逆向分析实验一

[1. 实验内容 1](#_Toc68550069)

[2. 实验准备 1](#_Toc68550070)

[3. 实验过程 1](#_Toc68550071)

[3.1. 代码编写 1](#_Toc68550072)

[3.2. 逆向实例程序 3](#_Toc68550073)

[3.3. “Flag”分析和跟踪 3](#_Toc68550074)

[4. 总结 6](#_Toc68550075)

# 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”。

# 实验准备

1. 深入理解“第2讲 PE和ELF文件格式”中的内容。
2. 下载IDA Pro7.0软件。

# 实验过程

## 代码编写

1. **了解PE文件格式**

在查看老师的PPT以及老师所给的博客后，对于PE文件格式有了更加深刻的理解。

 PE（Portable Execute）文件是Windows下可执行文件的总称，常见的有DLL，EXE，OCX，SYS等，事实上，一个文件是否是PE文件与其扩展名无关，PE文件可以是任何扩展名。

PE 文件格式有一系列的字段，固定在一个已知的位置上，定义文件的其他部分。PE 表头内含信息包括程序代码和资料区域的大小、位置、适用的OS以及堆栈（stack）的最初大小等。PE文件的布局如下图所示：

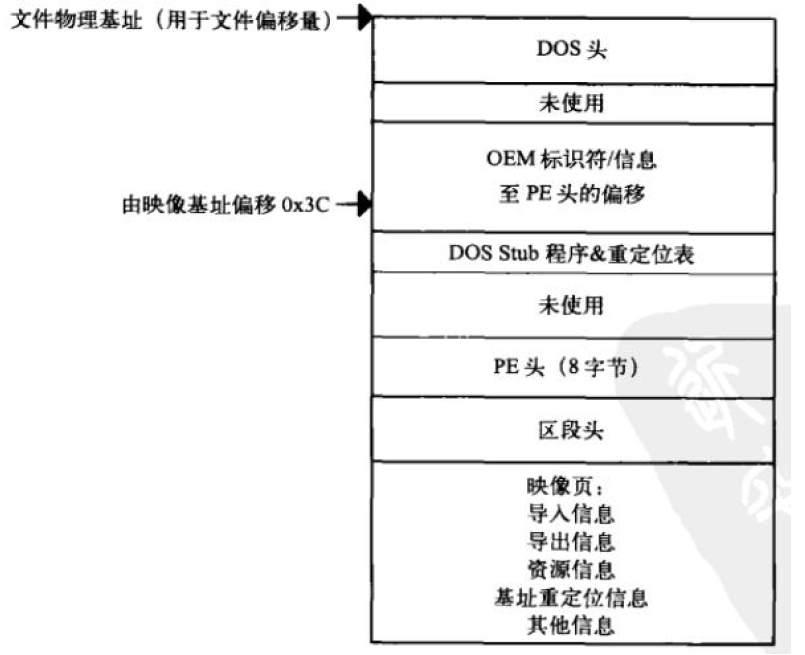


图3.1-1 PE文件布局

在博客中也了解到了PE文件各个结构的代码定义，因此理解另一个博客代码的编写时就有了相应的基础。

1. **代码编写**

参考博客的代码后，理解了代码的实现过程，以下便给出部分代码的说明与解释：

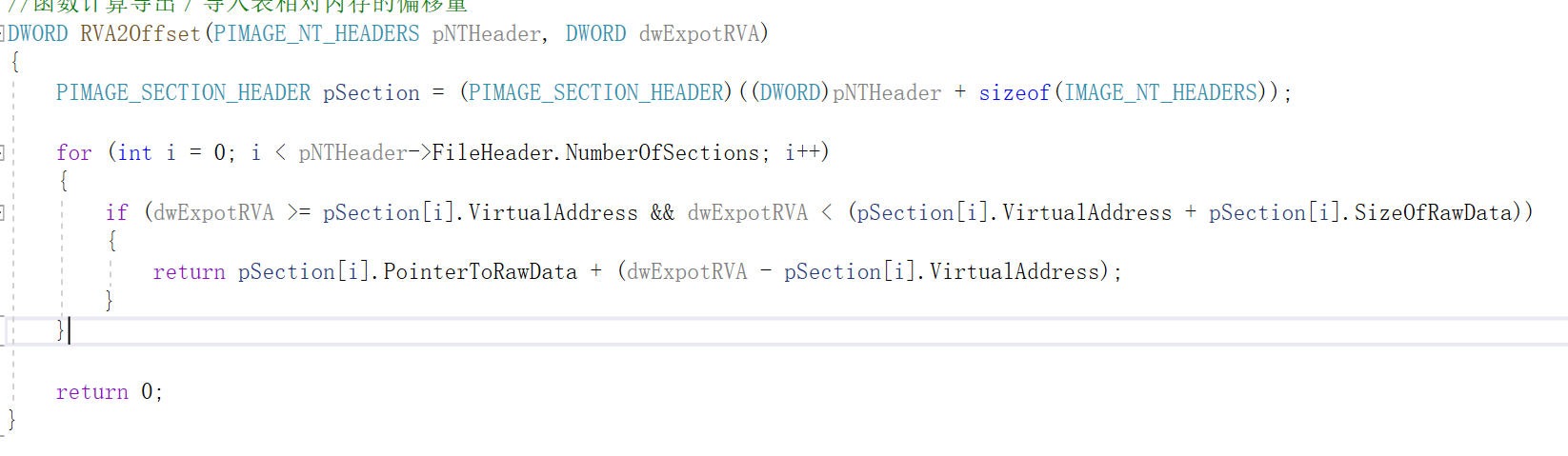


图3.1-2 偏移量计算函数

在代码中需要在导出表和导入表中计算一些地址相对内存的偏移量，函数中所使用的结构体PIMAGE\_NT\_HEADERS是PE相关结构的映像头，其结构如下：

