

VirtualProtectEx(hProcess, lpAddress, 5u, 4u, &flOldProtect); //这里犯下第三个错误就是我一开始看不懂这三行代码

函数理解：

VirtualProtectEx(进程，修改地址，修改区域大小，[修改其中权限，1，2，4对应执行，写，读]，一个保存修改前权限的变量)，

所以这三行作用是对lpAddress所存储的地址处进行了5字节的权限修改操作，先将关键地址区改成可读写再往此处写入前面E9 偏移量地址处的5个字节（即上面的跳转指令），最后恢复权限，完成修改。

WriteProcessMemory(hProcess, lpAddress, &byte_40C9BC, 5u, 0); //在原WriteFile函数地址处覆盖写入5字节的JMP手动机器码

return VirtualProtectEx(hProcess, lpAddress, 5u, flOldProtect, &v1); //恢复权限

所以这里执行到WriteFile函数的时候就会变成一个跳转语句，跳转到目标地址sub_401080处，双击跟踪该函数：

int __stdcall sub_401080(HANDLE hFile, LPCVOID input_flag, DWORD nNumberOfBytesToWrite, LPDWORD lpNumberOfBytesWritten, LPOVERLAPPED lpOverlapped)

{

int v5; // ebx

v5 = sub_401000((int)input_flag, nNumberOfBytesToWrite); //真正加密函数，后面分析

sub_401140(); //重写WriteFile函数地址处的内存，和前面4010D0处函数的三句类似,只是代码WriteProcessMemory(hProcess, lpAddress, &unk_40C9B4, 5u, 0);写入的地址为&unk_40C9B4,这的确是WriteFile函数地址,就是为了下面那个WriteFile函数不受影响,真正是写入文件函数

WriteFile(hFile, input_flag, nNumberOfBytesToWrite, lpNumberOfBytesWritten, lpOverlapped); //真正的写入函数

if (v5)

*lpNumberOfBytesWritten = 1; //这里*lpNumberOfBytesWritten已经赋值为1了，也是全局变量，你会猛的发现主函数的sub_401240(input_flag, &NumberOfBytesWritten);这个幌子函数就算对不上数组字符也不会设置*lpNumberOfBytesWritten = 0；所以只要这里真正加密的对了，后面错误退出*lpNumberOfBytesWritten还是等于1.

return 0;

}

分析真正加密函数 sub_401140()：

int __cdecl sub_401000(int input_flag, int _19)

{

char i; // al

char v3; // bl

char v4; // cl

int v5; // eax

for (i = 0; i < _19; ++i)

{

if (i == 18)

{

*(_BYTE *)(input_flag + 18) ^= 19u; // 这里犯下第五个错误，写脚本时这种单独的if语句应该直接在外面单独成行，不然就要多写几个elif语句了，界面就会很乱！

}

else

{

if (i % 2)

v3 = *(_BYTE *)(i + input_flag) - i;

else

v3 = *(_BYTE *)(i + input_flag + 2);

*(_BYTE *)(i + input_flag) = i ^ v3;

}

}

v4 = 0;

if (_19 <= 0) // 不会小于等于，所以不会在这里返回

return 1;

v5 = 0;

while (aAjygkfm[v5] == *(_BYTE *)(v5 + input_flag)) //这里犯下第四个错误，前面的加密逻辑中我们没有可以参照的字符串，因为flag是推出来的，而真正的字符串在后面，所以逆向逻辑时首先要找到非输入的字符串才行！这里双击跟踪 aAjygkfm等于ajygkFm.\x7f_~-SV{8mLn

{

v5 = ++v4;

if (v4 >= _19)

return 1;

}

return 0;

}

所以最后脚本：

```
flag=list("------------------") #这是学别人的，因为我们要的是19个元素的数组，所以List函数很不错喔

print(len(flag))

key1="ajygkFm.\x7f_~-SV{8mLn"

flag[18]=chr(ord(key1[18])^19)

for i in range(18):

v3=ord(key1[i])^i

if i%2==1: #把18的判断条件抽出去后里面就只有一层条件了，简便很多。

flag[i]=chr(v3+i)

else:

flag[i+2]=chr(v3)

print(''.join(flag))
```

结果，改第一个字符为f即可：

```
19
-lag{Ho0k_w1th_Fun}
```

引用别人博客的一句话：

现在程序流程就很明朗了

粗略来看程序流程是CreateFileA->(lpAddress里存的指令)WriteFile->sub_401240

但是在经过sub_401220()的处理以后，变成了CreateFileA->(lpAddress里存的指令)sub_401080->sub_401240。

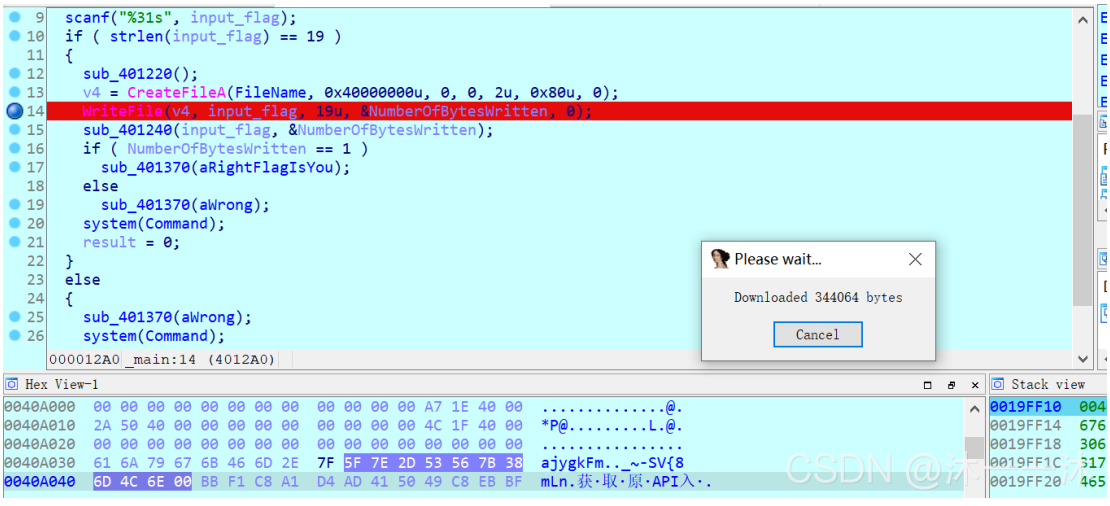
那么最后的sub_401240又干了什么事呢，分析一下代码：(这里我重命名了一些变量名)：

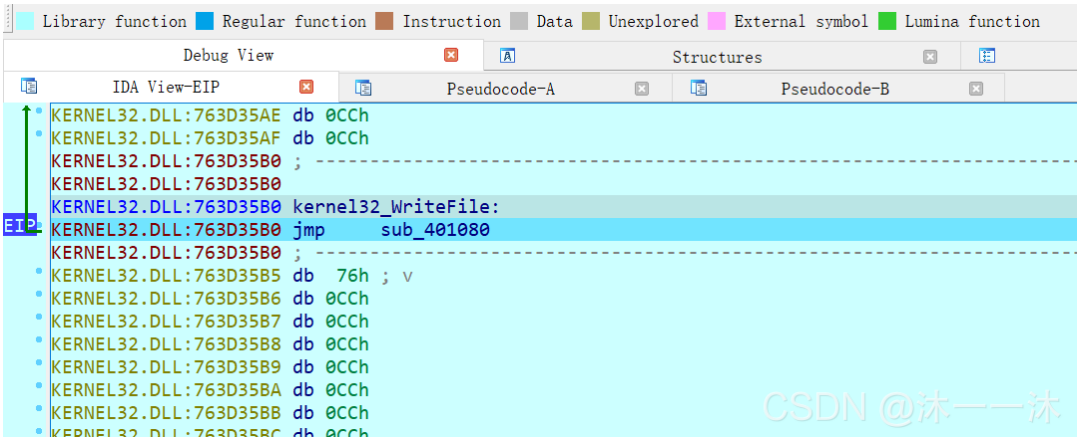
```
1  int __cdecl sub_401240(const char *input_flag, _DWORD *nNumberOfBytesToWrite)
2  {
3    int result; // eax
4    unsigned int v3; // kr04_4
5    char v4[24]; // [esp+Ch] [ebp-18h] BYREF
6
7    result = 0;
8    strcpy(v4, "This_is_not_the_flag");
9    v3 = strlen(input_flag) + 1;
10   if ( (int)(v3 - 1) > 0 )
11   {
12     while ( v4[input_flag - v4 + result] == v4[result] )
13     {
14       if ( ++result >= (int)(v3 - 1) )
15       {
16         if ( result == 21 )
17         {
18           result = (int)nNumberOfBytesToWrite;
19           *nNumberOfBytesToWrite = 1;
20         }
21         return result;
22       }
23     }
24   }
25   return result;
26 }
```


这里最有意思的是这个函数完全符合就会执行nNumberOfBytesToWrite = 1;，但是不符合也没有nNumberOfBytesToWrite = 0;的操作啊，我们前面真加密函数的时候就赋值*nNumberOfBytesToWrite = 1;了，也就是说不管这里对还是错都会输出you flag is right啊！而且动态调试的时候你会发现，这里循环执行一次就退出来了，因为while (v4[input_flag - v4 + result] == v4[result])这个代码根本不合逻辑。

最后动态调试就不想搞了：

我记得IDA动态运行到主函数WriteFile处的时候用F7单步执行会看到一个jmp的指令，继续动态到sub_401240函数的时候也是正如我说的循环运行第一次就退出来了，但是照样输出you flag is right。





攻防世界梅津美治郎：（函数积累、非预期行为、环境准备函数、IDA动态调试、int3断点考察）

32位无壳，先运行一下程序看看主要回显信息：

照例扔入IDA32中查看伪代码，有main函数看main函数：

```
61  v60 = &argc;
62  puts_space();
63  puts(
64    "             .     \n"
65    "            _|_    ROBOTIC AUTHENTICATION SYSTEM\n"
66    "    /\\/\\  (. .)  /\n"
67    "    `||'   |#|  \n"
68    "     ||__.-\"-\"-.___    \n"
69    "     `---| . . |--.\\  \n"
70    "         | : : |  ,||,\n"
71    "         `..-..'  \\/\\/\n"
72    "          || ||   \n"
73    "          || ||    \n"
74    "         |__|__|  \n");
75  v20 = 1337;
76  v21 = 1617194321;
77  v22 = 1679910002;
78  v23 = 1935963503;
79  v24 = 1684632865;
80  v25 = 1936221985;
81  v26 = 1361147250;
82  v27 = 1987211872;
83  v28 = 996504430;
84  v29 = 1;
85  v18[0] = 1617194321;
86  v18[1] = 1679910002;
87  v18[2] = 1935963503;
88  v18[3] = 1684632865;
```

一堆变量和数字，如果不是地址连续小数组或冗余算法的话这些就是关键变量，后续操作中会进一步处理。

Pseudocode-A | Pseudocode-B | IDA View-A | Strings window | Hex View-1 | Imports | Exp

```
115  v14[0] = 1869557876;
116  v14[1] = 1718432615;
117  v15 = 3;
118  v42 = 'osdj';
119  v43 = '32md';
120  v44 = 'mme/';
121  v45 = '\x01?\x01\x01';
122  v13[0] = 'osdj';
123  v13[1] = '32md';
124  v13[2] = 'mme/';
125  v13[3] = '\x01?\x01\x01';
126  v46 = 'Wee@';
127  v47 = 'nubd';
128  v48 = 'Deds';
129  v49 = 'qdby';
130  v50 = 'onhu';
131  v51 = 'eo`I';
132  v52 = '\x01sdm';
133  v12[0] = 'Wee@';
134  v12[1] = 'nubd';
135  v12[2] = 'Deds';
136  v12[3] = 'qdby';
137  v12[4] = 'onhu';
138  v12[5] = 'eo`I';
139  v12[6] = '\x01sdm';
140  v53 = 561278552;
141  v54 = 795830390;
142  v55 = 1869496865;
```

又是一堆变量和数字，后续中会有进一步处理

(这里积累第一个经验)

第一、二副图中可以看到前面一堆数字，肯定有用的，冗余代码不会长这个样子。然后就是第三幅图中两个系统函数GetModuleHandleA和GetProcAddress

查了手上所有的手册，一个获取模块名，一个获取模块名内函数。

GetModuleHandle

VB声明	
Declare Function GetModuleHandle Lib "kernel32" Alias "GetModuleHandleA" (ByVal lpModuleName As String) As Long	
说明	
获取一个应用程序或动态链接库的模块句柄	
返回值	
Long，如执行成功成功，则返回模块句柄。零表示失败。会设置GetLastError	
参数表	
参数	类型及说明
lpModuleName	String，指定模块名，这通常是与模块的文件名相同的一个名字。例如，NOTEPAD.EXE程序的模块文件名就叫作NOTEPAD
注解	
只有在当前进程的场景中，这个句柄才会有效	

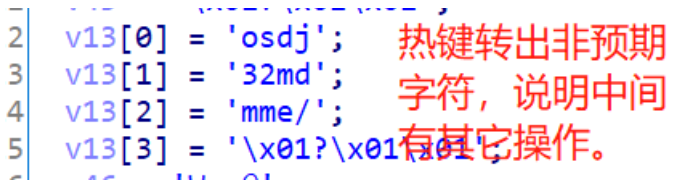
top

GetProcAddress函数返回指定的导出动态链接库（DLL）函数的地址。

FARPROC GetProcAddress

HMODULE //处理DLL模块
【HMODULE】,
LPCSTR *【lpProcName】* //函数名称
);

所以前面的大量静态数字赋值就是用来间接构成字符串的，可是我用热键R转字符也没看出是什么函数啊：



(这里积累第二个经验)

现在回顾一下以前的积累，凡是与预期不符的结果，都是在中间有其他的操作，动调后发现，中间有个不起眼的sub_401500(0);函数进行前面字符串的修改：

```
152  v10[5] = 1919905384;
153  v11 = '\x01 ';
154  v9 = '\x059';
155  strcpy(Str2, "r0b0RUlez!");
156  dword_40AD94 = (int)&v9;
157  dword_40ADA0 = (int)&v20;
158  dword_40AD8C = (char *)v18;
159  dword_40AD90 = (char *)v16;
160  off_40AD98 = (int)v14;
161  lpProcName = (LPCSTR)v12;
162  lpModuleName = (LPCSTR)v13;
163  Buffer = (char *)v10;
164  sub_401500(0);
165  v3 = lpProcName;
166  v4 = GetModuleHandleA(lpModuleName);
167  v5 = (void (__stdcall *)(HMODULE, LPCSTR))GetProcAddress(v4, v3);
168  v5((HMODULE)1, (LPCSTR)sub_40157F);
169  puts(dword_40AD8C);
170  scanf("%20s", input_flag);
171  if ( !strcmp(input_flag, Str2) )
00000E67 _main:154 (401A67)
```

上面变量到下面系统函数调用中都没有用户输入的地方，说明前面都是程序环境准备部分。这里有个关键自定义函数sub_401500函数就起到了字符串转化的作用。

双击跟踪进入该函数：

Pseudocode-A | Pseudocode-B | IDA View-A | Strings window | Hex View-1

```
1  int __cdecl sub_401500(int a1)
2  {
3    int result; // eax
4    _BYTE *i; // [esp+1Ch] [ebp-Ch]
5
6    if ( a1 <= 9 )
7      return sub_401500(a1 + 1);
8    for ( i = (_BYTE *)dword_40AD94; ; ++i )
9    {
10     result = dword_40ADA0;
11     if ( (unsigned int)i >= dword_40ADA0 )
12       break;
13     *i ^= 1u;
14   }
15   return result;
16 }
```

递归

从94到A0的字符串操作

这里把前面乱码的变量数字值转换成有意义的字符串，后面两个系统函数也就调用得通了。

(这里积累第三个经验)

动态调试后发现是kernel32.dll模块库的AddVectoredExceptionHandler函数，然后又把手册翻了个遍，大概是捕抓异常，然后用回调函数处理，所以v5((HMODULE)1, (LPCSTR)sub_40157F);就是捕抓异常然后给第一个处理函数sub_40157F处理。

```
161  lpProcName = (LPCSTR)v12;
162  lpModuleName = (LPCSTR)v13;
163  Buffer = (char *)v10;
164  sub_401500(0);
165  v3 = lpProcName;
166  v4 = GetModuleHandleA(lpModuleName);
167  v5 = (void (__stdcall *)(HMODULE, LPCSTR))GetProcAddress(v4, v3);
168  v5((HMODULE)1, (LPCSTR)sub_40157F);
169  puts(dword_40AD8C);
170  scanf("%20s", input_flag);
171  if ( !strcmp(input_flag, Str2) )
```

kernel32.dll

AddVectoredExceptionHandler函数给了v5

AddVectoredExceptionHandler函数传入参数，即异常和回调函数

```
[00003DBC]:0060FD7F db    0
[00003DBC]:0060FD80 aAddvectoredexc db 'AddVectoredExceptionHandler',0
[00003DBC]:0060FD9C aKernel32Dll db 'kernel32.dll',0        ; DATA XREF: .bss:lpModuleName↑o
[00003DBC]:0060FDA9 db    0
[00003DBC]:0060FDAA db  3Eh ; >
[00003DBC]:0060FDAB db    0
```

动态调试出来的两个字符串

继续往下走，下面是一个输入和比较，str2是明文，所以直接复制粘贴r0b0RUlez!通过第一关。然后就程序回显出进please enter the second password!，找了strings窗口也没找到对应字符串在哪里，后来才发现下面还有个sub_4015EA(0);函数，真的，这种函数个体又小，又只有0参数，很容易让人忽略啊~

```
153  v11 = '\x01 ';
154  v9 = '\x059';
155  strcpy(Str2, "r0b0RUlez!");
156  dword_40AD94 = (int)&v9;
157  dword_40ADA0 = (int)&v20;
158  dword_40AD8C = (char *)v18;
159  dword_40AD90 = (char *)v16;
160  off_40AD98 = (int)v14;
161  lpProcName = (LPCSTR)v12;
162  lpModuleName = (LPCSTR)v13;
163  Buffer = (char *)v10;
164  sub_401500(0);
165  v3 = lpProcName;
166  v4 = GetModuleHandleA(lpModuleName);
167  v5 = (void (__stdcall *)(HMODULE, LPCSTR))GetProcAddress(v4, v3);
168  v5((HMODULE)1, (LPCSTR)sub_40157F);
169  puts(dword_40AD8C);
170  scanf("%20s", input_flag);
171  if ( !strcmp(input_flag, Str2) )
172  {
173    puts("You passed level1!");
174    sub_4015EA(0);
175  }
176  return 0;
177}
```

