

**课 程 实 验 报 告**

**课程名称： 逆向工程分析技术**

**专业班级： 信息安全 1805 班**

**学 号： U201810679**

**姓 名： 蒋树齐**

**指导教师： 鲁宏伟**

**报告日期： 2021年3月26日**

**网络空间安全学院**

1. **实验准备**

（1）下载IDA\_Pro\_v7.0版本并安装调试。

（2）配置老师给出的CSDN上的代码，理解PE文件。

1. **实验内容**

（1）编程（Java、C/C++或Python）分析PE文件（experient-1.exe）的结构，提取文件相关信息，描述文件的主要参数和构成；

（2）结合程序实例，分析函数“sub\_4A19B0”出现“栈平衡”问题的原因，并根据程序运行流程，借助IDA工具调整使其达到栈平衡；

（3）在上述工作基础上，尝试分析实例代码中的“Flag”。

1. **实验目标**

（1）通过自己编写代码或者利用工具加深对PE文件格式的理解。

（2）掌握反汇编工具的使用方法以及简单的软件逆向分析过程。

1. **分析方法和过程**

## **4.2.PE文件信息输出**

直接利用老师给的链接的代码，修改字符集，然后运行，可以看到输出了以下的信息。

但是直接调用别人的代码完成实验或许也太草率了，虽然代码看的并不是很懂，但是为了加深对PE文件格式的理解，我们可以从输出的信息中分析并理解PE文件。

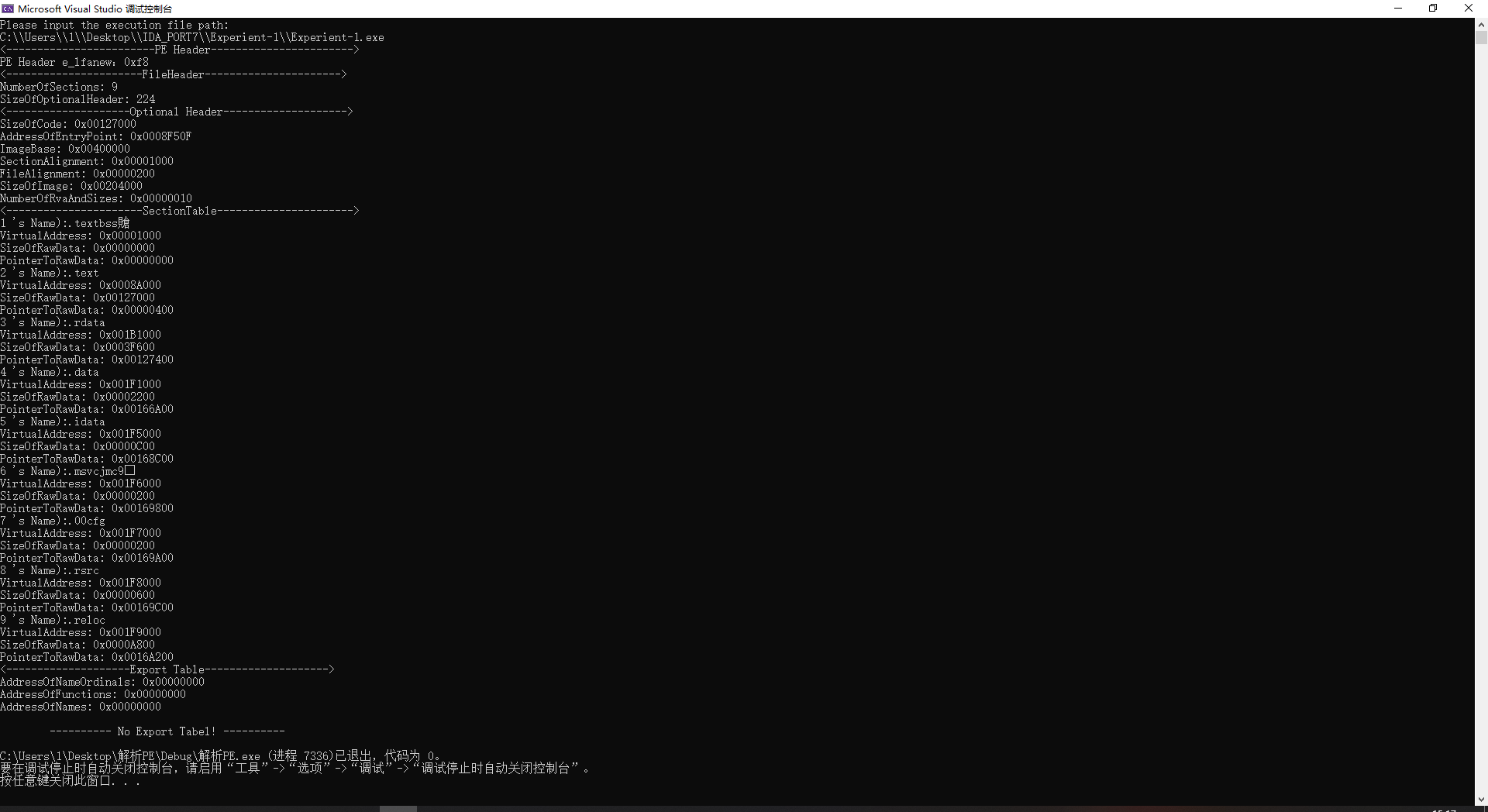


图 1输出信息

1. PE Header：PE Header是PE相关结构NT映像头（IMAGE\_NT\_HEADER）的简称，其中包含许多PE装载器用到的重要字段。执行体在支持PE文件结构的操作系统中执行时，PE装载器将从IMAGE\_DOS\_HEADER结构中的e\_lfanew字段里找到PE Header的起始偏移量，加上基址得到PE文件头的指针。PNTHeader = ImageBase + dosHeader->e\_lfanew