typedef struct \_IMAGE\_NT\_HEADERS {

DWORD Signature;

IMAGE\_FILE\_HEADER FileHeader;

IMAGE\_OPTIONAL\_HEADER32 OptionalHeader;

} IMAGE\_NT\_HEADERS32, \*PIMAGE\_NT\_HEADERS32;

IMAGE\_NT\_HEADERS总的大小为248字节，其中签名占用4字节，文件头占用20字节，可选头占用224字节。其中Signature字段被设置成00004550h ,ASCII码为PE00 ,标志着PE头文件的开始。总的来说IMAGE\_FILE\_HEADER就是记录文件的各种信息的，例如运行平台、文件区块数目、创建日期等等信息。IMAGE\_OPTIONAL\_HEADER结构，他是一个可选结构，是对IMAGE\_FILE\_HEADER的一个补充。

这样就可以理解了上述代码的含义：

在输入PE头地址和所求目标的RVA，在函数中对文件的每个块进行判断，来获取该节的大小，从而获取相对内存的偏移量。如果满足判断的条件，那么就返回对应的大小，否则返回0，即相对偏移量为0。

介于对代码中的结构体逐一讲解需要花费太大篇幅，同时我自己对于其中的结构体也是一知半解，大多通过在网络上的查阅了解的，所以以下便不在讲解具体的代码，而是介绍以下代码的整体流程与实现。

**代码流程：**

在打开目标PE文件后，创建一个该文件的映射内核对象，再将该文件映射对象映射到内存中。通过这三个指针开始实现对PE文件的分析。

首先通过e\_lfanew字段输出文件头地址。而后通过签名是否为0x00004550判断该文件是否是PE文件，0x00004550h对应的ASCII码就似乎PE00 。而后输出PE文件的文件节数和可选映像头长度，输出各个映像头地址和节表地址。随后通过PIMAGE\_SECTION\_HEADER输出各个节表的信息。