第一关直接明文比较

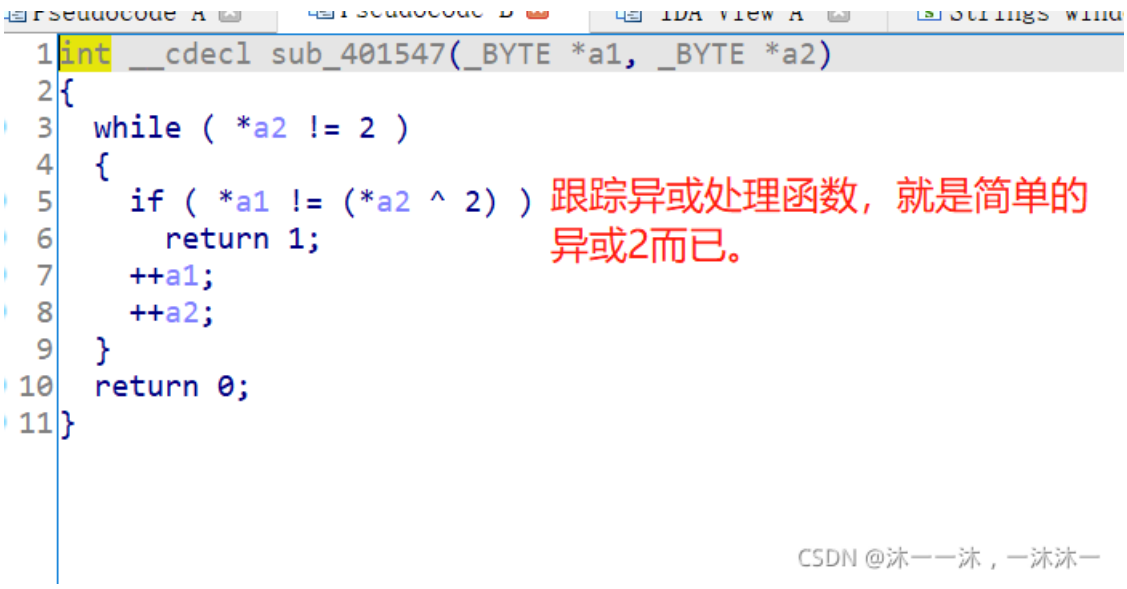
通过第一关后应该往下找才对，sub_4015EA就是通过第一关后的关键函数。

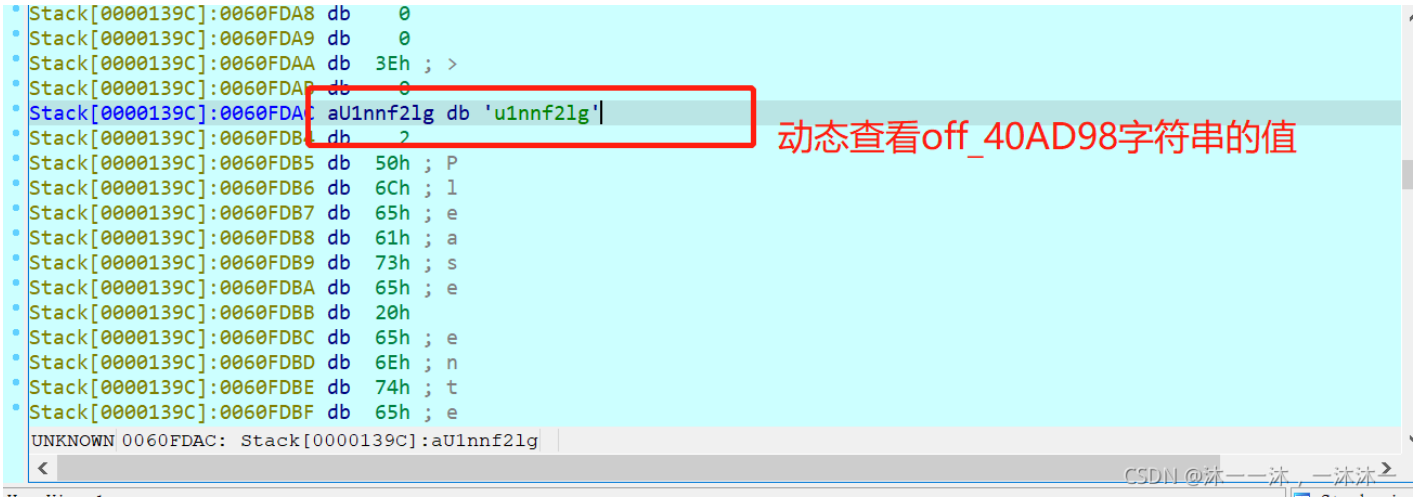
(这里积累第四个经验)

跟踪sub_4015EA(0);函数，一个递归打印dword_40AD90数组，这个数组在前面动调中是**please enter the second password!**字符串，关键是后面的__debugbreak函数，这会抛出int3断点异常，下面附上别人透彻的分析：

这里有个debugbreak函数，这函数的话，也就是相当于一个int 3指令，引发一个中断，把执行权移交给调试器，如果没有调试器，那就移交给其他异常处理回调函数，如果都没有，那么程序就自己断下来，这里就是为了触发回调函数，如果没有调试器附加，那么debugbreak产生的异常会被AddVectoredExceptionHandler添加的回调函数捕获来处理，这里添加的回调函数也就是sub_40157F函数。

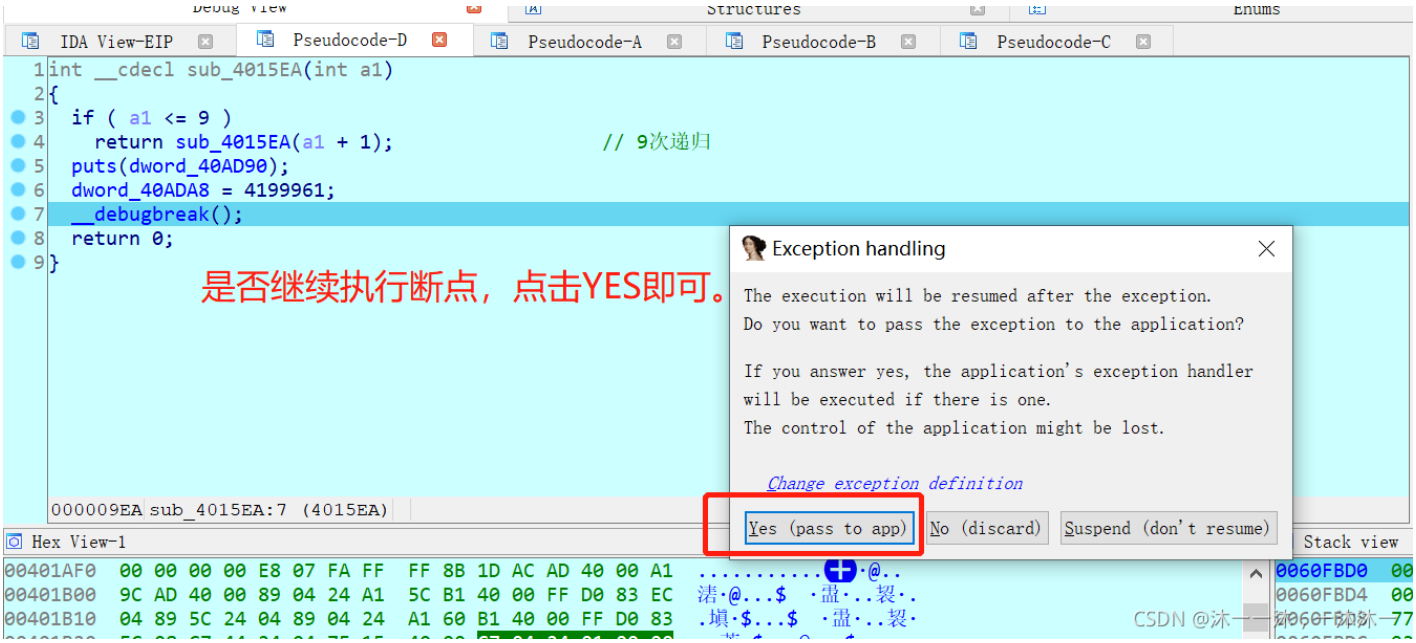
那么整个程序每个环节都对上了，现在跟踪sub_40157F函数，关键就是框中的比较函数sub_401547(input_flag2, off_40AD98)，是一个简单的异或而已，当然off_40AD98也是要动调才能看的，是u1nnf2lg：





(这里积累第五个经验)

补充一下IDA停在__debugbreak函数中时继续单步调试要点击YES才可以，弹框内容大概是是否继续执行断点。



直接附上解题脚本：

key1="u1nnf2lg"

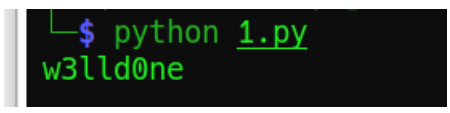
flag2=""

for i in key1:

flag2+=chr(ord(i)^2)

print(flag2)

结果：

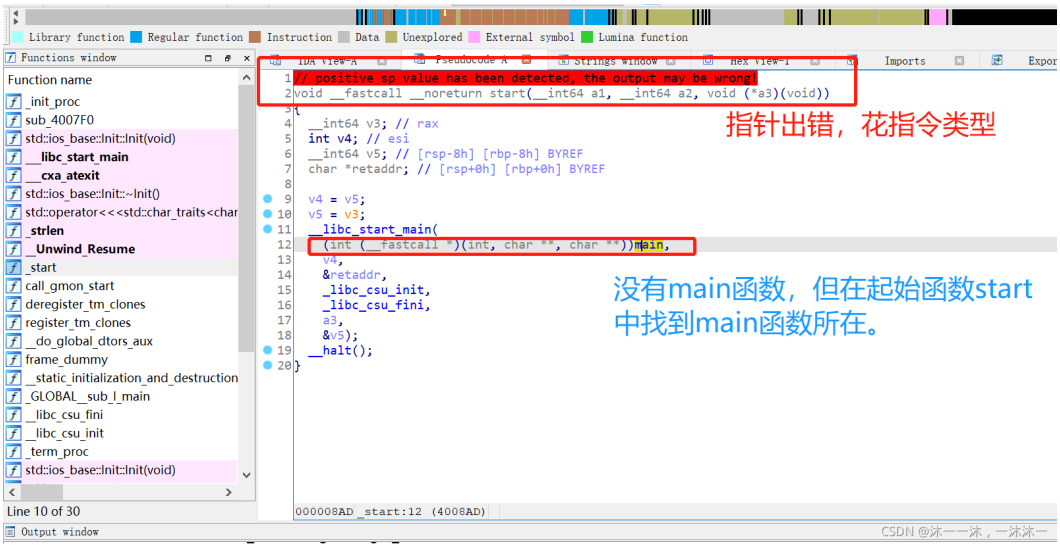


总flag就是下划线连起来flag{r0b0RUlez!_w3lld0ne}

地址运算动态跳转：

攻防世界serial-150：（IDA热键重新反汇编、nop修补垃圾代码、动态地址运算处理、F7和F8交叉使用）

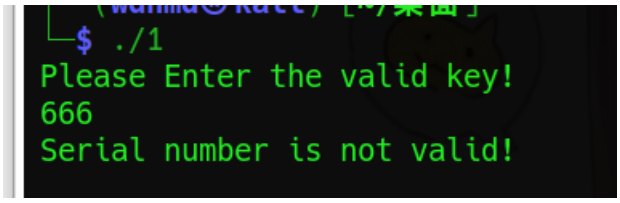
64位ELF文件，无壳，照例扔入IDA64中查看伪代码信息，function窗口中没有main函数，看一下启动的start函数后发现main函数直接放在里面了：



(这里积累第一个经验)

但是双击跟踪又发现无法转成伪代码，后来查了资料发现了，这种代码和数据混合的就是花指令，在《IDA权威指南》中叫模糊指令。书上解释说IDA是静态反汇编，对于第一个红框中jz short near ptr loc_4009F3+2无法执行加法指令，所以直接从loc_4009F3处反汇编，一个指令错了，后面空间全部错误。

想起来还没有运行程序，现在运行一下程序，看一下主要回显信息：



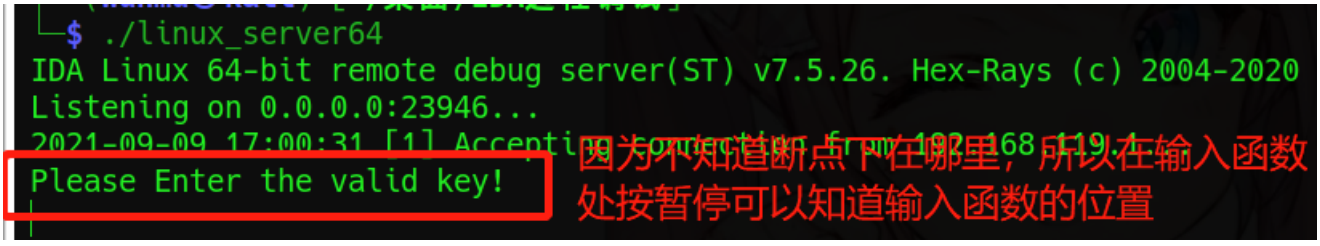
在strings窗口双击跟踪，也只是回显到rdata段中，并没有什么太大的作用：

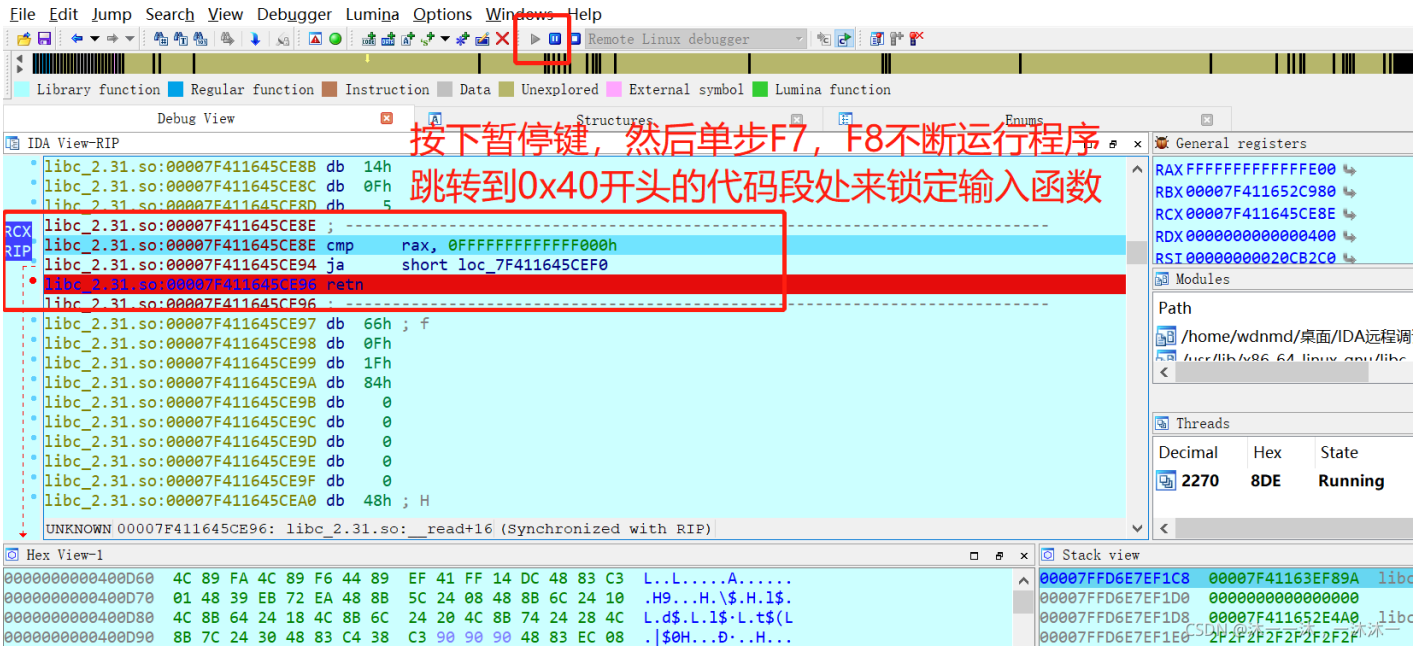
```
.rodata:0000000000400DA8 ; Segment type: Pure data
.rodata:0000000000400DA8 ; Segment permissions: Read
.rodata:0000000000400DA8 _rodata         segment dword public 'CONST' use64
.rodata:0000000000400DA8                 assume cs:_rodata
.rodata:0000000000400DA8                 ;org 400DA8h
.rodata:0000000000400DA8                 public _IO_stdin_used
.rodata:0000000000400DA8 _IO_stdin_used  db    1
.rodata:0000000000400DA9                 db    0
.rodata:0000000000400DAA                 db    2
.rodata:0000000000400DAB                 db    0
.rodata:0000000000400DAC aPleaseEnterThe db 'Please Enter the valid key!',0Ah,0
.rodata:0000000000400DC9 aSerialNumberIs db 'Serial number is valid :)',0Ah,0
.rodata:0000000000400DE4 aSerialNumberIs_0 db 'Serial number is not valid!',0Ah,0
.rodata:0000000000400DE4 _rodata         ends
.rodata:0000000000400DE4
LOAD:0000000000400E01 ; ========================================================================
```

没跟踪到代码段，所以没啥作用。

(这里积累第二个经验)

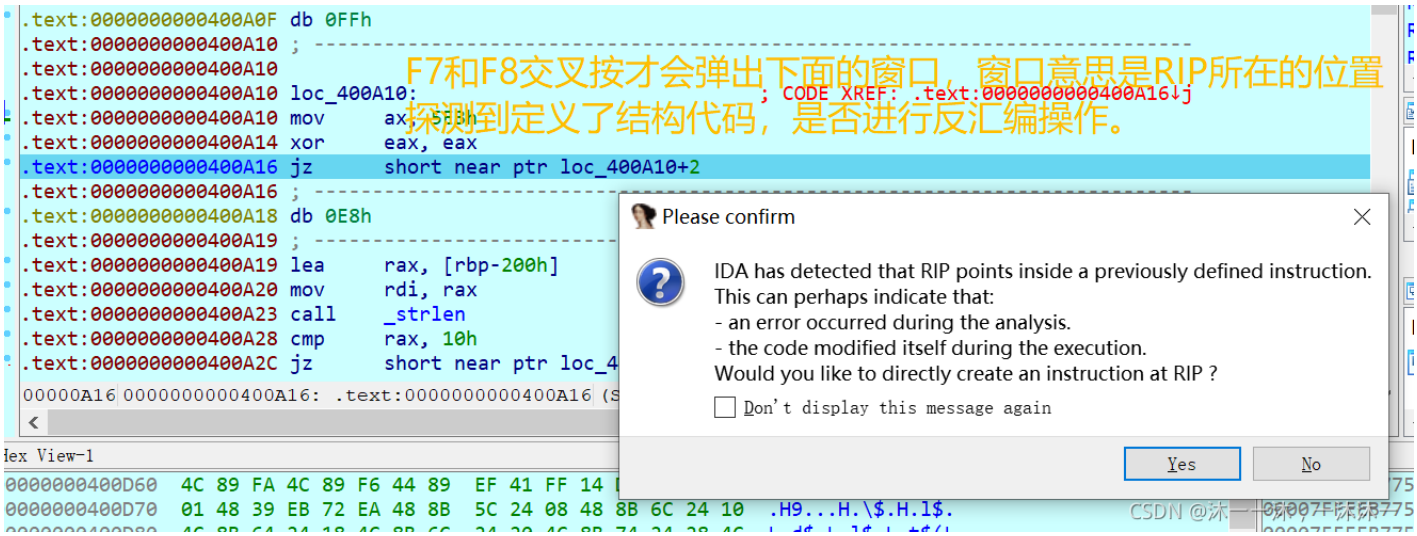
先跟着资料走一遍，动态调试，看程序跑起来之后，代码段里的数据会变成什么代码。因为没有代码参照，所以不知道断点下在那里，只能让它自己在输入暂停时下断点一步步调试(先按开始再按)：