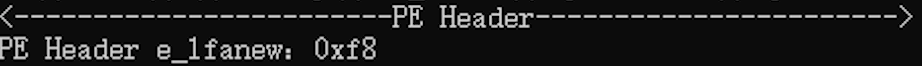


图 2PE Header

1. FileHeader：结构包含了文件的物理层信息及文件属性。该程序文件的区块数目是9，IMAGE\_OPTIONAL\_HEADER32结构的大小为224个字节。

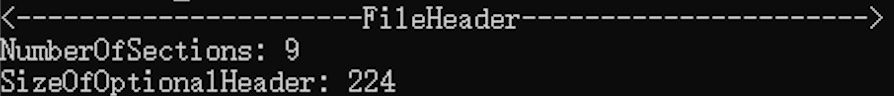


图 3FileHeader

1. Optional Header： 为可选映像头，是为了补充Image\_FILE\_HEADER结构不足以定义的PE文件属性。从下图信息中可以看出，所有含有代码表的总大小、程序执行入口地址、程序默认装入基地址、内存中表的对齐值、文件中表的对齐值、映像装入内存后的总大小、数据目录表的项数。

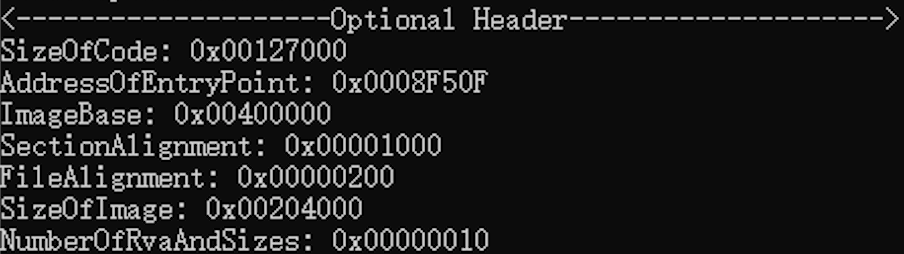


图 4Optional Header

1. Section Table：区块表是一个IMAGE\_SECTION\_HEADER结构数组，包含每个块在映像中的信息，且分别指向不同的section实体。以一个空的IMAGE\_SECTION\_HEADER结构作为结束。如图5所示为“.textbss”的区块实体示例。本exe文件共包括“ .textbss”、“.text”、“.rdata”、“ .data”、“ .idata”、“.msvcjmc”、“.00cfg”、“.rsrc”、“.reloc”