后续导出表和导入表相对内存的偏移量输出就是调用编写的RVA2Offset函数实现的。

## 逆向实例程序

1. 借助于IDA逆向实例程序，比较该工具显示的信息与自己代码提取的信息是否一致；

使用代码提取的信息如下图所示：

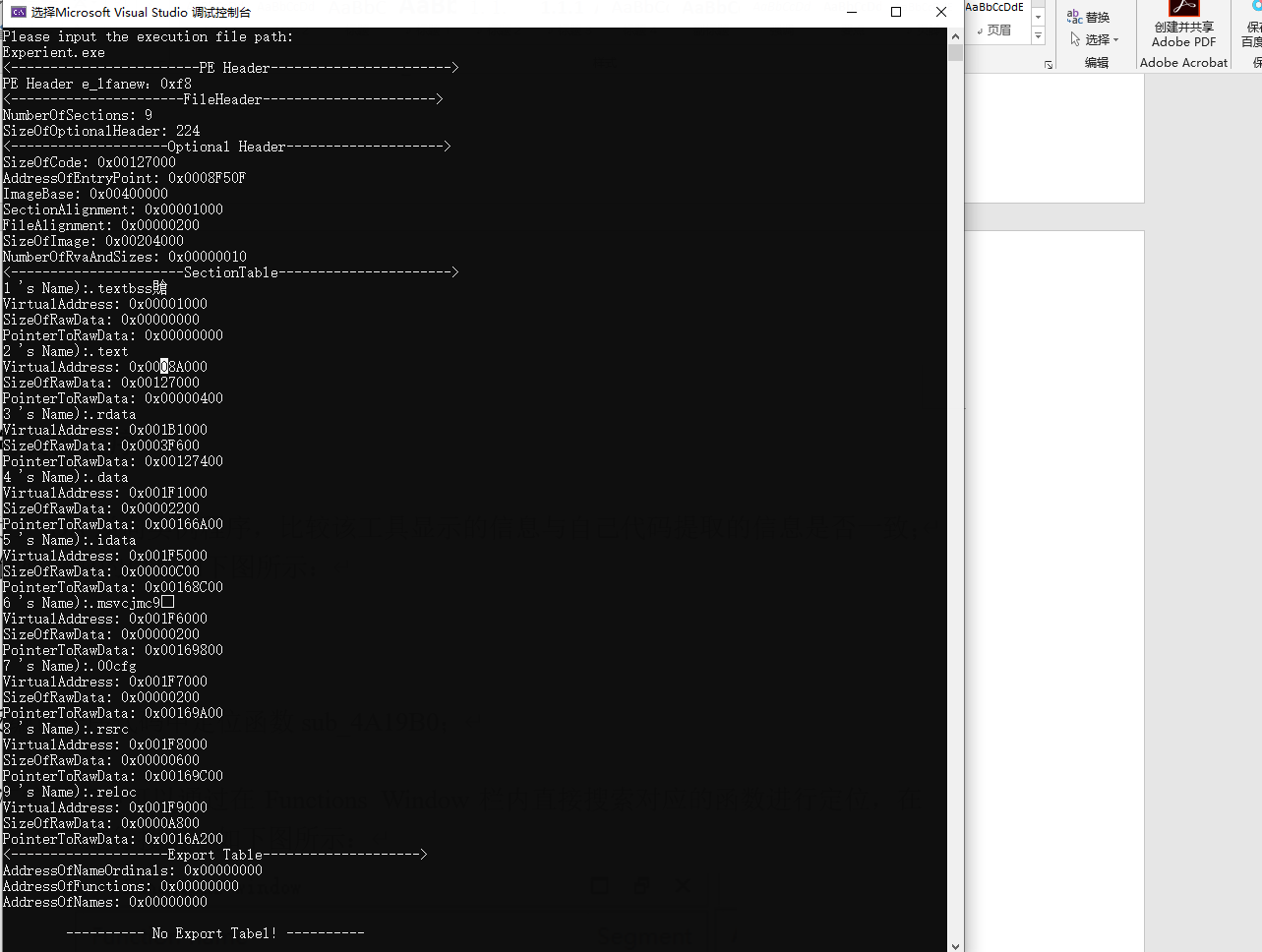


图3.2-1 PE文件信息

介于对于IDA的使用不太熟练不知如何对比两者的信息，在网上也未查到关于使用IDA来查看PE文件头信息的教程，故使用PELord来查看PE文件信息，对应信息如下所示：

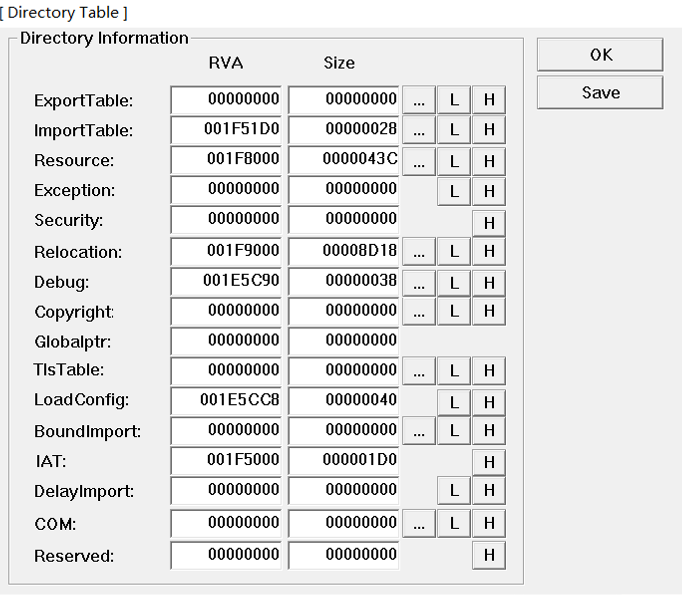


图3.2-2 PE文件信息

通过对比发现代码提取的信息是正确的。

1. 在反编译程序代码中定位函数sub\_4A19B0；

在反汇编程序中可以通过在Functions Window栏内直接搜索对应的函数进行定位，在栏内使用 ctrl+F 进行搜索，如下图所示：



图3.2-3

但在实际应用过程中，一般是不会直接得知运行函数的名字的，所以需要通过PE文件运行的信息来定位，当运行该文件时发现有字符串“Please Input”输出，那么可以通过这个信息来定位运行函数。可以在Strings窗口直接搜索，或者在反汇编代码中搜索也能找到。