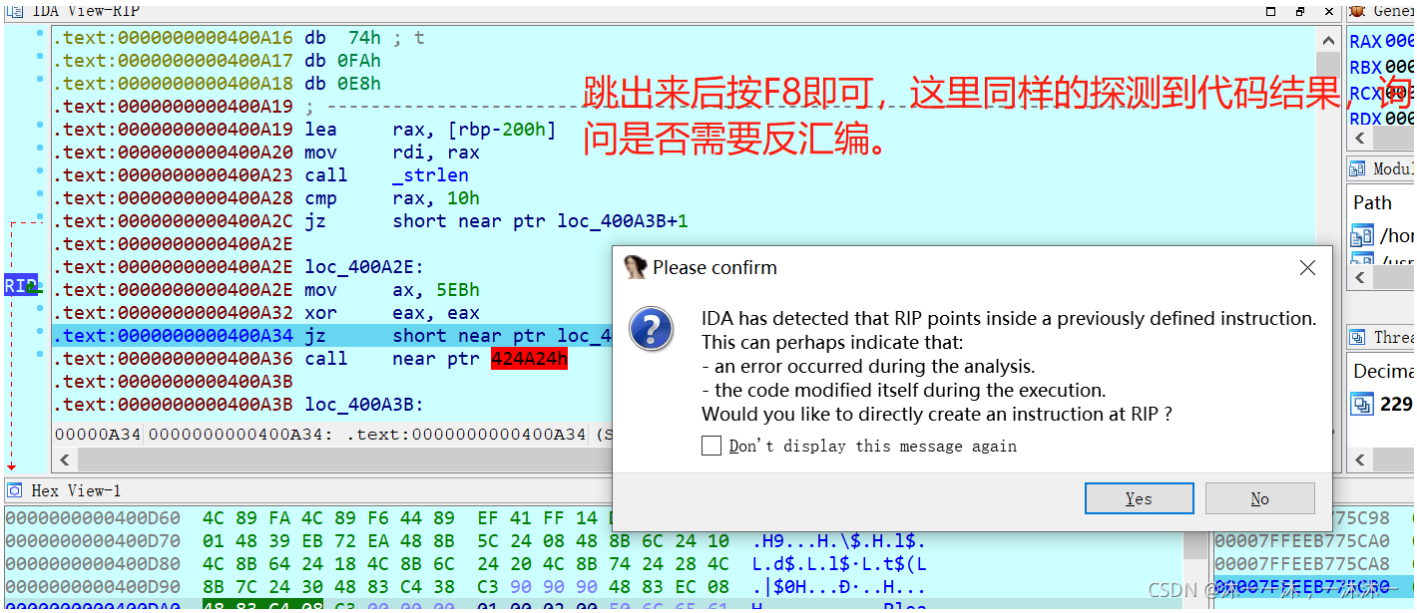


(这里积累第三个经验)

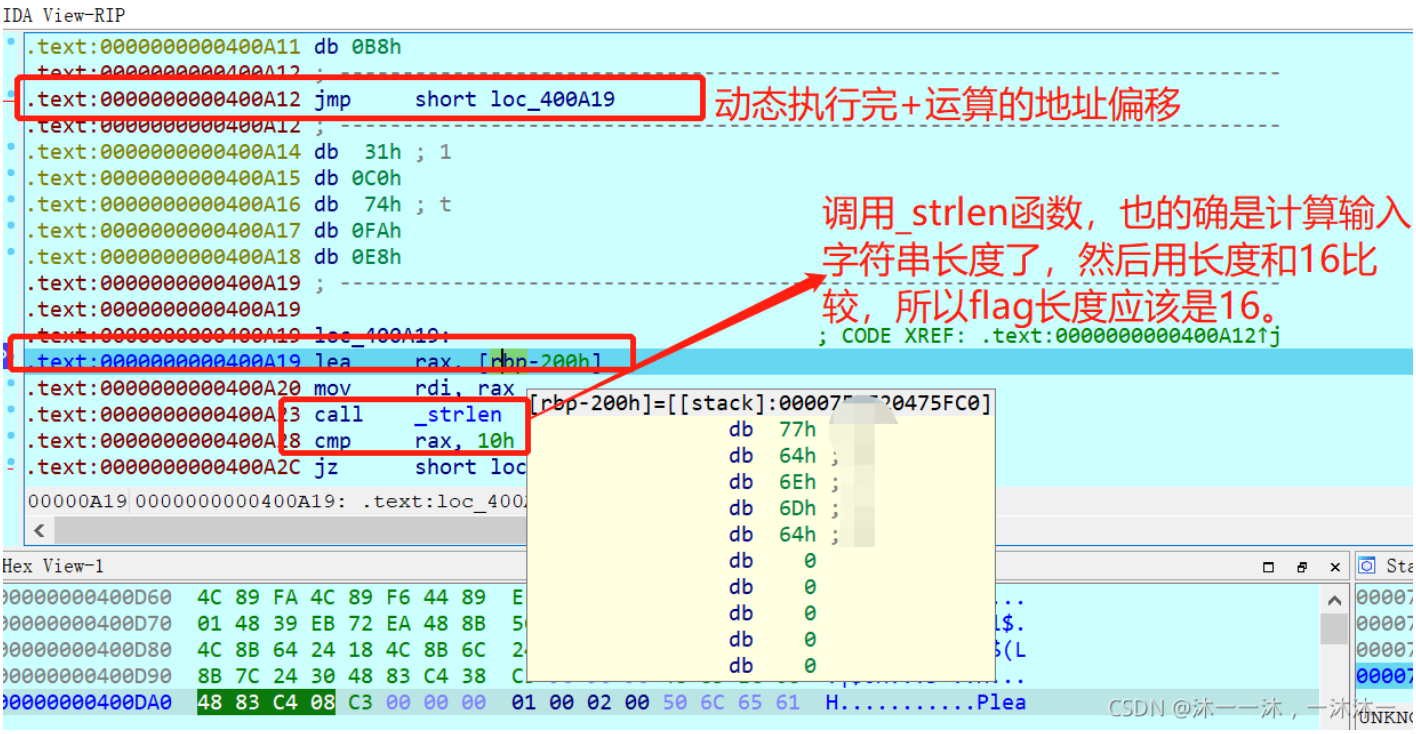
然后F7和F8动态调试，7开头的代码好像是外部链接库的，4开头的代码才是当前程序真正地址。然后就是这里有个小插曲就是一直按F8或者一直按F7都会陷入7开头的地址的不断循环中，只有按一会F8再按一会F7后才能弹出下图的框中并跳出到4开头的真实地址中。(这里应该涉及反调试技术)



跳出来之后一直按F8即可，虽然有时候会跳回7开头的外部地址中，但是很快就会跳出来了。



注意这里F8单步执行中就算跳出到4开头的真实地址中也要自己按C键来转成代码，不然自己是不知道这里是关键代码的，图中这些代码就是按C键按出来的。然后这里鼠标悬停在第二个红框处说明我们输入的字符在这里开始操作，第三个红框说明我们输入的字符长度得是10h，也是是16位。



(这里积累第四个经验)

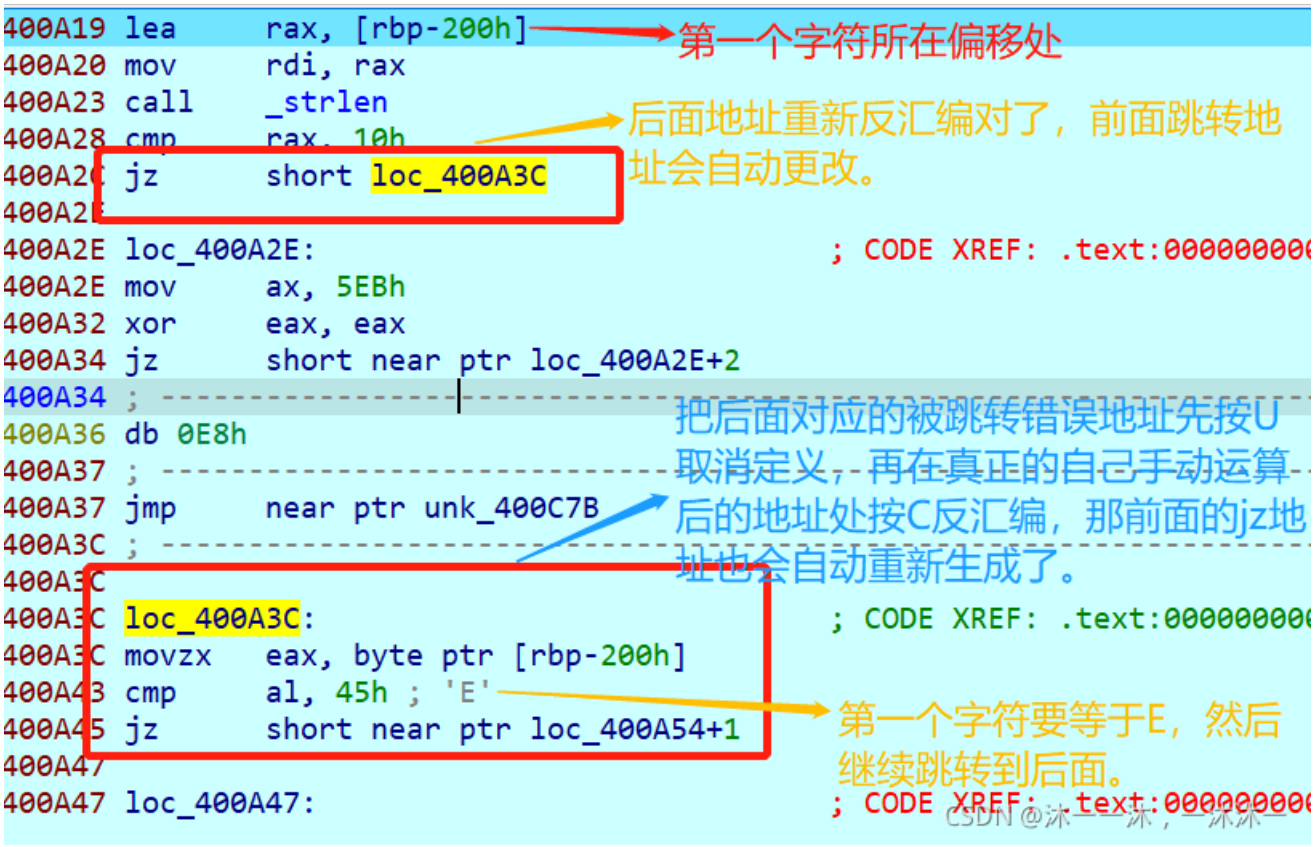
然后，然后就有意思了，很多博客说重新输入16个字符然后重新调试，这太费时间了，代码都在这里了，直接自己协助IDA反汇编代码不就行了？前面满足16字符长后就跳转到后面一段代码中，后面取第一个字符与E比较，为什么是第一个呢，因为直接双击[rbp-200h]即可看到栈的内容了。

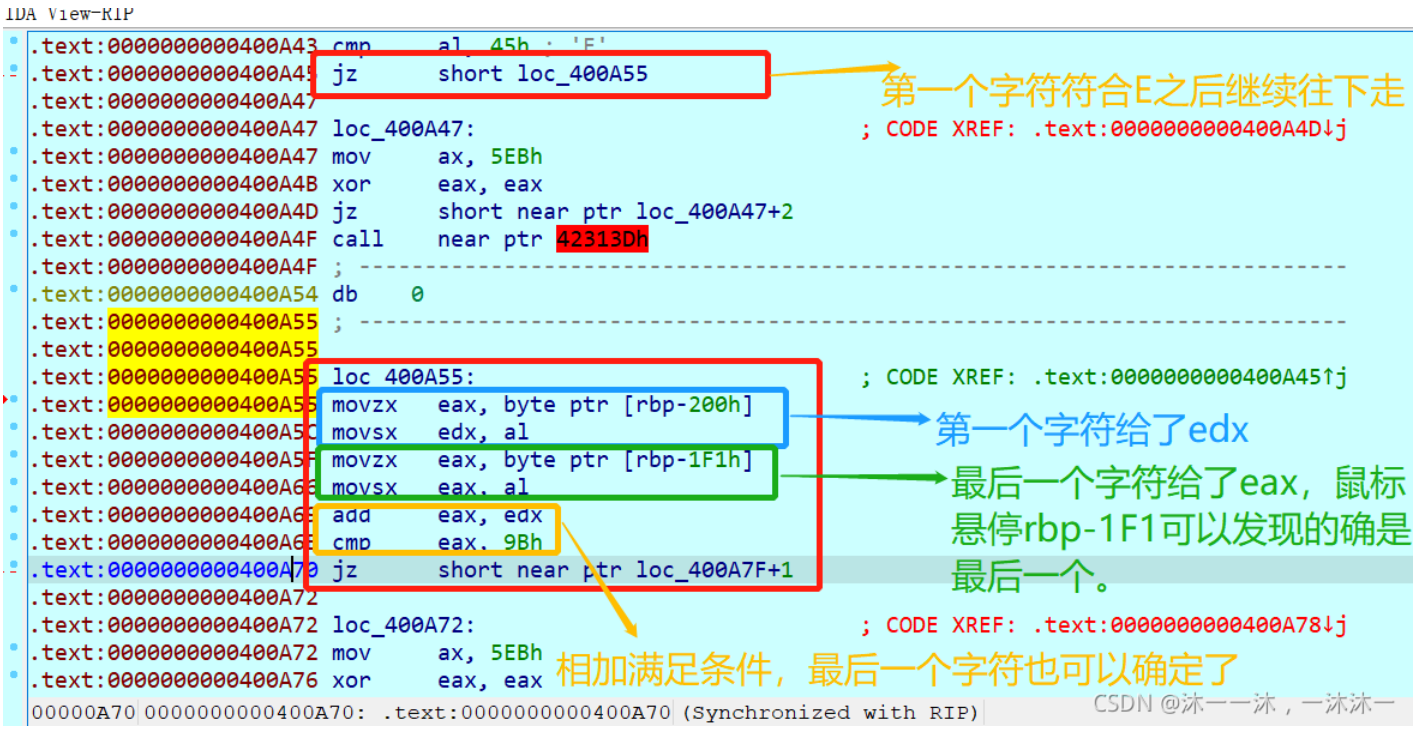
.

然后jz short near ptr loc_400A54+1IDA是识别不了的，我们用U取消对应代码段的定义，然后在400A55出用热键C重新定义即可，重定义后前面jz short near ptr loc_400A54+1会直接变成jz short near ptr loc_400A55。

.

后面的代码基本都是确定前第n个字符，后面把该字符给到edx。然后取对应的倒数第n个字符，并把该字符给到eax，最后add eax, edx 和cmp eax, 9Bh满足相加和即可。





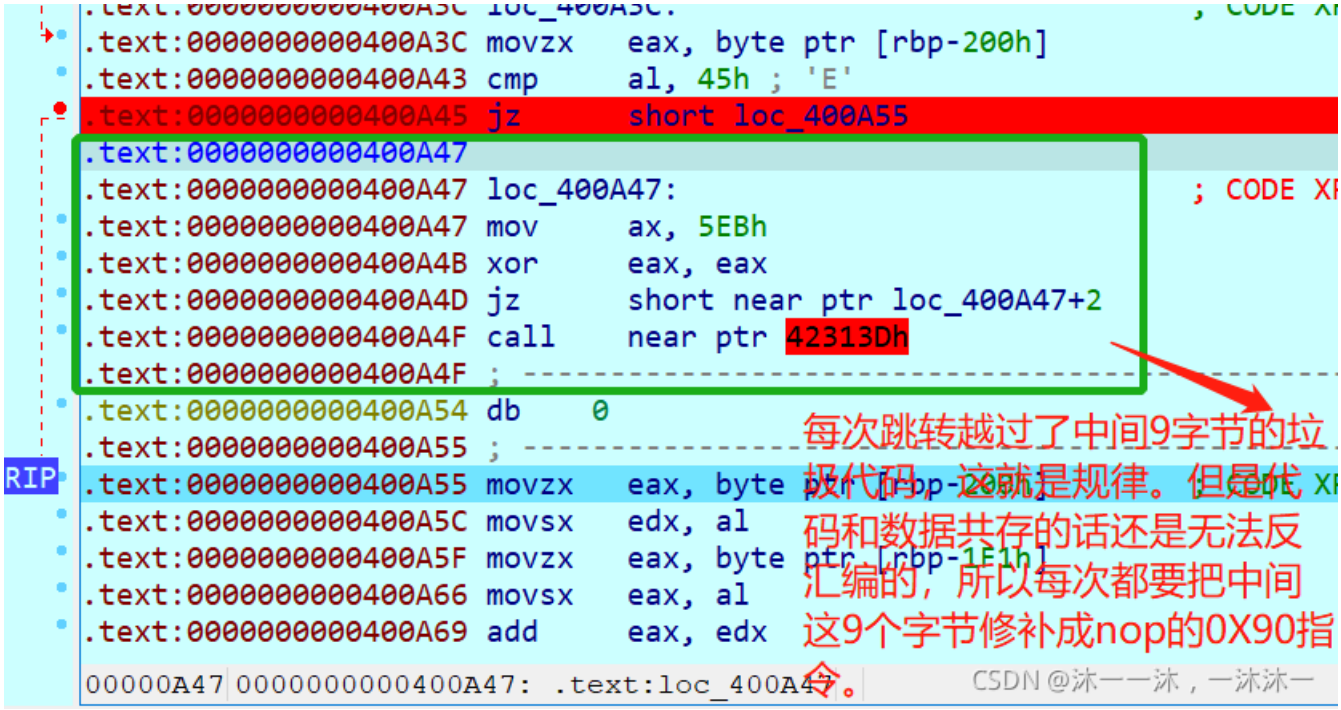
所以最后不断的热键U和热键C不断修改jz short near ptr loc_400A7F+1这类IDA识别不了的代码，即可得出flag：

EZ9dmq4c8g9G7bAV

(这里积累第五个经验)

后来查了很多资料，找到几个直接修改源汇编代码然后F5转伪代码的高手，其实他们也是在前面动态调试中发现逻辑代码之间的规律才能修补反汇编代码的。

下图中绿色框中jz直接就跳过了，所以这部分代码是无用的，后面对输入flag前第n个字符和后第n个字符的操作也是和绿框中一样隔了9个字节的垃圾代码，0x400A54处的db 0其实也是一条代码来的，只是被绿框中垃圾代码挤占了空间反汇编出错而已。



然而就算我们知道了真实代码和垃圾代码的间距，也在主函数中帮IDA把jz short near ptr loc_400A7F+1这类指令一个个修改回来，我们还是不能F5反汇编伪代码，因为代码中不能混有数据，所以我们必须把下图第二个第三个红框中的数据改成nop空指令才行，这样一来工作量就大了。

所以我修改了别人一个IDC的脚本为IDApython嵌入脚本来用，脚本中0x400cac-0x40099c是main函数的代码范围。0x05ebb866是垃圾代码中mov ax, 5EBh的机器指令，紧跟它后面的xor eax, eax机器指令是0xfa74c031，这两个都满足的就是垃圾代码量。然后用for a in range(9):填充间隔的9个Nop

addr=0x40099c

for i in range(0x400cac-0x40099c):

if Dword(addr+i)==0x05ebb866:

if Dword(addr+i+4)==0xfa74c031:

for a in range(9):

PatchByte(addr+i+a,0x90)