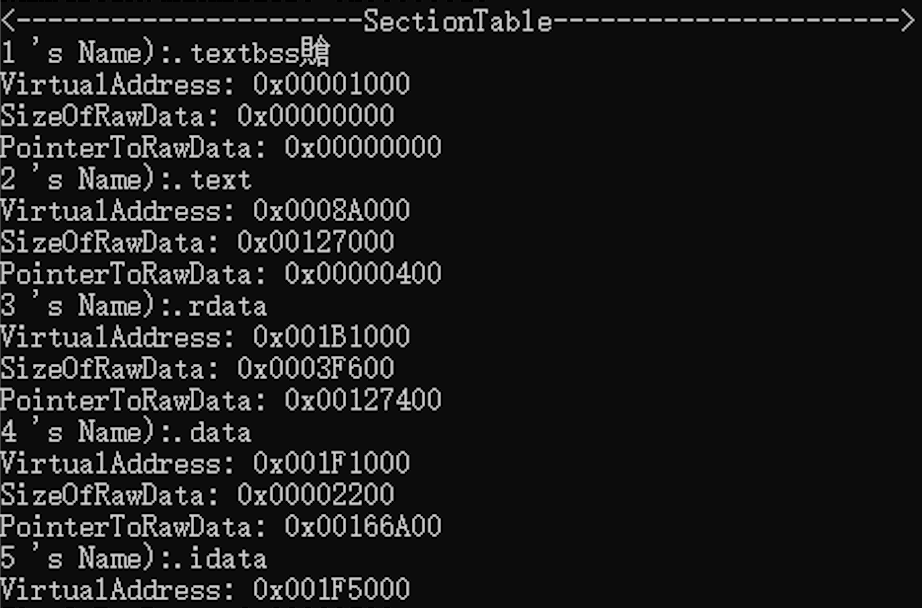


图 5Section Table

1. Export Table：输出表包含函数名称，输出序数等，序数是指定DLL中某个函数的16位数字，在所指向的DLL里是独一无二的。该图显示了输出表的指向输出序号数组、指向函数地址数组和函数名字的指针地址。

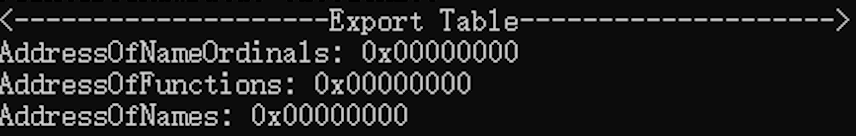


图 6Export Table

## **4.2.逆向程序实例**

为了更加清晰的表达实验过程和内容，我依照老师上课带着我们做的讲述方法描绘一遍实验的具体过程和自己内心的所思所想。

Step1:运行.exe文件获取实验的线索，先运行程序，运行图如下，可以看到程序中暴露了一串字符串。

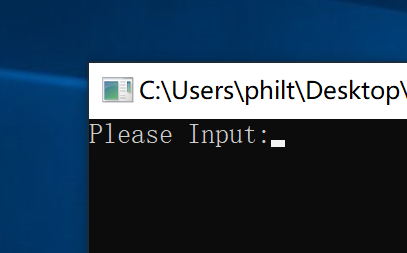


图 7字符串信息

Step2:使用IDA\_Pro\_7.0打开“Experient-1.exe”文件，接下来就是对线索字符串的寻找。使用快捷键Shift+F12快速打开Strings window。并在其中寻找“Please Input：”结果如下图所示：

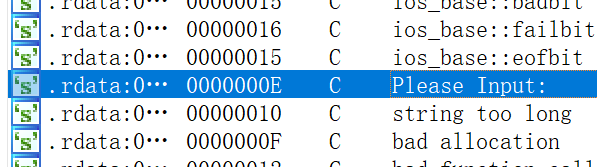


图 8字符串定位

定位该字符串后，从其交叉引用中定位到目标函数“sub\_4A19B0”

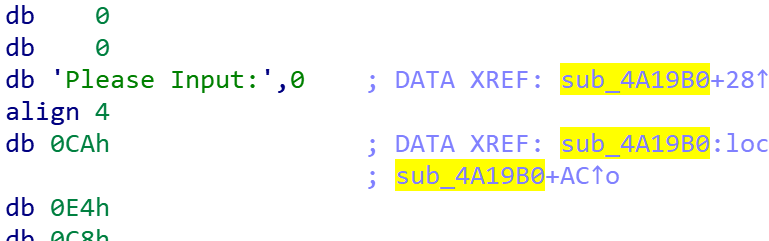


图 9字符串交叉引用的函数

Step3:双击进入该函数，可见汇编代码，执行反汇编，在IDAPRO中为按下F5按键。可见以下报错，报错信息为“positive sp value”

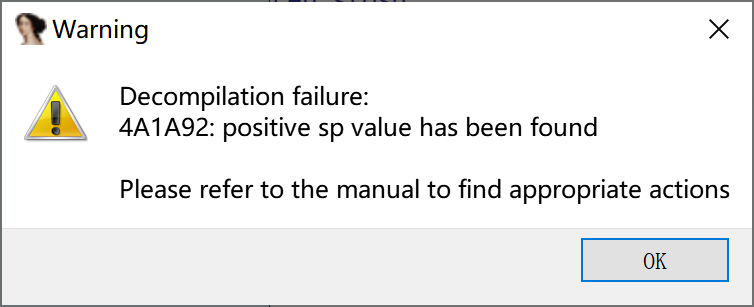


图 10报错信息

可以知道报错信息为栈平衡问题（其实在实验之前就知道了）意思是在函数结束时，栈指针sp应该为0，但是此时出现的数值为“positive”，即一个正数，说明在 push后缺少了pop。接下来开始排查。