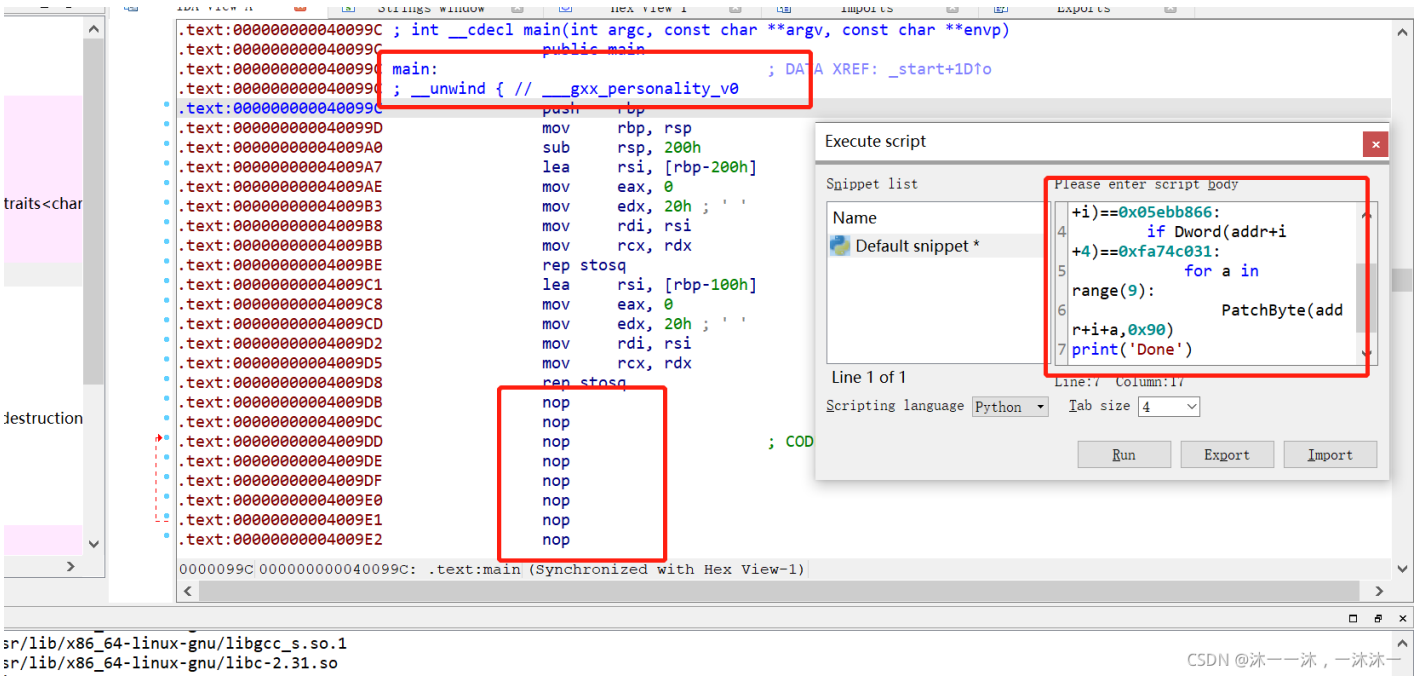
print('Done')

补充，用来脚本中用来判断界限的机器指令怎么看，直接热键U即可看到，从后往前，小端顺序：

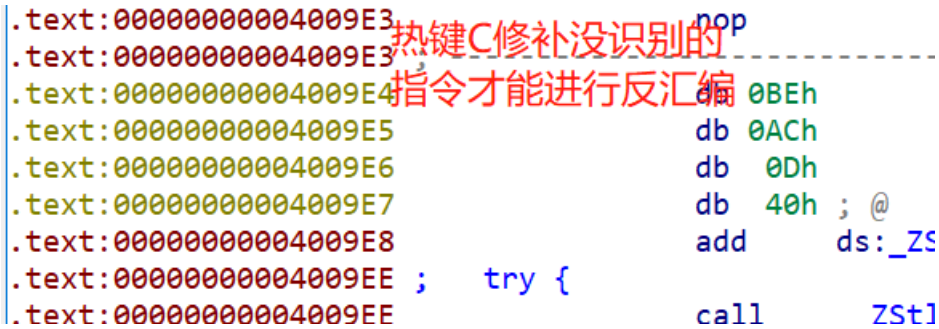
```
0000000000400A36 db 0E8h
0000000000400A37 ; ---------------------------------------------------------------
0000000000400A37 jmp     near ptr unk_400C7B
0000000000400A3C ; ---------------------------------------------------------------
0000000000400A3C
0000000000400A3C loc_400A3C:                             ; CODE XREF: .text:000
0000000000400A3C movzx   eax, byte ptr [rbp-200h]
0000000000400A43 cmp     al, 45h ; 'E'
0000000000400A45 jz      short near ptr unk_400A55
0000000000400A45 ; ---------------------------------------------------------------
0000000000400A47 db  66h ; f
0000000000400A48 db 0B8h
0000000000400A49 db 0EBh
0000000000400A4A db    5
0000000000400A4B db  31h ; 1
0000000000400A4C db 0C0h
0000000000400A4D db  74h ; t
0000000000400A4E db 0FAh
0000000000400A4F db 0E8h
0000000000400A50 db 0E9h
0000000000400A51 db  26h ; &
0000000000400A52 db    2
0000000000400A53 db    0
0000000000400A4A: .text:0000000000400A4A (Synchronized with RIP)
```

取消定义后即可看到对应机器指令，内存中是小端顺序，抽出来时要恢复大端顺序才行。

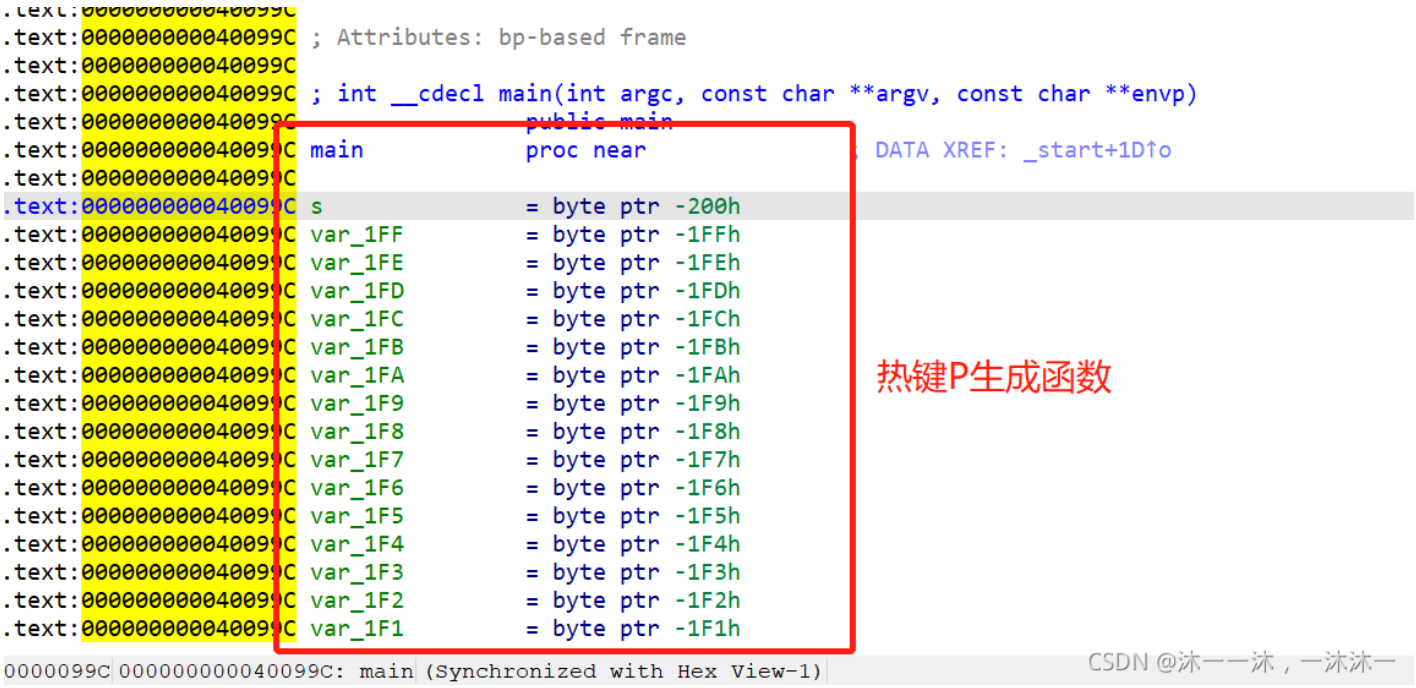
结果：

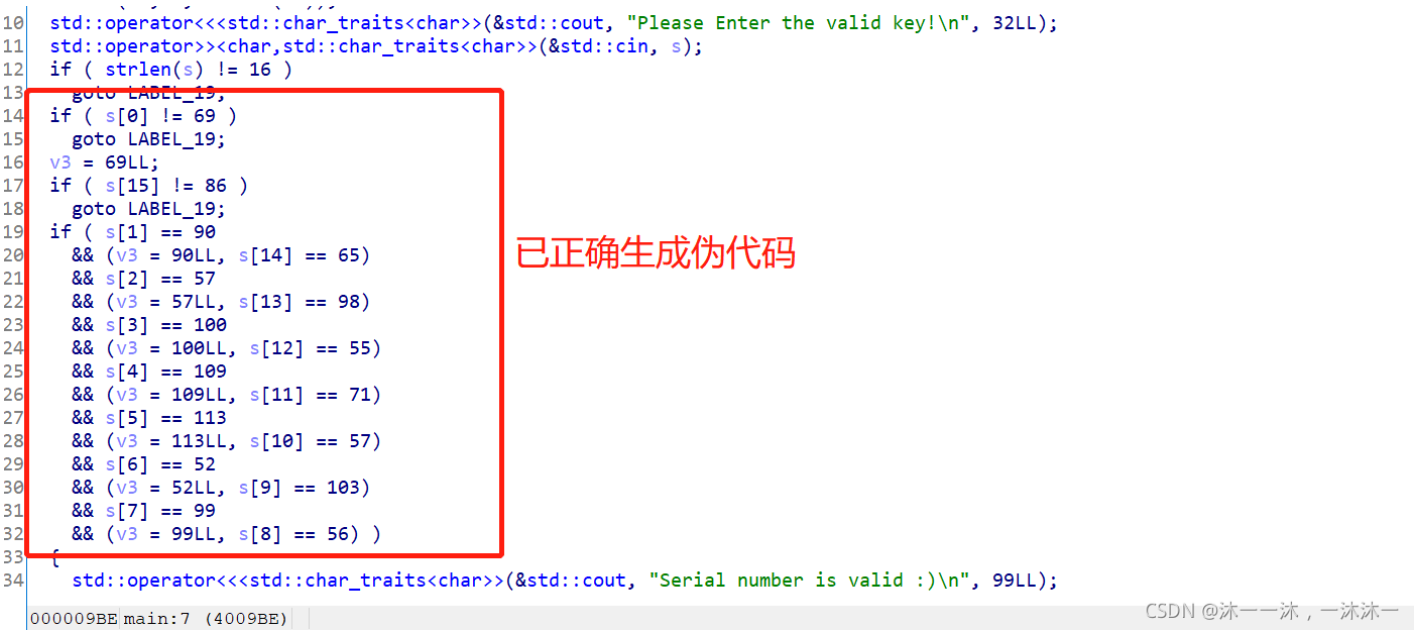


再用热键C修改一些没识别数据指令：（不然会报错Your request has been put in the autoanalysis queue.）



在main函数开头按创造函数的热键P即可创建函数，再按F5即可创建伪代码：





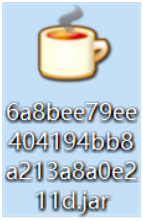
最后虽然很直观也很简单，但是这的确是要在前面动态调试的基础上才能写出脚本的。

安卓java类逆向分析

java逻辑平铺：

攻防世界Guess-the-Number：（代码截断重写）

下载了一个jar文件，根据题目描述猜个数字然后找到flag.，估计题目类型是flag存储型，满足条件就有flag：



用我之前做安卓逆向下载的jar.gui打开，查看代码逻辑：

```
import java.math.BigInteger;

public class guess {
  static String XOR(String _str_one, String _str_two) {
    BigInteger i1 = new BigInteger(_str_one, 16);
    BigInteger i2 = new BigInteger(_str_two, 16);
    BigInteger res = i1.xor(i2);
    String result = res.toString(16);
    return result;
  }

  public static void main(String[] args) {
    int guess_number = 0;
    int my_num = 349763335;
    int my_number = 1545686892;
    int flag = 345736730;
    if (args.length > 0) {
      try {
        guess_number = Integer.parseInt(args[0]);
        if (my_number / 5 == guess_number) {
          String str_one = "4b64ca12ace755516c178f72d05d7061";
          String str_two = "ecd44646cfe5994ebeb35bf922e25dba";
          my_num += flag;
          String answer = XOR(str_one, str_two);
          System.out.println("your flag is: " + answer);
        } else {
          System.err.println("wrong guess!");
          System.exit(1);
        }
      } catch (NumberFormatException e) {
        System.err.println("please enter an integer \nexample: java -jar guess 12");
        System.exit(1);
      }
    } else {
      System.err.println("wrong guess!");
      int num = 1000000;
      num++;
      System.exit(1);
    }
```

xor与输入无关，且生成flag。

逻辑不难，果然是满足条件就有的存储型flag，这里我直接修改截断代码即可，xor是生成存储型flag的代码，要保留：

```
import java.math.BigInteger;

public class guess {

static String XOR(String _str_one, String _str_two) {

BigInteger i1 = new BigInteger(_str_one, 16);

BigInteger i2 = new BigInteger(_str_two, 16);

BigInteger res = i1.xor(i2);

String result = res.toString(16);

return result;

}

public static void main(String[] args) {

int guess_number = 0;

int my_num = 349763335;
```

```
int my_number = 1545686892;

int flag = 345736730;

String str_one = "4b64ca12ace755516c178f72d05d7061";

String str_two = "ecd44646cfe5994ebeb35bf922e25dba";

my_num += flag;

String answer = XOR(str_one, str_two);

System.out.println("your flag is: " + answer);

}

}
```

这里截断代码的时候遇到第一个错误，我竟然忘记编译命令是什么了，还傻傻的用java -c，真的是马冬梅啊马冬梅，后来查看了百度才想起编译命令是javac，运行即得flag：

```
└─$ javac guess.java
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true

┌──(wdnmd㉿kali)-[~/桌面]
└─$ java guess
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
your flag is: a7b08c546302cc1fd2a4d48bf2bf2ddb
```

当然还有第二种方法，看输入后处理的判断逻辑，输入正确的数即可：

```
public static void main(String[] args) {
  int guess_number = 0;
  int my_num = 349763335;
  int my_number = 1545686892;
  int flag = 345736730;
  if (args.length > 0) {
    try {
      guess_number = Integer.parseInt(args[0]);
      if (my_number / 5 == guess_number) {
```

这里判断逻辑是my_number / 5 == guess_number，一开始不记得前面的guess_number = Integer.parseInt(args[0])是什么意思，所以就没往这里想，后来才发现是简单的获取整数参数而已，所以计算机运算1545686892 / 5再取整数就是我们要输入的数：(PS:参数写在右边)

```
C:\Users\Lenovo>java -jar D:\桌面\6a8bee79ee404194bb8a213a8a0e211d.jar 309137378
your flag is: a7b08c546302cc1fd2a4d48bf2bf2ddb

C:\Users\Lenovo>
```

JS类逆向分析

攻防世界secret-string-400：（函数名称暗示、JS控制台操作、涉及虚拟机、字节码相关操作）

下载附件，照例扔入exeinfope中查看信息：

一开始愣了一下，后来发现是压缩包啊，然后继续进一步解压：

一个html和配套js，看起来想我前面做的密码学，毕竟密码学和逆向是相通的嘛：

```
<html>
	<head>
		<title>JSMachine</title>
		<meta charset="UTF-8">
		<script type='text/javascript' src='Machine.js'></script>
	</head>
	<body>
		<h1>Secret key</h1><br/>
		<h2>Test your luck! Enter valid string and you will know flag!</h2><br/>
		<input type='text'></input><br/>
		<br/>
		<input type='button' onclick="check()" value='Проверить'></button>
	</body>
</html>
```

点击按钮后执行js中的check函数


```
function check(){
machine = new Machine;
machine.loadcode([11, 1, 79, 98, 106, 101, 99, 116, 0, 12, 1, 120, 0, 114, 101, 116, 117, 114, 110, 32, 100, 111, 99, 117, 109,
105, 115, 116, 101, 114, 115, 91, 49, 93, 46, 117, 115, 101, 114, 105, 110, 112, 117, 116, 47, 47, 10, 118, 97, 114, 32, 105, 32
32, 53, 48, 44, 32, 52, 48, 44, 32, 51, 55, 44, 32, 52, 56, 44, 32, 50, 52, 44, 32, 49, 48, 44, 32, 53, 54, 44, 32, 53, 53, 44, 32, 52
9, 33, 39, 41, 59, 125, 47, 47, 255, 9, 255, 255, 255, 12, 10, 97, 108, 101, 114, 116, 40, 51, 41, 59, 47, 47, 15, 1, 234, 120, 255
0, 1, 103, 0, 15, 0, 1, 104, 0])
machine.run();
}
```

check函数中有一个machine类和一堆预定义参数，然后就执行run了

继续跟踪，如第一个红框所示，machine是一个函数，然后第二个红框中的run也是一个函数，第三个红框中command调用了调用了parsecommand函数，parsecommand中文是解析命令，这里是一个中文类暗示，但是我一开始并没有看懂。parsecommand跟踪下去发现是一堆连续的调用，基本把整个js剩下的函数都串起来了，逆向起来相当麻烦。

```
function Machine() {
	createRegisters(this);
	this.code = [0]
	this.PC = 0;
	this.callstack = [];
	this.pow = Math.pow(2,32)
};

Machine.prototype = {
	opcodesCount: 16,
	run: run,
	loadcode: function(code){this.code = code},
	end: function(){this.code=[]}
};

function run(){
	while(this.PC < this.code.length){
		var command = parseCommand.call(this)
		command.execute(this);
	}
	//this.end()
}

function getOpcodeObject(){
	var opNum = (this.code[this.PC] % this.opcodesCount);
```

调用函数machine

调用函数run

调用函数parsecommand

下图中Loadcode函数对应一开始的check函数之一，js中只有这一个loadcode函数，就一个预先赋值和，但是这里code中文是代码的意思，一开始我还是没看出来。

```
function Machine() {
        createRegisters(this);
        this.code = [0]
        this.PC = 0;
        this.callstack = [];
        this.pow = Math.pow(2,32)
};

Machine.prototype = {
        opcodesCount: 16,
        run: run,
        loadcode: function(code){this.code = code},
        end: function(){this.code=[]}
};


function run(){
        while(this.PC < this.code.length){
                var command = parseCommand.call(this)
                command.execute(this);
        }
        //this.end()
```

预先赋值代码，跟最开始的一堆数字的虚拟机预定义字节码联系起来，通过预定义字节码和虚拟机来准备起始环境，这部分可以归为系统准备函数。

由于我不太懂js，所以这些线索我都没法串起来，我甚至不知道输入关键字Input在哪里。

（这里积累第一个经验）

没办法了，查了资料，才发现这machine是虚拟机，一开始的loadcode这预加载列表的确有意义，但不是ascii码，所以看不懂，这些是字节码，用于给后面opcodes转化的，检查输入的关键字符Input在转换字节码后的代码里，不在这里。

查看到文件时一个简单的虚拟机，需要了解到所有的opcodes的意义，并且将字节码转化为可读的代码，或者替换run函数来跟踪进我们的程序：

可以看出是个设计了个虚拟机，虚拟机的指令和数据来自 machine.loadcode() 。调试可以发现，其实现原理是根据不同的Opcodes，进行特定的操作，而这些操作是通过js下的 eval() 来执行。loadcode() 里面有不少明文