Step4：为了更加便捷的排查栈的信息，可以直接使用IDAPRO自带的栈功能，点击Options->General->Stack pointer。

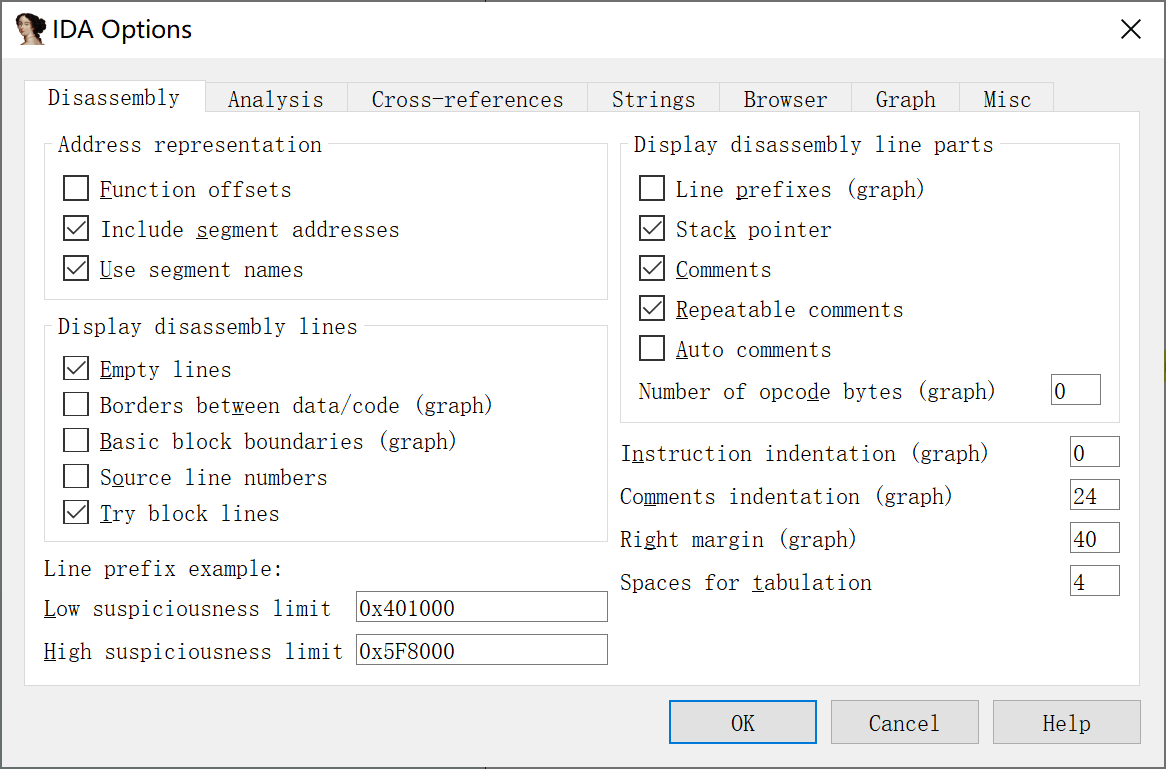


图 11勾选处

勾选后就能看到堆栈的长度了，观察报错处的堆栈信息如下，可以看到堆栈长度并未归0。有8个长度。结果如下：

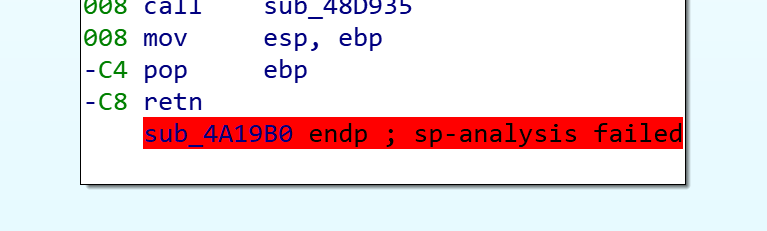


图 12错误位置堆栈信息

在函数内会执行push和pop的操作，其中push+4，pop-4。栈的长度主要由这两个指令发生变化。

Step5:顺着代码向下寻找，在函数开头保护现场后，堆栈值为0DC，此相当于函数栈为空时的值。当正常操作而函数堆栈值非此值时，说明之前的语句中涉及对堆栈的破坏。

结果为寻找到两处“push 0”，但未修正栈。两次push恰好使得栈变为8。出错位置截图如下：

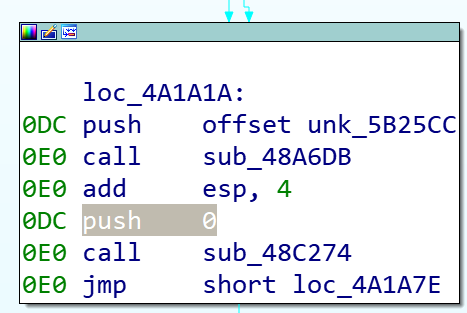


图 13第一处错误

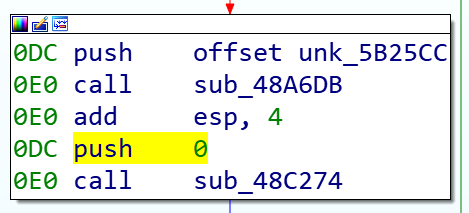


图 14第二处错误

但是也不能平白无故的断定这个0是错误根源，于是进入“sub\_48C274”观察函数内部，发现并没有和0相关的内容。函数内部如下：

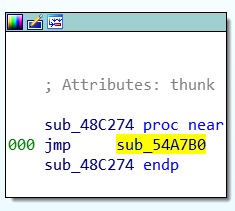


图 15sub\_48C274

故只需要修改掉两处push 0 即可让堆栈恢复正常。

Step6:修改指令：点击Options->General->Number of opcode bytes(graph)修改为16即可。

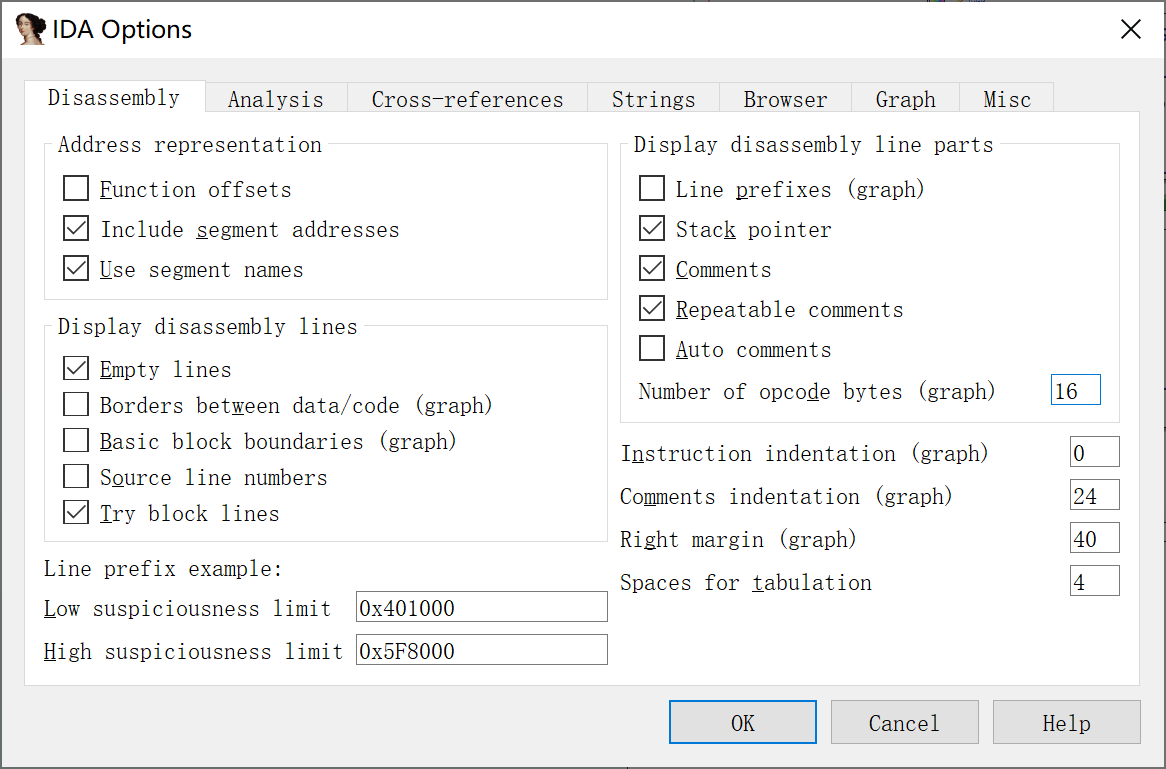


图 16勾选

然后选择对应的地址，点击：Edit->Patch Program->Change byte,修改为90 90即可。修改完后的指令如下：

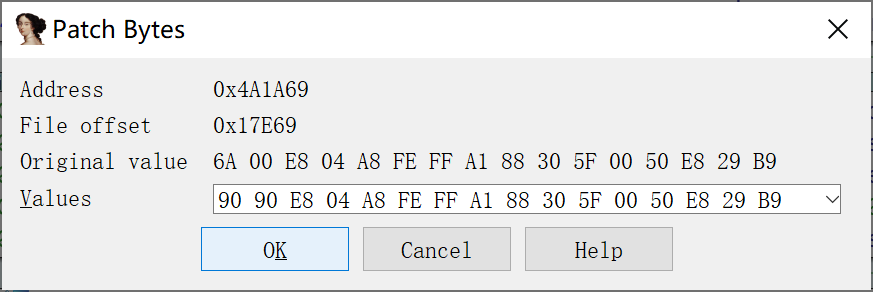


图 17修改界面

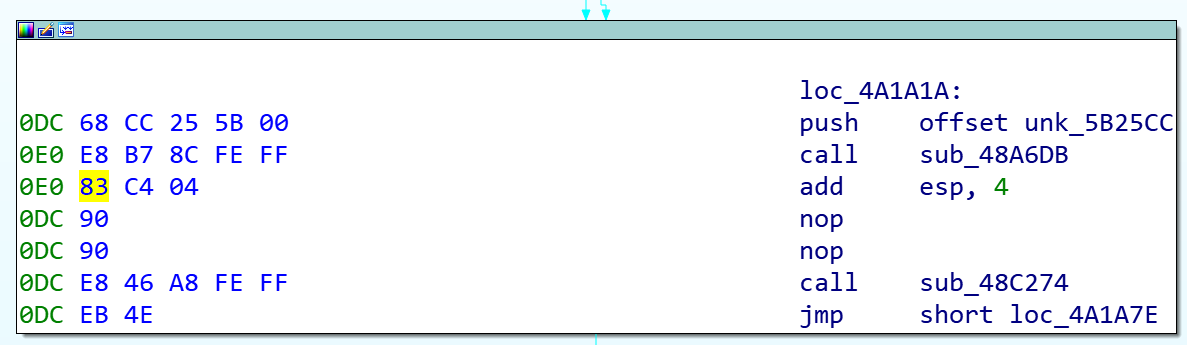


图 18修改后-1

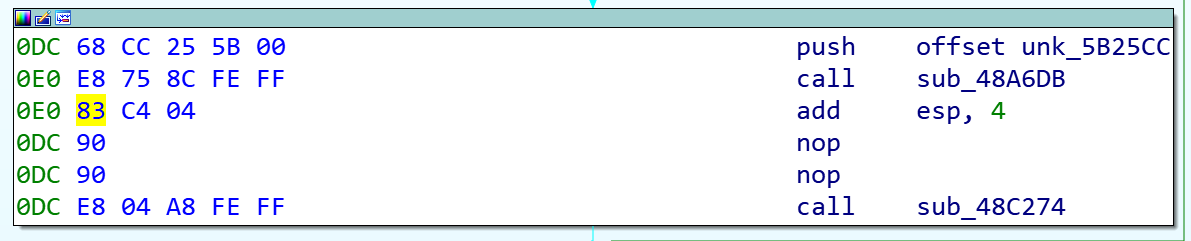


图 19修改后-2

修改完后为了验证结果是否正确，按下F5进行反编译。终于成功，编译生成的C代码如下图所示！

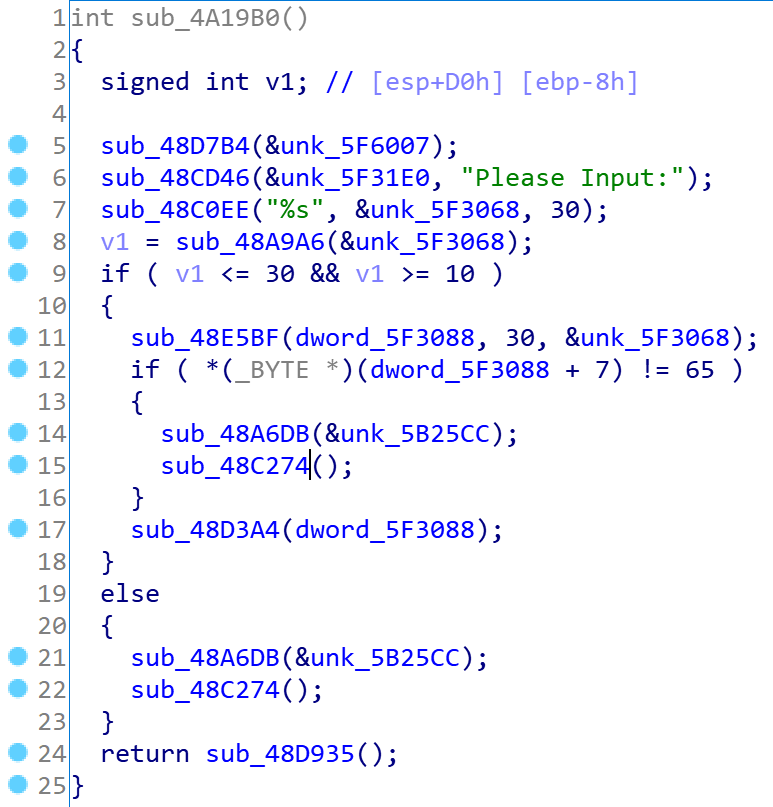


图 20反编译结果！

**4.2.“Flag”分析与追踪**

在获得C语言代码后，对各种函数进行观察，可以有一些较为明显的结论，因为有的函数有着比较明显的特征，比如sub\_48C0EE中自带的参数“%s”，十分类似与C语言的scanf函数，sub\_48CD46又类似“printf”输出信息，因为程序的最开始我们看到的就是Please Input：的字符串。

在获取以下线索后，继续往下分析。观察变量v1的行为，是接受字符串后对字符串进行操作并将返回值赋予给v1，观察v1的声明，为signed int，结合后方对v1的判断可以并结合软件安全中数组越界的知识，可以猜测函数sub\_48A9A6可能为“strlen”函数。以此对输入串的长度进行把控。