说到这里基本就可以理清了，opcodes函数把Loadcode中的预加载字符字节码转化为代码，然后才执行run，关键代码就在经过opcodes转化后的字节码中。

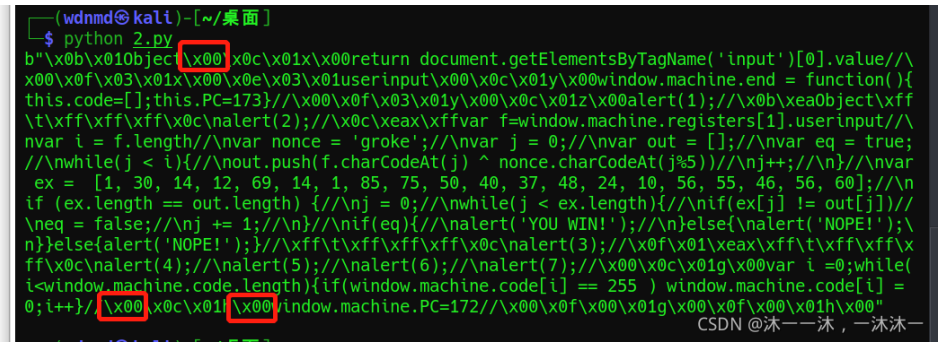
有两种方法获取字节码转换后的代码。

第一种方法，直接转换字节码：

我也是第一次知道可以直接用bytes函数把字节码直接转换为字节串的，也是开了眼界。bytes函数是以字节序列二进制的形式表示字节串，以字节为单位。可能对得上字节码吧。

print(bytes([11, 1, 79, 98, 106, 101, 99, 116, 0, 12, 1, 120, 0, 114, 101, 116, 117, 114, 110, 32, 100, 111, 99, 117, 109, 101, 110, 116, 46, 103, 101, 116, 69, 108, 101, 109, 101, 110, 116, 115, 66, 121, 84, 97, 103, 78, 97, 109, 101, 40, 39, 105, 110, 112, 117, 116, 39, 41, 91, 48, 93, 46, 118, 97, 108, 117, 101, 47, 47, 0, 15, 3, 1, 120, 0, 14, 3, 1, 117, 115, 101, 114, 105, 110, 112, 117, 116, 0, 12, 1, 121, 0, 119, 105, 110, 100, 111, 119, 46, 109, 97, 99, 104, 105, 110, 101, 46, 101, 110, 100, 32, 61, 32, 102, 117, 110, 99, 116, 105, 111, 110, 40, 41, 123, 116, 104, 105, 115, 46, 99, 111, 100, 101, 61, 91, 93, 59, 116, 104, 105, 115, 46, 80, 67, 61, 49, 55, 51, 125, 47, 47, 0, 15, 3, 1, 121, 0, 12, 1, 122, 0, 97, 108, 101, 114, 116, 40, 49, 41, 59, 47, 47, 11, 234, 79, 98, 106, 101, 99, 116, 255, 9, 255, 255, 255, 12, 10, 97, 108, 101, 114, 116, 40, 50, 41, 59, 47, 47, 12, 234, 120, 255, 118, 97, 114, 32, 102, 61, 119, 105, 110, 100, 111, 119, 46, 109, 97, 99, 104, 105, 110, 101, 46, 114, 101, 103, 105, 115, 116, 101, 114, 115, 91, 49, 93, 46, 117, 115, 101, 114, 105, 110, 112, 117, 116, 47, 47, 10, 118, 97, 114, 32, 105, 32, 61, 32, 102, 46, 108, 101, 110, 103, 116, 104, 47, 47, 10, 118, 97, 114, 32, 110, 111, 110, 99, 101, 32, 61, 32, 39, 103, 114, 111, 107, 101, 39, 59, 47, 47, 10, 118, 97, 114, 32, 106, 32, 61, 32, 48, 59, 47, 47, 10, 118, 97, 114, 32, 111, 117, 116, 32, 61, 32, 91, 93, 59, 47, 47, 10, 118, 97, 114, 32, 101, 113, 32, 61, 32, 116, 114, 117, 101, 59, 47, 47, 10, 119, 104, 105, 108, 101, 40, 106, 32, 60, 32, 105, 41, 123, 47, 47, 10, 111, 117, 116, 46, 112, 117, 115, 104, 40, 102, 46, 99, 104, 97, 114, 67, 111, 100, 101, 65, 116, 40, 106, 41, 32, 94, 32, 110, 111, 110, 99, 101, 46, 99, 104, 97, 114, 67, 111, 100, 101, 65, 116, 40, 106, 37, 53, 41, 41, 47, 47, 10, 106, 43, 43, 59, 47, 47, 10, 125, 47, 47, 10, 118, 97, 114, 32, 101, 120, 32, 61, 32, 32, 91, 49, 44, 32, 51, 48, 44, 32, 49, 52, 44, 32, 49, 50, 44, 32, 54, 57, 44, 32, 49, 52, 44, 32, 49, 44, 32, 56, 53, 44, 32, 55, 53, 44, 32, 53, 48, 44, 32, 52, 48, 44, 32, 51, 55, 44, 32, 52, 56, 44, 32, 50, 52, 44, 32, 49, 48, 44, 32, 53, 54, 44, 32, 53, 53, 44, 32, 52, 54, 44, 32, 53, 54, 44, 32, 54, 48, 93, 59, 47, 47, 10, 105, 102, 32, 40, 101, 120, 46, 108, 101, 110, 103, 116, 104, 32, 61, 61, 32, 111, 117, 116, 46, 108, 101, 110, 103, 116, 104, 41, 32, 123, 47, 47, 10, 106, 32, 61, 32, 48, 59, 47, 47, 10, 119, 104, 105, 108, 101, 40, 106, 32, 60, 32, 101, 120, 46, 108, 101, 110, 103, 116, 104, 41, 123, 47, 47, 10, 105, 102, 40, 101, 120, 91, 106, 93, 32, 33, 61, 32, 111, 117, 116, 91, 106, 93, 41, 47, 47, 10, 101, 113, 32, 61, 32, 102, 97, 108, 115, 101, 59, 47, 47, 10, 106, 32, 43, 61, 32, 49, 59, 47, 47, 10, 125, 47, 47, 10, 105, 102, 40, 101, 113, 41, 123, 47, 47, 10, 97, 108, 101, 114, 116, 40, 39, 89, 79, 85, 32, 87, 73, 78, 33, 39, 41, 59, 47, 47, 10, 125, 101, 108, 115, 101, 123, 10, 97, 108, 101, 114, 116, 40, 39, 78, 79, 80, 69, 33, 39, 41, 59, 10, 125, 125, 101, 108, 115, 101, 123, 97, 108, 101, 114, 116, 40, 39, 78, 79, 80, 69, 33, 39, 41, 59, 125, 47, 47, 255, 9, 255, 255, 255, 12, 10, 97, 108, 101, 114, 116, 40, 51, 41, 59, 47, 47, 15, 1, 234, 120, 255, 9, 255, 255, 255, 12, 10, 97, 108, 101, 114, 116, 40, 52, 41, 59, 47, 47, 10, 97, 108, 101, 114, 116, 40, 53, 41, 59, 47, 47, 10, 97, 108, 101, 114, 116, 40, 54, 41, 59, 47, 47, 10, 97, 108, 101, 114, 116, 40, 55, 41, 59, 47, 47, 0, 12, 1, 103, 0, 118, 97, 114, 32, 105, 32, 61, 48, 59, 119, 104, 105, 108, 101, 40, 105, 60, 119, 105, 110, 100, 111, 119, 46, 109, 97, 99, 104, 105, 110, 101, 46, 99, 111, 100, 101, 46, 108, 101, 110, 103, 116, 104, 41, 123, 105, 102, 40, 119, 105, 110, 100, 111, 119, 46, 109, 97, 99, 104, 105, 110, 101, 46, 99, 111, 100, 101, 91, 105, 93, 32, 61, 61, 32, 50, 53, 53, 32, 41, 32, 119, 105, 110, 100, 111, 119, 46, 109, 97, 99, 104, 105, 110, 101, 46, 99, 111, 100, 101, 91, 105, 93, 32, 61, 32, 48, 59, 105, 43, 43, 125, 47, 47, 0, 12, 1, 104, 0, 119, 105, 110, 100, 111, 119, 46, 109, 97, 99, 104, 105, 110, 101, 46, 80, 67, 61, 49, 55, 50, 47, 47, 0, 15, 0, 1, 103, 0, 15, 0, 1, 104, 0]).split(b'\x00')) #除去bytes转换后的\x00残余

这里.split(b'\x00')是为了分隔字符，如果没有它就会这样：



用.split(b'\x00')分隔后就没有这些混淆符了：

依稀可以看出代码就在上面，整理后就是：(input关键字也在这里面)

[b'\x0b\x01Object',

b'\x0c\x01x',

b"return document.getElementsByTagName('input')[0].value//",

b'\x0f\x03\x01x',

b'\x0e\x03\x01userinput',

b'\x0c\x01y',

b'window.machine.end = function(){this.code=[];this.PC=173}//',

b'\x0f\x03\x01y',

b'\x0c\x01z',

b"alert(1);//\x0b\xeaObject\xff\t\xff\xff\xff\x0c\nalert(2);//\x0c\xeax\xffvar

f=window.machine.registers[1].userinput//\nvar i = f.length//\nvar nonce =

'groke';//\nvar j = 0;//\nvar out = [];//\nvar eq = true;//\nwhile(j < i)

{//\nout.push(f.charCodeAt(j) ^ nonce.charCodeAt(j%5))//\nj++;//\n}//\nvar ex =

[1, 30, 14, 12, 69, 14, 1, 85, 75, 50, 40, 37, 48, 24, 10, 56, 55, 46, 56,

60];//\nif (ex.length == out.length) {//\nj = 0;//\nwhile(j < ex.length)

{//\nif(ex[j] != out[j])//\neq = false;//\nj += 1;//\n}//\nif(eq){//\nalert('YOU

WIN!');//\n}else{\nalert('NOPE!');\n}}else{alert('NOPE!');}//\xff\t\xff\xff\xff\

x0c\nalert(3);//\x0f\x01\xeax\xff\t\xff\xff\xff\x0c\nalert(4);//\nalert(5);//\na

lert(6);//\nalert(7);//",

b'\x0c\x01g',

b'var i =0;while(i<window.machine.code.length){if(window.machine.code[i] == 255

) window.machine.code[i] = 0;i++}//',

b'\x0c\x01h',

b'window.machine.PC=172//',

```
b'\x0f',
b'\x01g',
b'\x0f',
b'\x01h',
b'']
```

整理后：

```
f = window.machine.registers[1].userinput//
var i = f.length
var nonce = 'groke';
var j = 0;
var out = [];
var eq = true;
while (j < i) {
out.push(f.charCodeAt(j) ^ nonce.charCodeAt(j % 5))
j++;
}
var ex = [1, 30, 14, 12, 69, 14, 1, 85, 75, 50, 40, 37, 48, 24, 10, 56, 55, 46,
56, 60];
if (ex.length == out.length) {
j = 0;
while (j < ex.length) {
if (ex[j] != out[j]) #异或后要与特定数组相等
eq = false;
j += 1;
}
if (eq) {
alert('YOU WIN!');
} else {
alert('NOPE!');
}
```

} else {

alert('NOPE!');

}

逻辑很简单，因为考点在转换，所以直接给逆向脚本：

key1=[1, 30, 14, 12, 69, 14, 1, 85, 75, 50, 40, 37, 48, 24, 10, 56, 55, 46, 56, 60]

key2="groke"

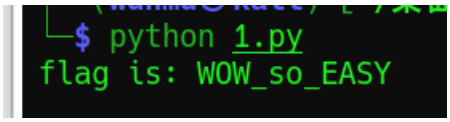
flag=""

for i in range(len(key1)):

flag+=chr(key1[i]^ord(key2[i%5]))

print(flag)

结果：



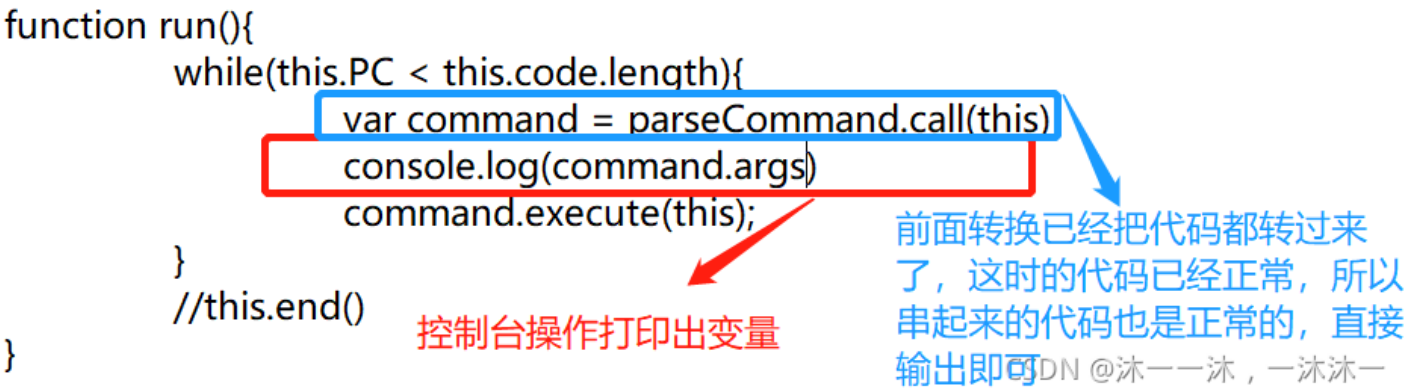
第二种方法，在调用完opcodes后(即字节码转换完代码后)把代码导出来到控制台上：(js日常控制台操作)

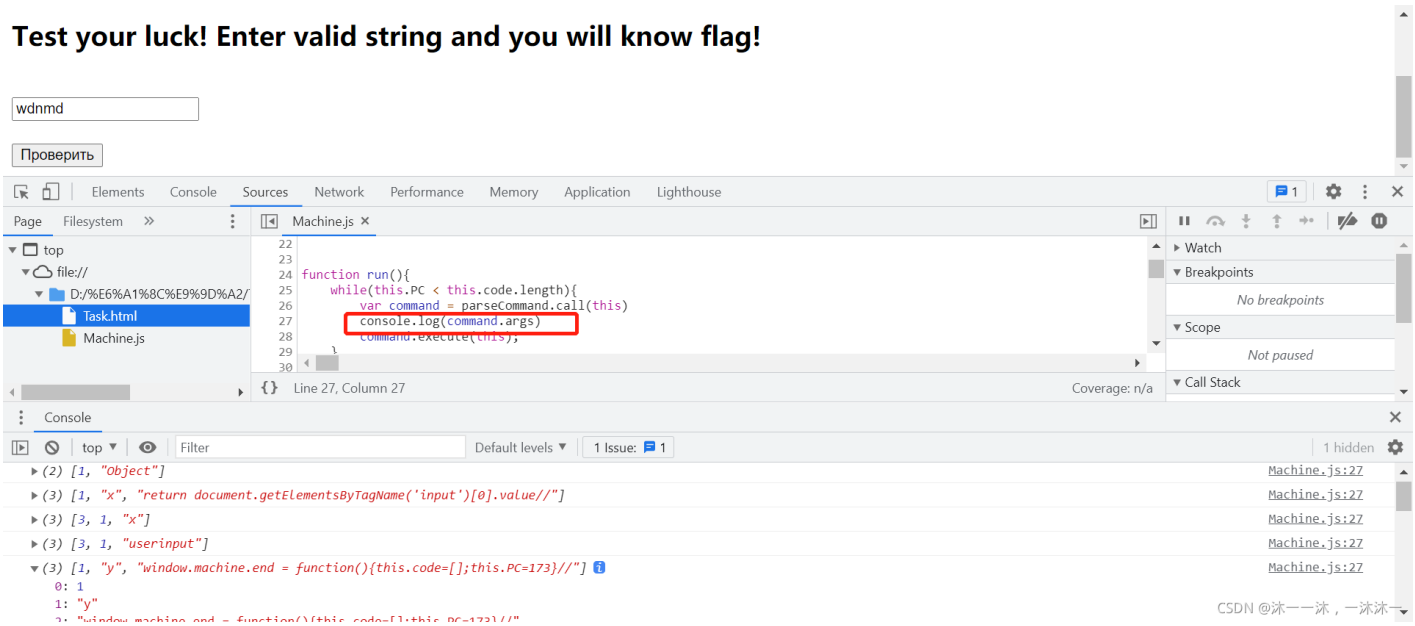
(这里积累第三个经验)

前面我们说check函数中Loadcode导入预加载列表后就执行run函数了，查看run函数中发现parsecommand.call函数是多层调用和嵌套函数，把所有的opcodes函数串起来执行了，也就是这个command变量接受了字节码经opcodes函数

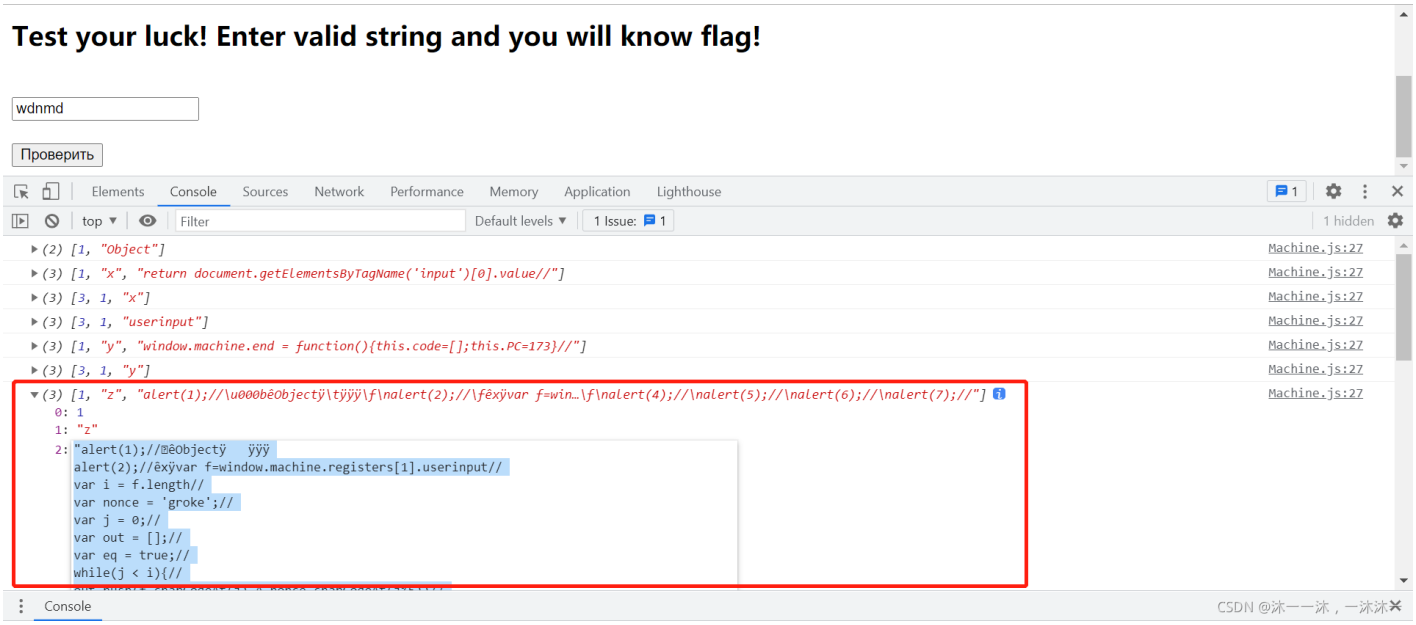
转换后的代码，然后才用下面的command.execute来执行代码。

所以我们用console.log(变量.args)导出变量内容，可以在源码中添加，也可以在浏览器F12的source中添加。(我也不知道为什么console.log(command.args)中要加args这个属性，可能变量.args才能导出变量内容吧。)





在控制台console窗口中一堆折叠数据中逐个展开，双击查看代码即可发现最长的就是字节码转换后的代码。



python逆向分析

pyc文件反编译：

攻防世界re4-unvm-me：（逆向解题模板、md5加密/解密算法、）

下载附件，是.pyc文件，用在线网站反编译：

https://tool.lu/pyc/

#!/usr/bin/env python

visit https://tool.lu/pyc/ for more information

import md5

md5s = [

0x831DAA3C843BA8B087C895F0ED305CE7L,

```
0x6722F7A07246C6AF20662B855846C2C8L,
0x5F04850FEC81A27AB5FC98BEFA4EB40CL,
0xECF8DCAC7503E63A6A3667C5FB94F610L,
0xC0FD15AE2C3931BC1E140523AE934722L,
0x569F606FD6DA5D612F10CFB95C0BDE6DL,
0x68CB5A1CF54C078BF0E7E89584C1A4EL,
0xC11E2CD82D1F9FBD7E4D6EE9581FF3BDL,
0x1DF4C637D625313720F45706A48FF20FL,
0x3122EF3A001AAECDB8DD9D843C029E06L,
0xADB778A0F729293E7E0B19B96A4C5A61L,
0x938C747C6A051B3E163EB802A325148EL,
0x38543C5E820DD9403B57BEFF6020596DL]
print 'Can you turn me back to python ? ...'
flag = raw_input('well as you wish.. what is the flag: ')
if len(flag) > 69:
print 'nice try'
exit()
if len(flag) % 5 != 0:
print 'nice try'
exit()
for i in range(0, len(flag), 5):
s = flag[i:i + 5]
if int('0x' + md5.new(s).hexdigest(), 16) != md5s[i / 5]:
print 'nice try'
exit()
continue
print 'Congratz now you have the flag'
```

突然遇到python逆向有点懵住了，以前都是C语言逆向,一下子竟然忘了流程,现在回顾一下:

2:
这里积累第二个经验：主要回顾一下每一步的思路，给日后自己增添一些解题模板。 .
第一步确定Flag字符数量，第一个红框处得到flag数量是35。
第二步找到已确定的字符串作为基点来反推flag字符，如第二个红框处。
第三步找出逻辑中与flag直接相关的部分，该部分可以正向爆破或者从尾到头的反向逻辑，如第一个红框所示。然后找到与flag没有直接关联的部分，该部分无需逆向逻辑，直接正向流程复现即可，如第二个红框所示。

照着流程我们开始分析，但在分析之前先纠正一个我的误区：

(这里积累第一个经验)

列表是以逗号分隔元素的，md5s[i/5]取的是每一个逗号前的元素，这里一个元素是0x831DAA3C843BA8B087C895F0ED305CE7L 这种这么长的，一开始我直接被带偏了，以为是从这么长的十六进制字符串中抽取，现在明白了，md5s列表中就是以这么长的数为一个基本元素的，i/5就是从这十三个长数中挑一个。

```
import md5
md5s = [
    0x831DAA3C843BA8B087C895F0ED305CE7L,
    0x6722F7A07246C6AF20662B855846C2C8L,
    0x5F04850FEC81A27AB5FC98BEFA4EB40CL,
    0xECF8DCAC7503E63A6A3667C5FB94F610L,
    0xC0FD15AE2C3931BC1E140523AE934722L,
    0x569F606FD6DA5D612F10CFB95C0BDE6DL,
    0x68CB5A1CF54C078BF0E7E89584C1A4EL,
    0xC11E2CD82D1F9FBD7E4D6EE9581FF3BDL,
    0x1DF4C637D625313720F45706A48FF20FL,
    0x3122EF3A001AAECDB8DD9D843C029E06L,
    0xADB778A0F729293E7E0B19B96A4C5A61L,
    0x938C747C6A051B3E163EB802A325148EL,
    0x38543C5E820DD9403B57BEFF6020596DL]
print 'Can you turn me back to python ? ...'
flag = raw_input('well as you wish.. what is the flag: ')
if len(flag) > 69:
    print 'nice try'
    exit()
if len(flag) % 5 != 0:    #5的倍数而且要小于等于65
    print 'nice try'
    exit()
for i in range(0, len(flag), 5):
    s = flag[i:i + 5]         #左闭右开相差5
    if int('0x' + md5.new(s).hexdigest(), 16) != md5s[i / 5]:
        print 'nice try'
```

逗号分隔，一个列表元素就是一个完整的md5值，就是这么长。

取列表中一个完整的md5值来比较。

那么按照流程来第一步确定flag的数量，md5s列表有13组，flag是5的倍数而且小于等于65，md5s[i/5]中i为65的话就刚好遍历md5s数组，所以flag是65个字符。

```
import md5
md5s = [
    0x831DAA3C843BA8B087C895F0ED305CE7L,
    0x6722F7A07246C6AF20662B855846C2C8L,
    0x5F04850FEC81A27AB5FC98BEFA4EB40CL,
    0xECF8DCAC7503E63A6A3667C5FB94F610L,
    0xC0FD15AE2C3931BC1E140523AE934722L,
    0x569F606FD6DA5D612F10CFB95C0BDE6DL,
    0x68CB5A1CF54C078BF0E7E89584C1A4EL,
    0xC11E2CD82D1F9FBD7E4D6EE9581FF3BDL,
    0x1DF4C637D625313720F45706A48FF20FL,
    0x3122EF3A001AAECDB8DD9D843C029E06L,
    0xADB778A0F729293E7E0B19B96A4C5A61L,
    0x938C747C6A051B3E163EB802A325148EL,
    0x38543C5E820DD9403B57BEFF6020596DL]
print 'Can you turn me back to python ? ...'
flag = raw_input('well as you wish.. what is the flag: ')
if len(flag) > 69:
    print 'nice try'
    exit()
if len(flag) % 5 != 0:    #5的倍数而且要小于等于65
    print 'nice try'
    exit()
for i in range(0, len(flag), 5):
    s = flag[i:i + 5]         #左闭右开相差5
    if int('0x' + md5.new(s).hexdigest(), 16) != md5s[i / 5]:
        print 'nice try'
```

flag位数满足条件的有65，65/5=13，刚好遍历上面13个元素的列表，所以flag是65位

第二步，找到已确定的字符串作为基点反推字符：很明显基点是md5s[i/5]。

```
  0xC11E2CD82D1F9FBD7E4D6EE9581FF3BDL,
  0x1DF4C637D625313720F45706A48FF20FL,
  0x3122EF3A001AAECDB8DD9D843C029E06L,
  0xADB778A0F729293E7E0B19B96A4C5A61L,
  0x938C747C6A051B3E163EB802A325148EL,
  0x38543C5E820DD9403B57BEFF6020596DL]
int 'Can you turn me back to python ? ...'
ag = raw_input('well as you wish.. what is the flag: ')
 len(flag) > 69:
  print 'nice try'
  exit()
 len(flag) % 5 != 0:    #5的倍数而且要小于等于65
  print 'nice try'
  exit()
r i in range(0, len(flag), 5):
 s = flag[i:i + 5]          #左闭右开相差5
  if int('0x' + md5.new(s).hexdigest(), 16) != md5s[i / 5]:
      print 'nice try'
```

每次取flag的5个字符来满足一列md5值，所以md5解密即可

第三步，找到逻辑中与flag直接相关的部分，爆破或反向。不直接相关的部分正向复现：

下面是用s顺序取flag的5个字符（flag[i:i+5]中i+5优先级大于:），然后md5加密后与md5s的13组元素顺序比较，那逆向逻辑不就是md5得13组元素每组对应flag的5个字符吗？

所以直接取md5s列表的13组元素md5转明文即可：

https://www.somd5.com/batch.html

(这里积累第二个经验)

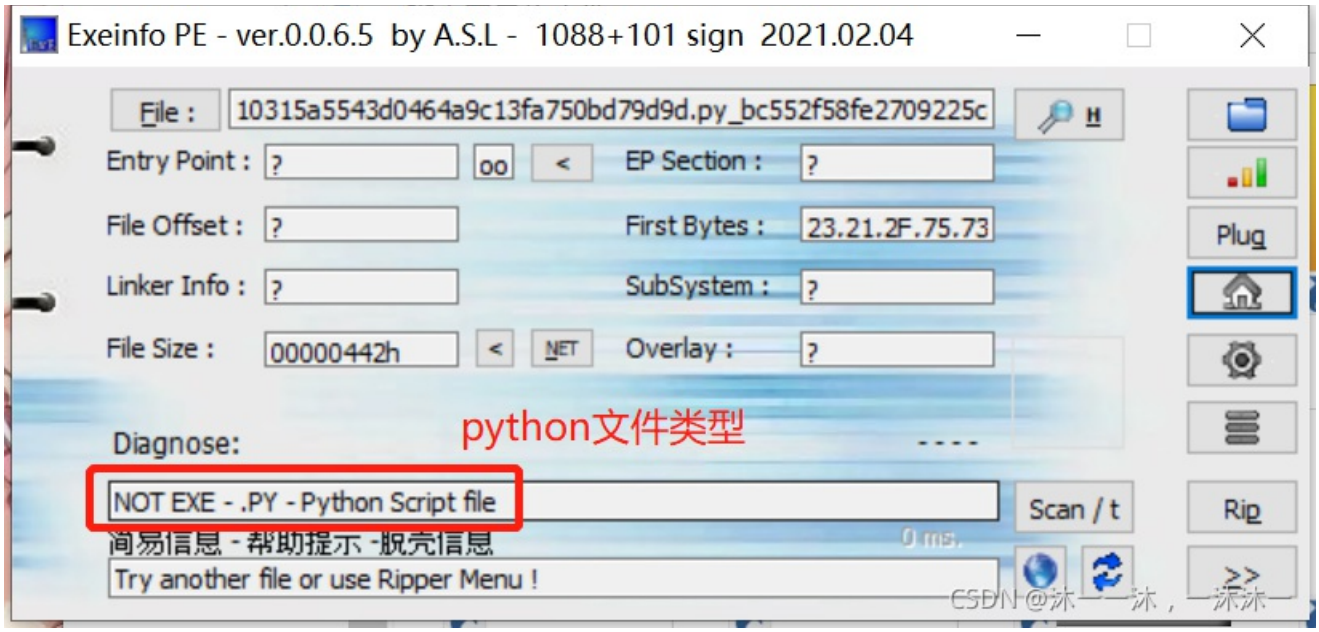
这里md5s列表中的元素大都是32为的刚好是md5的位数，只有一个不是，如下列红框框起来的，这种要在前面高位补0才行。

(这里积累第三个经验)

这里还有个小插曲，一开始我用alexctf来正向尝试爆破md5值，因为比赛通常是以赛名为flag前缀的嘛，结果当然是错的。正确的5位字符多试试说不定可以，但是alexctf是6位字符，md5加密不管多少字符都给你输出16位或32位密文，所以多一位或少一位就完全对不上了。

攻防世界handcrafted-pyc：（函数积累、涉及虚拟机、代码截断重写、字节码相关操作）

下载附件，照例扔入exeinfope中查看信息：



(这里积累第一个经验)

额，这里写着py脚本文件，但是因为我下载的附件没有后缀名，以为是编译好的pyc文件，于是用https://tool.lu/pyc/在线反编译来反编译文件。编译的结果一言难尽，我以为是出题人和 https://tool.lu/pyc/串通好了，结果发现是我错了：

```
#!/usr/bin/env python
# encoding: utf-8
# 如果觉得不错，可以分享给你的朋友 https://tool.lu/pyc/
# 2018-03-05 1.3 支持中文字符串的反编译

x = {   "a":37,"b":42,

"c":927}

y = "hello" "tool.lu"
z = "hello "+"tool.lu"
a = "hello {}".format("tool.lu")
class foo  (       object  ):
   def f     (self    ):
      return          37*-+2
   def g(self, x,y=42):
         return y
def f  (    a ) :
   return          37+-+a[42-x :   y**3]
```

反编译了个啥也不是

正常流程应该是打开文件，发现python代码：

#!/usr/bin/env python

-*- coding: utf-8 -*-

import marshal, zlib, base64

exec(marshal.loads(zlib.decompress(base64.b64decode('eJyNVktv00AQXm/eL0igiaFA01IO4clVCUGFBBJwqR/

代码量不多，开始查函数作用：

base64.b64decode()函数的作用是将Base64加密的字符串解码；

zlib.decompress()函数的作用是将压缩的字符串解压缩；

marshal.loads(bytes)函数的作用是将pyc字节码反序列化为Python模块对象；

exec()执行Python代码。

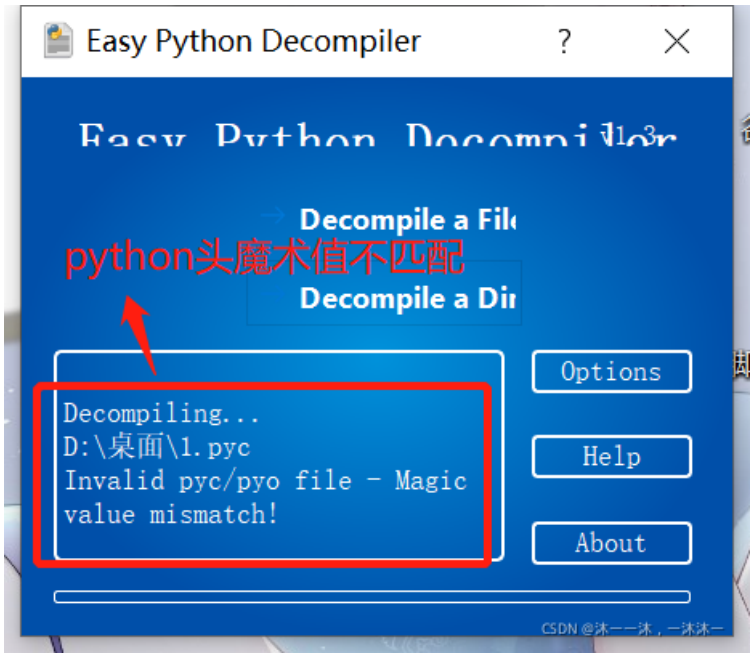
这里就涉及了虚拟机的概念了，回顾以前博客，虚拟机就是对字节码的操作以转换为代码：

所以这里Python反编译的正常做法应该是考虑将字节码用工具反编译成代码，然后逆向算法。

所以我们要在marshal.loads(bytes)之前把字节码打印出来到pyc文件中去：

```
import zlib, base64
f=open('1.pyc','wb') #二进制写入文件字节码（二进制数据）
pyc=zlib.decompress(base64.b64decode('eJyNVktv00AQXm/eL0igiaFA01IO4clVCUGFBBJwqRAckLhElQmtRfF
f.write(pyc)
f.close()
```

生成的1.pyc扔入工具EasyPythonDecompiler中报错，说是魔术值不匹配：



翻一下笔记中的python2的pyc文件头部内容：

pyc文件头部比较简单，在python2中只占用4个字节，包含两个字段magic和mtime。

MAGIC	long	魔数，区分不同版本的 python 字节码
MTIME	long	修改时间

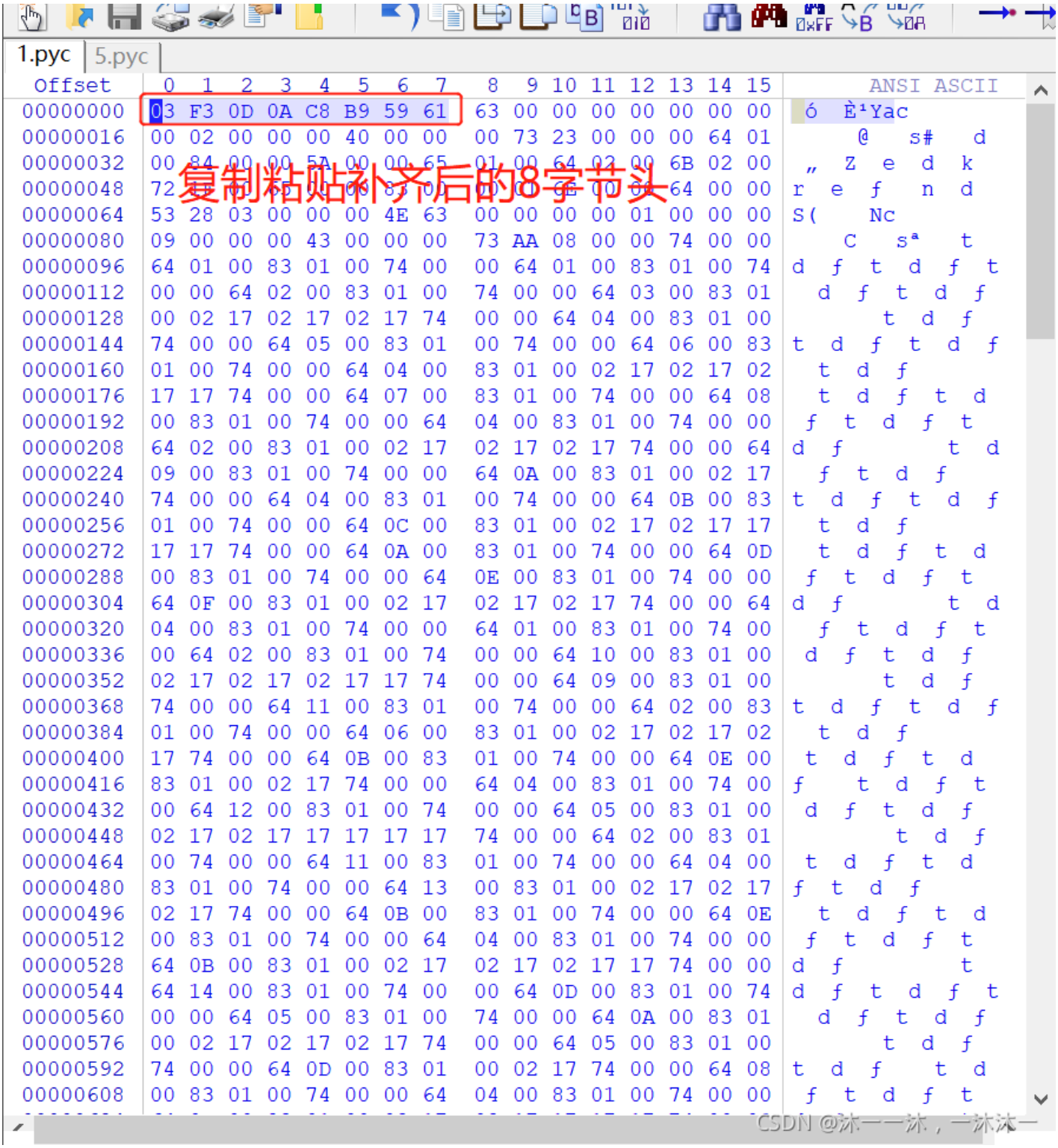
pyc文件头部：
前4个字节：03f3 0d0a，表示python版本
5-8个字节：0e6b 905d，表示pyc文件修改时间