接下来的函数是对v1的长度进行把控，只有当满足规定范围内的长度时才进行后续的判断，不满足会调用两个函数进行操作。在v1满足长度判断的分支内对“dword\_5F3088+7”再进行条件判断，比较数65实际上是ascii对应的‘A’，当两方不相等时，调用两函数。发现两次条件不满足时调用的两函数均为sub\_48A6D8与sub\_48C274，且参数与顺序相同，大概率为报错函数，用于在输入字符串不是正确Flag时的提示信息输出。

抛开sub\_48A6D8与sub\_48C274这两个在条件不符合时会-出现的功能，最重要的需要分析的函数就是sub\_48D34A。她操作的对象是dword\_5F3088。但是程序的输入确是存储在unk\_5F3068中的，可以推测sub\_48E5BF可能是将3068的内容赋值给3088，然后对3088进行判断和操作。

那么进入sub\_48D34A观察。

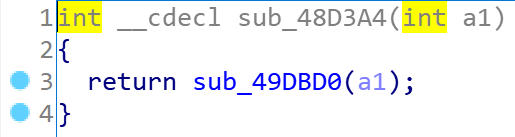


图 21sub\_48D34A内部

发现其又调用函数sub\_49DBD0，进入函数sub\_49DBD0观察。

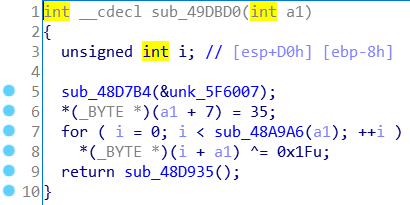


图 22sub\_49DBD0内部

我们可以看到a1被修改，具体操作为在与a1首地址偏移距离为7的字节处，修改为ascii的35，查表可以知道是‘#’，然后又把a1的所有字符和0x1F想异或，最后返回sub\_48D935，但是sub\_48D935并没有对a1的操作，所以暂时可以忽略不计。