所以找一个正常python2生成的pyc文件复制前8个字节过去即可：

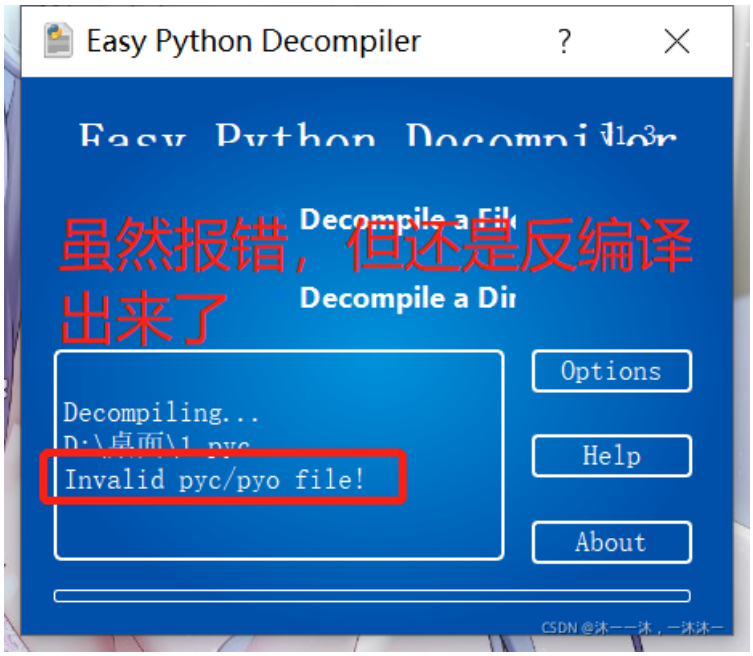
1.pyc 5.pyc

Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	ANSI ASCII
00000000	03	F3	0D	0A	C8	B9	59	61	63	00	00	00	00	00	00	00	ó È¹Yac
00000016	00	01	00	00	00	40	00	00	00	73	09	00	00	00	64	00	@ s d
00000032	00	47	48	64	01	00	53	28	02	00	00	00	69	9A	02	00	GHd S(iš
00000048	00	4E	28	00	00	00	00	28	00	00	00	00	28	00	00	00	N(((
00000064	00	28	00	00	00	00	73	04	00	00	00	35	2E	70	79	74	(s 5.pyt
00000080	08	00	00	00	3C	6D	6F	64	75	6C	65	3E	01	00	00	00	<module>
00000096	74	00	00	00	00												t

正常8字节头



继续扔入工具EasyPythonDecompiler中，虽然报错，但还是反编译出来了：(用uncompyle6反编译结果也是一样的)




```
# Embedded file name: <string>

def main--- This code section failed: ---

0       LOAD_GLOBAL      'chr'
3       LOAD_CONST       108
6       CALL_FUNCTION_1   None
9       LOAD_GLOBAL      'chr'
12      LOAD_CONST       108
15      CALL_FUNCTION_1   None
18      LOAD_GLOBAL      'chr'
21      LOAD_CONST       97
24      CALL_FUNCTION_1   None
27      LOAD_GLOBAL      'chr'
30      LOAD_CONST       67
33      CALL_FUNCTION_1   None
36      ROT_TWO          None
37      BINARY_ADD        None
38      ROT_TWO          None
39      BINARY_ADD        None
40      ROT_TWO          None
41      BINARY_ADD        None
```

反汇编出来的是python汇编，而不是常规的python代码。

（这里积累第四个经验）

Python字节码结构如下：

源码行号 | 跳转注释符 | 指令在函数中的偏移 | 指令符号（助记符） | 指令参数 | 实际参数值

字节码操作的详细信息：

starts_line（源码行号）：以此操作码开头的行（如果有），否则 None

is_jump_target（跳转注释符）：True 如果其他代码跳转到这里，否则 False

Offset（指令在函数中的偏移）：字节码序列中操作的起始索引

opcode：操作的数字代码，对应于下面列出的操作码值和操作码集合中的字节码值。

opname（指令符号（助记符））：人类可读的操作名称

arg（指令参数）：操作的数字参数（如果有），否则 None

argval：解析的 arg 值（如果已知），否则与 arg 相同

Argrepr（实际参数值）：操作参数的人类可读描述

例如：

1 0 LOAD_GLOBAL 0 ‘chr’

该字节码指令在源码中对应1行

此处不是跳转

0该字节指令的字节码偏移

操作指令对应的助记符为LOAD_GLOBAL

指令参数为0

操作参数对应的实际值为’chr’

LOAD_GLOBA：

将全局变量co_names[namei]加载到堆栈上。这里是第0个变量

LOAD_FAST(var_num)：

将对本地co_varnames[var_num]的引用推入堆栈。一般加载局部变量的值，也就是读取值，用于计算或者函数调用传参等。

STORE_FAST(var_num)：

将TOS存储到本地co_varnames[var_num]中。一般用于保存值到局部变量。

LOAD_CONST：

推入一个实整数值到计算栈的顶部。，比如数值、字符串等等，一般用于传给函数的参数。这里是108。

ROT_TWO：

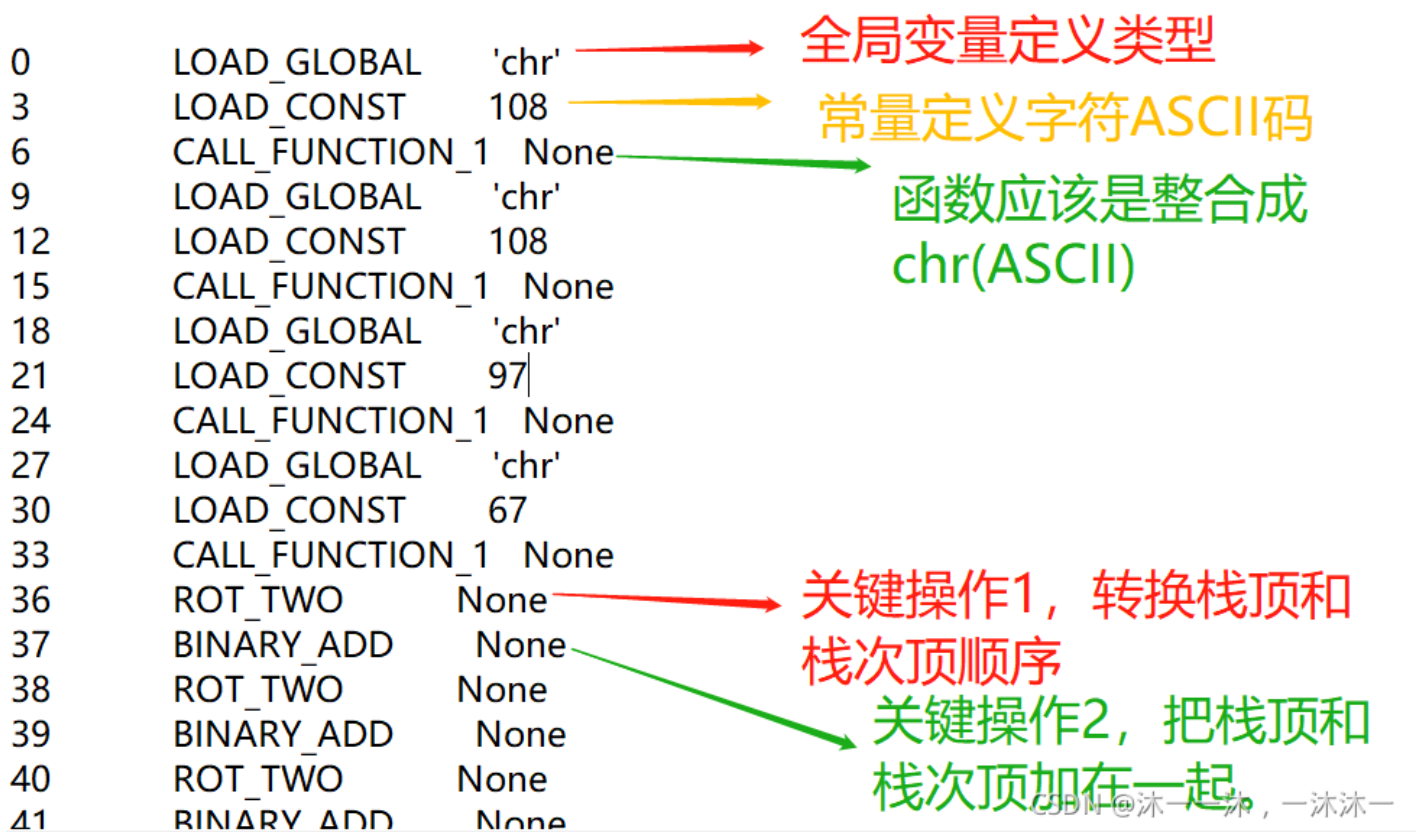
交换最顶部的两个堆栈项。

BINARY_ADD：

二元运算从堆栈中删除堆栈顶部 (TOS) 和第二个最顶部堆栈项 (TOS1)。他们执行操作，并将结果放回堆栈中，实施.TOS = TOS1 + TOS。这里是两个字符的相加，而不是ASCII码数字的相加。

那么这里的字节码我们应该这样理解：

```
# Embedded file name: <string>


def main--- This code section failed: ---

0       LOAD_GLOBAL      'chr'
3       LOAD_CONST       108
6       CALL_FUNCTION_1  None
9       LOAD_GLOBAL      'chr'
12      LOAD_CONST       108
15      CALL_FUNCTION_1  None
18      LOAD_GLOBAL      'chr'
21      LOAD_CONST       97
24      CALL_FUNCTION_1  None
27      LOAD_GLOBAL      'chr'
30      LOAD_CONST       67
33      CALL_FUNCTION_1  None
36      ROT_TWO          None
37      BINARY_ADD       None
38      ROT_TWO          None
39      BINARY_ADD       None
40      ROT_TWO          None
41      BINARY_ADD       None
```



所以这里前面一小部分的转换应该是这样的：

以字符形式加载常量108（l）、108（l）、97（a）、67（C），这个Python虚拟机入栈之后参数的顺序就是栈顶是‘l’、第二个值是‘l’、第三个值是‘a’、第四个值是‘C’，之后通过ROT_TWO和BINARY_ADD重新排列字符串：

36 ROT_TWO #交换栈顶的值‘l’和第二个值‘l’,变成‘l’和‘l’

37 BINARY_ADD #‘l’和‘l’字符相加，变成‘ll’存储在栈顶

38 ROT_TWO #交换栈顶的值‘ll’和第二个值‘a’,变成‘a’和‘ll’

39 BINARY_ADD #‘a’和‘ll’字符相加，变成‘all’存储在栈顶

40 ROT_TWO #交换栈顶的值‘all’和第二个值‘C’,变成‘C’和‘all’

41 BINARY_ADD #‘C’和‘all’字符相加，变成‘Call’存储在栈顶

因为有1000多行代码，所以必须用脚本批量执行才行，获取字符ASCII值和ROT_TWO、BINARY_ADD的操作位置，仿照逻辑输出即可：（所以这里也是代码截断重写）

这里在别人脚本学到的一个厉害的思路就是line = line.split()[-1] # 按空格切分取最后一个字符,就省去了复杂的正则表达式了

```
def BINARY_ADD(list1): # 模拟BINARY_ADD相加操作，pop方法可以模仿弹出栈（列表）

top1=list1.pop()

top2=list1.pop()

list1.append(top2+top1)
```

```
def ROT_TWO(list1): # 模拟ROT_TWO交换操作，pop方法可以模仿弹出栈（列表）
top1=list1.pop()
top2=list1.pop()
list1.append(top1)
list1.append(top2)
with open('3.py','r') as fd: # 将代码中的内容读取进来
lines=fd.readlines()
list1=[]
for line in lines: # 遍历每一行
if "LOAD_CONST" in line: # 包含LOAD_CONST
line = line.split()[-1] # 按空格切分取最后一个字符,就省去了复杂的正则表达式了
if line.isdigit(): # 如果是数字的话直接添加该字符
list1.append(chr(int(line)))
else:
list1.append(0) # 如果不是数字的话，栈中添0。代码333行中出现了None，即添加空字符常量
else:
if "BINARY_ADD" in line: # 如果包含BINARY_ADD，就执行相加操作
BINARY_ADD(list1)
elif "ROT_TWO" in line: # 如果包含ROT_TWO，就执行交换操作
ROT_TWO(list1)
print(list1) #正向复现整个流程，打印结果
```

结果：

```
└─$ python 2.py
[0, 'Call me a Python virtual machine! I can interpret Python bytecodes!!!', 'hitcon{Now yo
u can compile and run Python bytecode in your brain!}', 'password: ', 'Wrong password... Pl
ease try again. Do not brute force. =)']
```

D语言逆向分析（.d后缀）

攻防世界deedeedee：

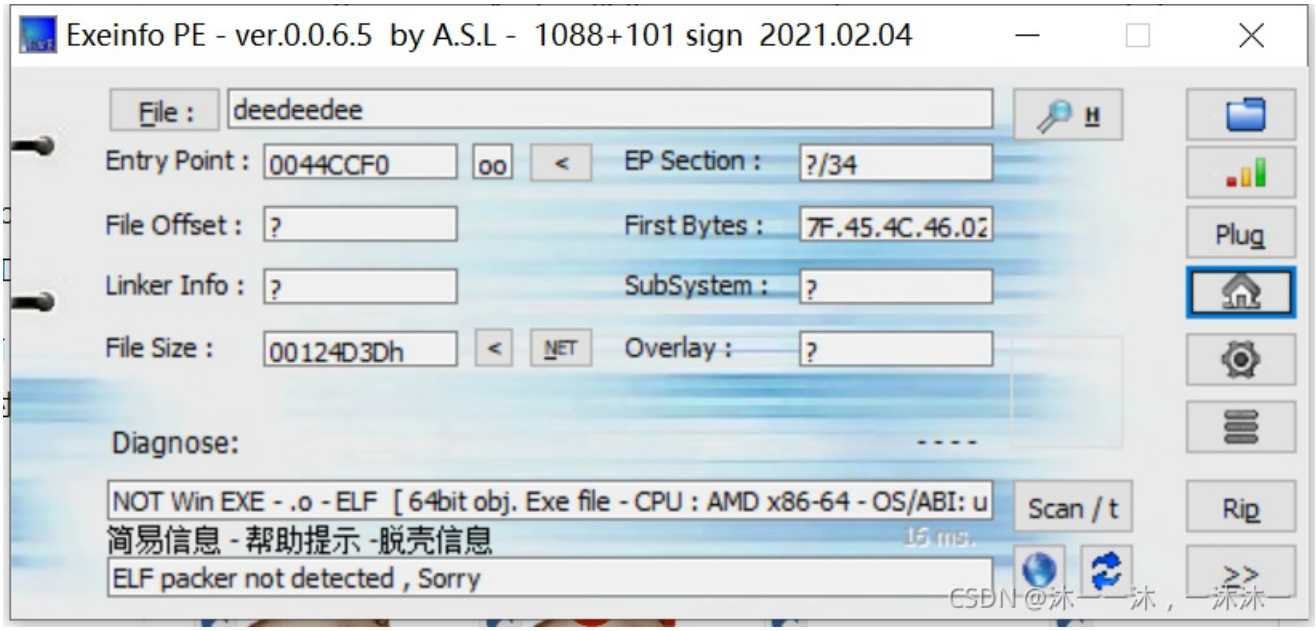
下载附件，这次还是两个附件：

额、两个有联系的附件题还是不知道怎么做的，各自用记事本打开看一下，一个是乱码，一个看起来是python语言：


```
 ELF  □□          > □   鹘D    @      楗□        @ 8
 @ "  □   □   @       @ @      @ @     0       0       □       □   □   p       p  @      p  @                □       □   □             @        @      ?□
     熸J     熸J     ,J      ,J      □        Q鈜d□                                                □         R鈜d□    P-□      P-l      P-l      ?       ?       □       /lib64/ld-l
H犿E□?€  %□€ ?  □`□□Q? $攊 苝聮E? €裑□    &□?□                ? □ @□□□( !hA□鑋D楁a□?  □□€   ? g?V □,P鳂⒆&??h  TE?$d?5?
A□□? ?Q0
d灑`D□暟□A□  €?Q??!PH□*   "XY□□□F @?@T□□D□a □□HT?□?褸?? □€  □□□□□?□ ?桷4?6
□`d□
H□□€5□8 14.□□□m?□`    □□%选瑚l        u€  *0X  $/F"爭Y□P亜睭?[@
                  0□€傻#???袗    □)F腎€D斉    □□?□@□a    □(   *0?□\□? ?`€□? B □省9   1□5"$垵   懜KM儶€?A 樌□?捊赣V
□@     P劍)?鋺?#□##傞d倠?□癳X□
€#? p€"?@□`刟?□掎  @b猔0F□J ?@漠? □?□J?□?@帢P案0?
$□T$□??□□幩□4T
棬  ?槵Lg?堡熸8?□B□
?? □琨?5毡?p  妻麦?□":儛l□
  @?? €(  □□□□2賕`E栽汼□癆?□悉           "(? 坉  □   H□0□J□  挠?6&悜?$V$□  鈥?D@妭?BX蠹□J□□(?癲&    @?? 湲□B"A?`  D □!@
□□□k□i榻□
K□N¨¨)?? □"??□d     剝 膓!□俸8  鋁嚚? □b □UF□□
6  ?? ???(?? 4□□  □JA ﾑ 8?□R?j卣)壛□"□$€@U\Gh楟□□诃!$ ?Dz□□□?□? ??□t□? 竦盔   □? `BCT輻aM□XcB0艪€€{?*P€垶袧? □□

湸0褦—Bz0□□. @?;H?□3   □    €J   @□?L□?'€□,     c@  "鈎B    K*    □兟€棬腎4床□□l^?@? □
?   ? `□
□鉞€(&□□€□  ?€??@□b?安□ yV詡?□ `□RbXD□@□A 8      y砵(□□Pc?@□?;T婊@□ 裹N€M□@琄(□9*  h
BDG卓
```


```
import std.stdio;
import std.range;
import std.algorithm;
import std.string;
import std.traits;
import std.conv;

template enc(string key) {
    string enc(string s) pure @safe {
        pragma(msg, "encrypting with key...", key);
        char enc_add = cast(char) (s.length.to!int);
        string output = "";
        foreach(a,b; cycle(key).zip(s)) {
            output ~= a ^ b ^ enc_add;
        }
        return output;
    }
}

mixin template ForeachUnrolled(string F, int N, int MIN=1) {
    pragma(msg, "ForEach...", N, " to ", MIN);
    static if (N > 1) {
        mixin ForeachUnrolled!(F, N-1, MIN);
```

突然想起来好像还没有扔入exeinfope中查看信息：



没有后缀的是64位ELF文件，照例扔入IDA中查看伪代码信息，可以看到红框位置处要输出加密代码的，动态调试一下也确实输出了。

```
1  __int64 Dmain()
2  {
3    unsigned __int64 v0; // rdx
4    __int64 v1; // rax
5    __int64 v2; // rdx
6
7    v0 = __readfsqword(0);
8    v1 = D9deedeedee9hexencodeFAyaZAya(
9           *(_QWORD *)((char *)&D9deedeedee8enc_flagAya + v0),
10          *(_QWORD *)((char *)&D9deedeedee8enc_flagAya + v0 + 8));
11   D3std5stdio20__T7writelnTAyaTAyaZ7writelnFNfAyaAyaZv(v1, v2, 36LL, (__int64)"Your hexencoded, encrypted flag is: ");
12   D3std5stdio16__T7writelnTAyaZ7writelnFNfAyaZv(34LL, "I generated it at compile time. :) ");
13   D3std5stdio16__T7writelnTAyaZ7writelnFNfAyaZv(26LL, "Can you decrypt it for me?");
14   return 0LL;
15 }
```

中文含义应该是输出加密flag


```
Your hexencoded, encrypted flag is: 676c60677a74326d716c6074325f6c6575347172316773616c6d686e665f68735e6773385e345e3377657379316e327d
```

但是拿着这串hexencode解密却发现不对劲，可能不是十六进制转字符：

加密或解密字符串长度不可以超过10M

```
676c60677a74326d716c6074325f6c6575347172316773616c6d686e665f68735e6773385e345e3377657379316e327d
```

16进制转字符 | 字符转16进制 | 测试用例 | 清空结果 | 复制结果


```
gl`gzt2mql`t2_leu4qr1gsalmhnf_hs^gs8^4^3wesy1n2}
```