回溯到3088，这是线索的核心，单击它，右键选中Jump to xref，获取其他调用处的线索。

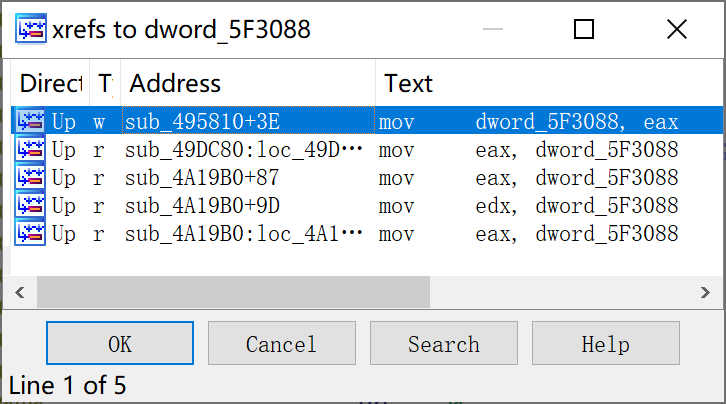


图 23 3088的xref

由于sub\_4A19B0就是反编译出的c函数本身，所以暂时可以忽略，那么还有两个函数分别是sub\_495810和sub\_49DC810就十分的可疑了。

首先进入sub\_495810观察，对3088进行了一次赋值，深入赋值函数发现是多个函数不断嵌套，参数均为a1。

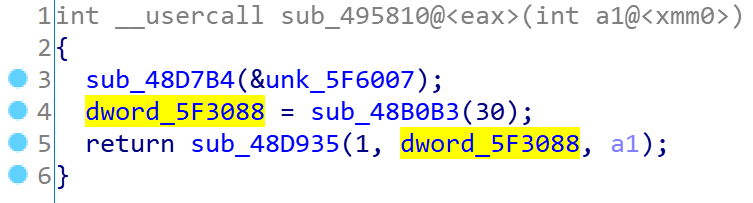


图 24sub\_495810内部

接下来进入sub\_49DC80观察，发现其对a1进行判断然后操作，具体的操作是将其每个字符与0x1C异或，然后进行一次判断，若使得sub\_48DB42的返回值为0，则进入一个循环，其中111和107对应的ascii为o和k，并且5F31E0之前被用作输出，所以大胆猜测这是满足条件后输出“ok”的字符串，大概率就是exe文件输入了正确字符串即“Flag”时，在程序中给出的反馈，告诉你输入正确了。

基于以上的分析，和字符串匹配的，sub\_48DB42就是c语言中的strcmp函数，当字符串一致时，返回0。

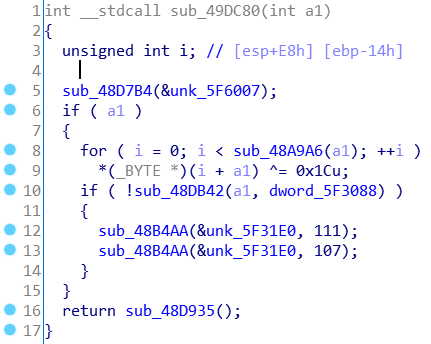


图 25sub\_49DC80内部

那么接下来的问题就在于这么找到a1的实际值，只要得到a1的值，就能通过异或运算获得最终的“Flag”。

现在顺着sub\_49DC80的交叉引用去寻找新线索，发现其在sub\_48DACA被调用，具体如下：

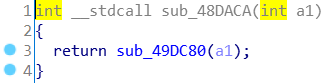


图 26sub\_48DACA

顺藤摸瓜，又找到sub\_49CEB0：

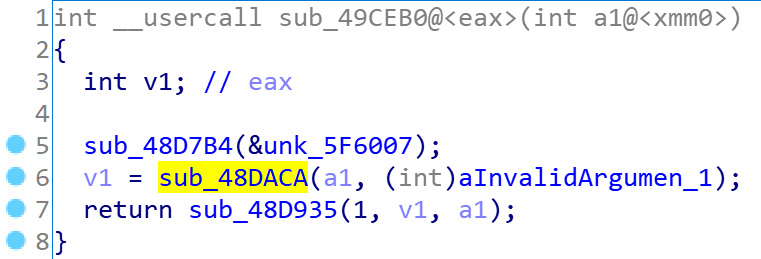


图 27sub\_49CEB0内部

不难发现aInvalidArgumen\_1就是a1的实际值，双击获得其对应的字符串，如下图所示：

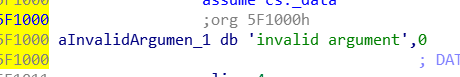


图 28 找到flag

接下来就是对该串进行一定的处理，具体的处理方法和之前分析过程中遇到的异或一致，与“0x1C”异或，输入的第8个字符为‘A’，‘A’被替换成‘#’后与“0x1F”异或。故我们可以将当前串即“invalid argument”与“0x1C”和“0x1F”异或，再将异或结果的第8个字符替换为‘A’。这便是最终的Flag！