然后浏览了一遍IDA并没有什么发现，上网查一下另一个类似python的作用，发现这个.d后缀的竟然是一种新语言D语言：

D语言 编辑

D语言，一种通用计算机程序语言，威力强大、功能丰富，支持多种编程范式，例如面向对象。

D语言最初由Digital Mars公司就职的Walter Bright于2001年发布，意图改进C++语言。目前最新D语言被简称为D2。最主要的D语言的实现是DMD。

D 语言源自C/C++，借鉴了众多编程语言的特色和现代编译器技术，融会贯通了设计者丰富的实践经验，使之具备了非凡的威力——既有 C/C++ 语言的强大威力，又有 Python 和 Ruby 的开发效率。它集众多系统级编程所需的功能于一身，例如垃圾回收、手工内存操作、契约式设计、高级模板技术、内嵌汇编、内置单元测试、Mixin 风格多继承、类 Java 包管理机制、内置同步机制、内建基本运行时信息。 [1]

中文名	D语言	D 不是	脚本语言也不是一种解释型语言
外文名	D Programming Language	适用各种	各种野心勃勃的编译器优化技术
D 语言	是一种通用的系统和应用编程语言	开发商	Digital Mars公司

好吧，现在java的APK逆向还没入手，没工夫去学这个D语言了，得先搞定java逆向先。然后查着查着发现flag在.d文件里面，这个.d文件就是64位ELF程序的源码啊：

```
}

string hexencode(string s) {
    string encoded = "";
    foreach (c; s) {
        encoded ~= format("%02x", c);
    }
    return encoded;
}

string enc_flag = encrypt("flag{t3mplat3_met4pr0gramming_is_gr8_4_3very0n3}");
void main() {
    writeln("Your hexencoded, encrypted flag is: ", enc_flag.hexencode);
    writeln("I generated it at compile time. :)");
    writeln("Can you decrypt it for me?");
}
```


RC4解密脚本：

```
#include<stdio.h>

void rc4_init(unsigned char* s, unsigned char* key, unsigned long Len_k) //初始化函数

{

int i = 0, j = 0;

char k[256] = { 0 };

unsigned char tmp = 0;

for (i = 0; i < 256; i++) {

s[i] = i;

k[i] = key[i % Len_k];

}

for (i = 0; i < 256; i++) {

j = (j + s[i] + k[i]) % 256;

tmp = s[i];

s[i] = s[j];

s[j] = tmp;

}

}

void rc4_crypt(unsigned char* Data, unsigned long Len_D, unsigned char* key, unsigned long Len_k) //加解密
```

```
{
unsigned char s[256];
rc4_init(s, key, Len_k);
int i = 0, j = 0, t = 0;
unsigned long k = 0;
unsigned char tmp;
for (k = 0; k < Len_D; k++) {
i = (i + 1) % 256;
j = (j + s[i]) % 256;
tmp = s[i];
s[i] = s[j];
s[j] = tmp;
t = (s[i] + s[j]) % 256;
Data[k] = Data[k] ^ s[t];
}
}
void main()
{
unsigned char key[] = "[Warnning]Access_Unauthorized"; //密钥
unsigned long key_len = sizeof(key) - 1;
unsigned char data[] = { 0xC3,0x82,0xA3,0x25,0xF6,0x4C,
0x36,0x3B,0x59,0xCC,0xC4,0xE9,0xF1,0xB5,0x32,0x18,0xB1,
0x96,0xAe,0xBF,0x08,0x35};//把加密脚本放在kali中打开，根据kali推荐的编码再复制到burp中用hex编码写出来放在这里。 kali ISO-8859-1编码显示类似于 ??£%?L6;Yì?é?µ2±???5
int i;
rc4_crypt(data, sizeof(data), key, key_len);
for ( i = 0; i < sizeof(data); i++)
{
printf("%c", data[i]);
}
printf("\n");
```

return;

}

//flag{RC4&->ENc0d3F1le}

INT3断点：

INT3断点，简单地说就是将你要断下的指令地址处的第一个字节设置为0xCC，软件执行到0xCC（对应汇编指令INT3）时，会触发异常代码为EXCEPTION_BREAKPOINT的异常。这样我们的调试程序就能够接收到这个异常，然后进行相应的处理。如果没有处理就会强制退出程序，就无法调试了。

INT3断点的信息结构体如下：

struct stuPointInfo

{

PointType ptType; //断点类型

int nPtNum; //断点序号

LPVOID lpPointAddr; //断点地址

BOOL isOnlyOne; //是否一次性断点（针对INT3断点）

char chOldByte; //原先的字节（针对INT3断点）

};

而每一个INT3断点信息结构体指针又保存到一个链表中。

INT3断点的设置：

设置INT3断点比较简单，只需要根据用户输入的断点地址和断点类型（是否一次性断点），将被调试进程中对应地址处的字节替换为0xCC，同时将原来的字节保存到INT3断点信息结构体中，并将该结构体的指针加入到断点链表中。

如果在被调试的某地址处已经存在一个同样的断点了，那么用户 还要往这个地址上设置相同的断点，则必然会因为重复设置断点导致错误。例如这里的INT3断点，如果不对用户输入的地址进行是否重复的检查，而让用户在同 一个地址先后下了两次INT3断点，则后一次INT3断点会误以为这里本来的字节就是0xCC而一直断点在这里。

所以在设置INT3断点之前应该先看该地址是否已经下过 INT3断点，如果该地址已经存在一个INT3断点，且是非一次性的，则不能再在此地址下INT3断点，如果该地址有一个INT3一次性断点，而用户要继续下一个INT3非一次性断点，则将原来存在的INT3断点的属性从一次性改为非一次性断点。

INT3断点被断下的处理：

INT3断点被断下后，首先从断点链表中找到对应的断点信息结构体。如果没有找到，则说明该INT3断点不是用户下的断点，调试器不做处理，交给系统程序去处理（其他类型的断点触发异常也需要做同样的处理，而系统通常是强制退出结束程序），这也是攻防世界的csaw2013reversing2的考点。

如果找到对应的断点，根据断点信息将断下地址处的字节还原为原来的字节，并将被调试进程的 EIP减一，因为INT3异常被断下后，被调试进程的EIP已经指向了INT3指令后的下一条指令，所以为了让被调试进程执行本来需要执行的指令，应该让 其EIP减1。

如以下代码：

地址 机器码 汇编代码

01001959 55 push ebp

0100195A 33ED xor ebp,ebp

0100195C 3BCD cmp ecx,ebp

0100195E 896C24 04 mov dword ptr ss:[esp+4],ebp

当用户在0100195A地址处设置INT3断点后，0100195A处的字节将改变为0xCC（原先是0x33）。

此时对应的代码如下：

地址 机器码 汇编代码

01001959 55 push ebp

0100195A CC int3

0100195B ED in eax,dx

0100195C 3BCD cmp ecx,ebp

0100195E 896C24 04 mov dword ptr ss:[esp+4],ebp

当被调试程序执行到0100195A地址处，触发异常，进入异常处理程序后，获取被调试线程的环境（GetThreadContext），可看 出此时EIP指向了0100195B，也就是INT3指令之后，所以我们除了要恢复0xCC为原来的字节之外，还要将被调试线程的EIP减一，让EIP指 向0100195A。否则CPU就会执行0100195B处的指令（0100195B ED in eax,dx），显然这是错误的。

如果查找到的断点信息显示该INT3断点是一个非一次性断点，那么需要设置单步，然后在进入单步后将这一个断点重新设置上（硬件执行断点和内存 断点如果是非一次性的也需要做相同的处理）。因为INT3断点同时只会断下一个，所以可以用一个临时变量保存要重新设置的INT3断点的地址，然后用一个 BOOL变量表示当前是否有需要重新设置的INT3断点。

关于INT3断点的一些细节：

1. 创建调试进程后，为了能够让被调试程序断在OEP（程序入口点），我们可以在被调试程序的OEP处下一个一次性INT3断点。

2. 在创建调试进程的过程中（程序还没有执行到OEP处），会触发一个ntdll.dll中的INT3，遇到这个断点直接跳出不处理。这个断点在使用微 软自己的调试工具WinDbg时会被断下，可以猜测，微软设置这个断点是为了能够在程序到达OEP之前就被断下，方便用户做一些处理（如设置各种断点）。

3. 因为INT3断点修改了被调试程序的代码内容，所以在进行反汇编和显示被调试进程内存数据的时候，需要检查碰到的0xCC字节是否是用户所下的 INT3断点，如果是需要替换为原来的字节，再做相应的反汇编和显示数据工作。这一点olldbg做的很不错，而有一些国产的调试器好像没有注意到这些小 的细节。

ESP脱壳定律：

https://www.52pojie.cn/thread-236872-1-1.html

狭义ESP定律

ESP定律的原理就是“堆栈平衡”原理。

让我们来到程序的入口处看看吧！

涉及的汇编知识：

call命令：

1.向堆栈中压入下一行程序的地址；

2.JMP到call的子程序地址处。

例如：

00401029.E8 DA240A00 call 004A3508

0040102E.5A pop edx

//在执行了00401029以后，程序会将0040102E（下一条指令地址）压入堆栈，然后JMP到004A3508地址处！

RETN命令：

1.将当前的ESP中指向的地址出栈；

2.JMP到这个地址。

这个就完成了一次调用子程序的过程。在这里关键的地方是：如果我们要返回父程序，则当我们在堆栈中进行堆栈的操作的时候，一定要保证在RETN这条指令之前，ESP指向的是我们压入栈中的地址。这也就是著名的“堆栈平衡”原理！

1.这个是加了ASPACK壳程序的入口时各个寄存器的值：（入壳，解压缩）

EAX 00000000

ECX 0012FFB0

EDX 7FFE0304 //堆栈值

EBX 7FFDF000 //堆栈值

ESP 0012FFC4

EBP 0012FFF0

ESI 77F57D70 ntdll.77F57D70

EDI 77F944A8 ntdll.77F944A8

EIP 0040D000 ASPACK.<ModuleEntryPoint>

2.这个是ASPACK壳JMP到OEP（Original Enter Point）后的寄存器的值：（出壳，回主程序OEP）

EAX 004010CC ASPACK.004010CC //保存了当前OEP的值

ECX 0012FFB0

EDX 7FFE0304 //堆栈值

EBX 7FFDF000 //堆栈值

ESP 0012FFC4

EBP 0012FFF0

ESI 77F57D70 ntdll.77F57D70

EDI 77F944A8 ntdll.77F944A8

EIP 004010CC ASPACK.004010CC

呵呵~是不是除了EIP不同以外，eax保存当前OEP值，其他都一模一样啊！

为什么会这样呢？我们来看看ASPACK加壳程序的起始代码：（压栈、入壳解压缩）

0040D000 A> 60 pushad

//PUSHAD就是把所有寄存器压栈！注意这里压入寄存器后ESP=0012FFC4

0040D001 E8 00000000 call ASPACK.0040D006

//这里压入地址后ESP=0012FFA4，压入的地址是call语句的下一行地址，存储在栈段中的A4段。然后开始调用的是壳内的代码，来给后面压缩的代码解压缩。

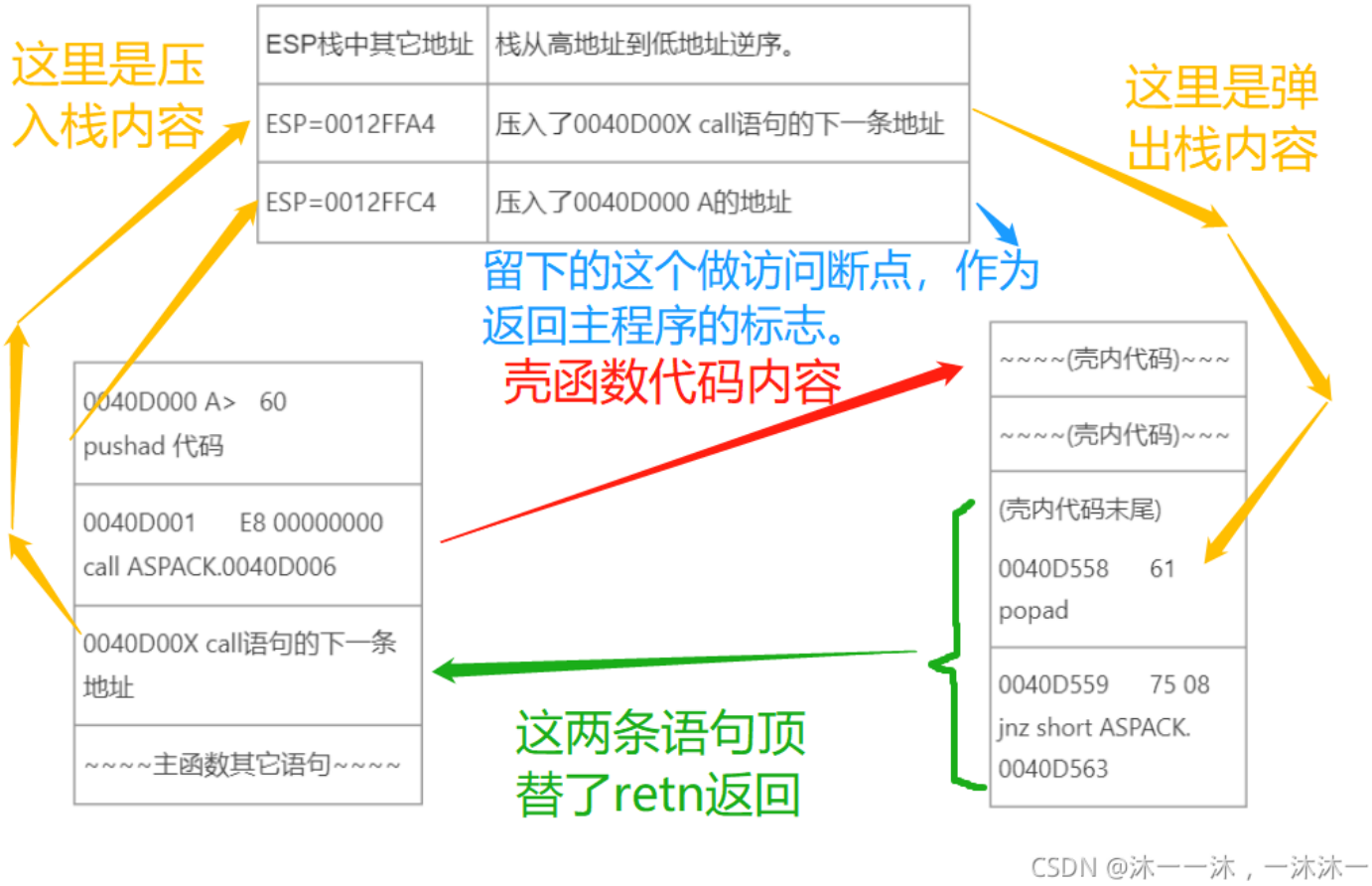
我们在到壳的最后看看：（弹栈、跳出壳外OEP）

0040D558 61 popad

//这里弹出的是ESP=0012FFA4存储的值，是前面call语句的下一行地址，然后栈段中就剩下ESP=0012FFC4了。

0040D559 75 08 jnz short ASPACK.0040D563

//注意这里ESP=0012FFC4，这里准备跳出壳外代码，用pop和Jmp操作代替壳内函数的retn语句，跳到call的下一行语句的地址处，ESP=0012FFC4还是留在了里面，于是这可以下访问断点，作为回到主程序的依据。



也就是说当我们对ESP的0012FFA4栈地址处下硬件访问断点之后。当程序要通过堆栈访问这些值，从而恢复原来寄存器的值，准备跳向苦苦寻觅的OEP的时候，OD帮助我们中断下来。

小结：我们可以把壳假设为一个子程序，当壳把代码解压前和解压后，他必须要做的是遵循堆栈平衡的原理。

1.ESP定律的原理是什么？

堆栈平衡原理。

2.ESP定律的适用范围是什么？

几乎全部的压缩壳，部分加密壳。只要是在JMP到OEP后，ESP=0012FFC4的壳，理论上我们都可以使用。但是在何时下断点避开校验，何时下断OD才能断下来，这还需要多多总结和多多积累。

3.是不是只能下断12FFA4的访问断点？

当然不是，那只是ESP定律的一个体现，我们运用的是ESP定律的原理，而不应该是他的具体数值，不能说12FFA4，或者12FFC0就是ESP定律，他们只是ESP定律的一个应用罢了！

什么是内存断点？

内存断点等效于命令bpm，他的中断要用到DR0-DR7的调试寄存器，也就是说OD通过这些DR0-DR7的调试寄存器来判断是否断下普通断点（F2下断）等效于bpx，他是在所执行的代码的当前地址的一个字节修改为CC（int3）。当程序运行到int3的时候就会产生一个异常，而这个异常将交给OD 处理，把这个异常给EIP-1（所以才会停在上一条代码处）以后，就正好停在了需要的中断的地方（这个根据系统不同会不一样），同时OD在把上面的int3修改回原来的代码。

内存断点分为：

内存访问断点，内存写入断点。

我们知道，在程序运行的时候会有3种基本的状态产生：

读取–>写入–>执行。

004AE242 A1 00104000 mov eax,dword ptr ds:[004AE24C] //004AE24C处的内存读取

004AE247 A3 00104000 mov dword ptr ds:[004AE24C],eax //004AE24C处的内存写入

004AE24C 83C0 01 add eax,1 //004AE24C处的内存执行

1.当对004AE24C下内存访问断点的时候，可以中断在004AE242也可以中断在004AE247和004AE24C。(因为访问包含执行、写入、读取。)

2.当对004AE24C下内存写入断点的时候，只能中断在004AE247。

3．当执行004AE24C的时候，只能中断在004AE24C。

如何在寻找OEP时使用内存断点?（解压为主函数code段写入，跳转为主函数code段执行）

我们要知道壳如果要把原来加密或压缩的代码运行起来就必须要解压和解密原来的代码。而这一个过程我们可以看做是对代码段（code段）的写入。

解压完毕后，我们要从壳代码的区段JMP到原来的代码段的时候，可以看成是对代码段（code段）的执行。

理清了上面的关系就好办了，那么如果载入OD后，我们直接对code段下内存访问断点的时候，一定会中断在壳对code段的写入的代码的上面，就像上面的 004AE247的这一行。而如果当他把code段的代码全部解压解密完毕了以后，JMP到OEP的时候，我们还会停在OEP的代码上面呢，而且每按下F9都会中断，因为这时code段在执行中，而我们下的是访问断点。

如果我们不在JMP OEP上面下断点而是在一开始就下断点，正入我上面所说的，如果你在前面下断很可能壳对code段还没解压完毕呢，这时如果你不停的按F9，你将会看到OD的下方不断的在提示你，“对 401000写入中断”“对401002写入中断”“对401004写入中断”。。。。

更便捷的两次内存断点：

假设我是一个壳的作者，一个EXE文件的有code段，data段，rsrc段…依次排列在你的内存空间中，那么我会怎么解码呢？

我会先将code段解码，然后再将data段解压，接着是rsrc段…那么你不难发现，只要你在data段或者rsrc段下内存访问断点，那么中断的时候code段就已经解压完毕了。这时我们再对code段下内存访问断点，不就可以到达OEP了吗？

这里注意上面虽然下了两次内存访问断点，但是本质是不一样的，目的也是不一样的：

1.对data段下内存访问断点而中断是因为内存写入中断，目的是断在对data段的解压时，这时壳要对data段写数据，但是code段已经解压完毕。

2.对code段下内存访问断点而中断是因为内存执行中断，目的当然就是寻找OEP了。

总结一下：

如果我们知道壳在什么地方对code段解压完毕我们就可以使用内存断点，找到OEP。

如果不知道，那么我们就依靠2次内存断点去找，如果还不行就用多次内存断点。总之明白了原理在多次的内存断点其实都一样。从这个过程中我们了解的是壳在对区段解码的顺序！