点击File->Script Command->Scripting language->Python。然后编写程序把字符串进行转化，具体程序如下：

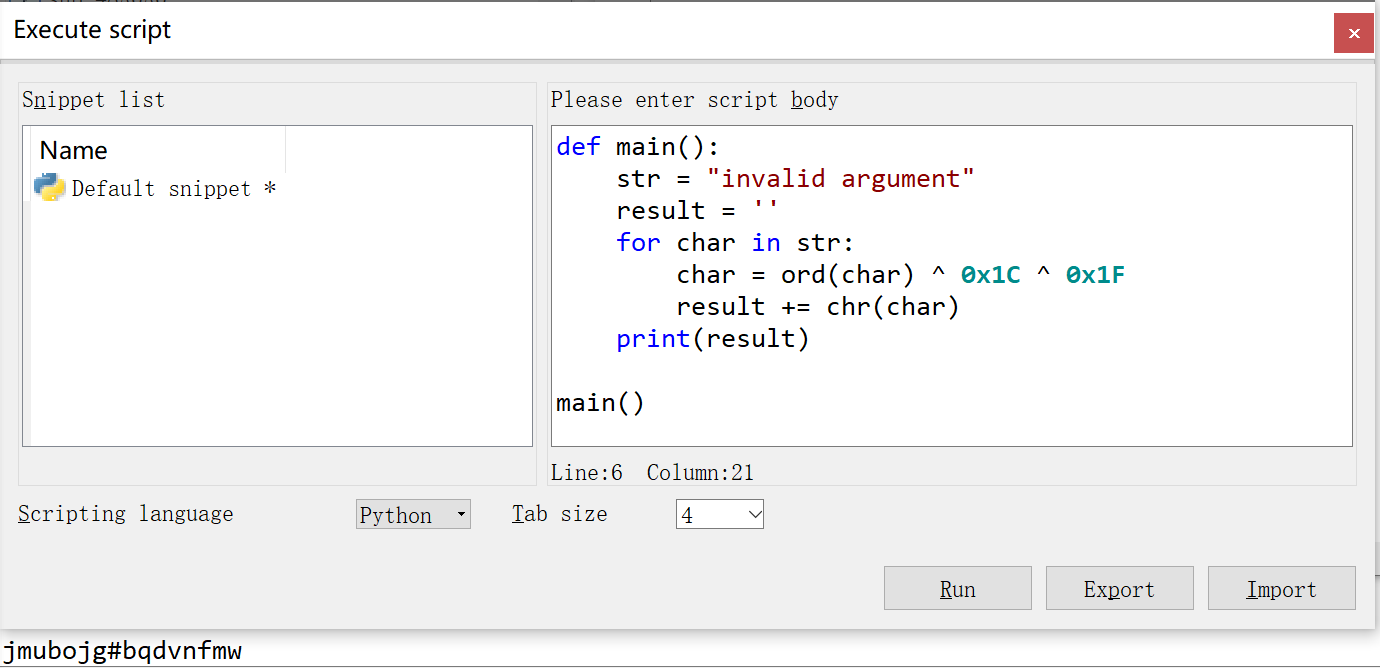


图 29Python程序

最后测试结果是否正确，打开程序输入Flag，但是程序闪退，为了保证能看到结果，选择在cmd内打开程序，避免程序运行过快看不到结果，最终显示ok效果如下，破解成功！



图 30破解成功！！！

1. **实验结论**

三个实验均顺利完成，结果均在上一部分展示，此处不赘述，在4分析方法和过程中不仅包含了实验步骤的记录也包含了本人对于实验的理解和分析，同时每个实验的结尾都附带了实验的结果依次为：PE信息输出，反汇编代码，Flag成功获取。

1. **实验心得**

这么多实验里，逆向工程实验给人的感觉就是格外的贴近工程实际，据说这和ctf的比赛过程很相似，自己也算是体验了一把跟着汇编代码不断解密的乐趣。

实验主要是关于PE文件的分析和利用IDAPRO完成逆向分析，关于PE部分可能收获并不是特别多，主要是理解老师给出的链接内的源代码，和学习PE文件的格式和作用。

最有趣的部分也就是逆向部分，与其说是学习逆向，不如说是学习IDAPRO的强大功能，真的太过于强大了。而且老师在课上带着我们做过一遍，所以只要认真听课，完成实验还是非常容易的事情。让我印象比较深刻的就是要学会大胆的猜测反汇编后的C代码的作用，不要很谨慎不敢分析，只要你敢猜函数的功能，你才能往下走，走着走着你就会发现猜的都是对的！大不了回溯到猜的位置换别的猜想继续往下走。而且这门课上给我最大的改变就是看程序不用看的百分百懂，你知道个大概不求甚解反而会有奇效。分析的过程就像寻找一张人际关系的大网，对代码和函数采取“社会工程”的手段去寻找联系发现最后想要得到的信息。

真的是十分有意思的实验，选到这门课真的是一点也不后悔！好玩有趣，能学到真正实用的东西，但是感觉有个问题，和老师所说的一样，对IDAPRO过度依赖，整个实验无时无刻不在感叹为什么有这么强大的软件！