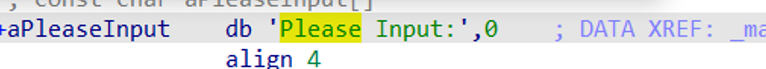
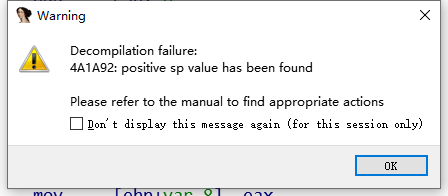


图3.2-4

双击该函数进入相应的反汇编代码，按下F5试图查看该函数的C语言伪代码，但是会出现问题，提示在4A1A92地址处sp栈指针的值未找到，看到这个问题猜测很大可能性就似乎栈的操作出现问题，在call或者jump到某一地址时，sp栈指针所指向的地址出错，详细解释在下一小问中说明。



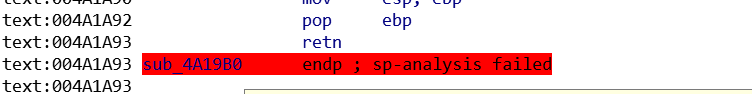


图3.2-5

1. 根据代码中“push”和“pop”指令以及“call”指令的特性，分析产生栈平衡问题的原因。

PUSH 指令首先减少 ESP 的值，再将源操作数复制到堆栈。操作数是 16 位的，则 ESP 减 2，操作数是 32 位的，则 ESP 减 4。

POP 指令首先把 ESP 指向的堆栈元素内容复制到一个 16 位或 32 位目的操作数中，再增加 ESP 的值。如果操作数是 16 位的，ESP 加 2，如果操作数是 32 位的，ESP 加 4。

CALL 指令将就相当于跳转到目标地址前还需要将下一条所需要执行的指令地址压入栈中，如果栈未保证平衡时，执行call指令会跳转到未被期待的地址中去，甚至跳转的地址不被允许访问。

栈平衡问题就是压栈后未进行相应的栈回收操作，实验中出现了两处栈平衡问题，在执行了pop 0后由于未对该操作进行恢复，在后续的指令执行中，又从栈中取出地址跳转，这样就出现问题了。

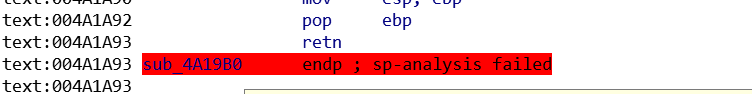


图3.2-6

在4A4A92处执行pop ebp，这是栈中的数据为0，而将该地址赋予给ebp显然是错误的，也因此IDA给出该地址的报错。

1. 借助于指令修改工具，消除栈平衡问题。

找到相应的工作函数sub\_4A19B0后，首先在菜单栏中Options/General/Disassembly/Number of opcode bytes设置IDA使其显示指令的机器码长度为5个字节。

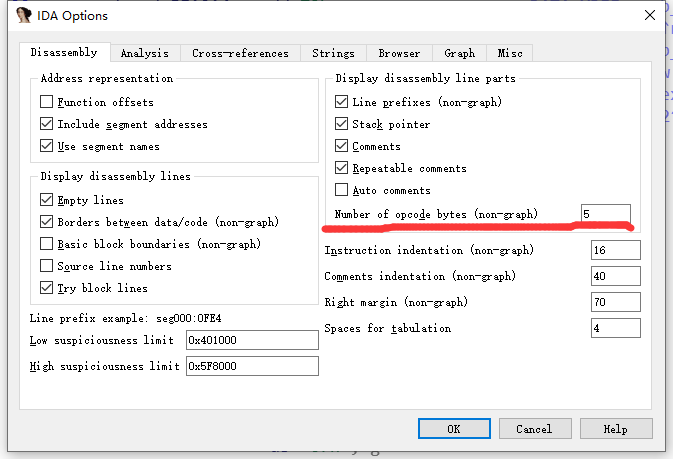


图3.2-7

再使用Edit/Patch program/Change byte功能将push 0指令的机器码改为90 90，即两个nop指令后，就解决了栈平衡的问题。



图3.2-8

## “Flag”分析和跟踪

1. 在上述的工作将函数转换为伪C代码；

找到相应的工作函数，首先在菜单栏中Options/General/Disassembly/Number of opcode bytes设置IDA使其显示指令的机器码长度为5个字节。

再将push 0指令的机器码改为90 90，即两个nop指令后，就解决了栈平衡的问题。而后按下F5即可将该函数转化为伪C语言代码，代码如下所示：

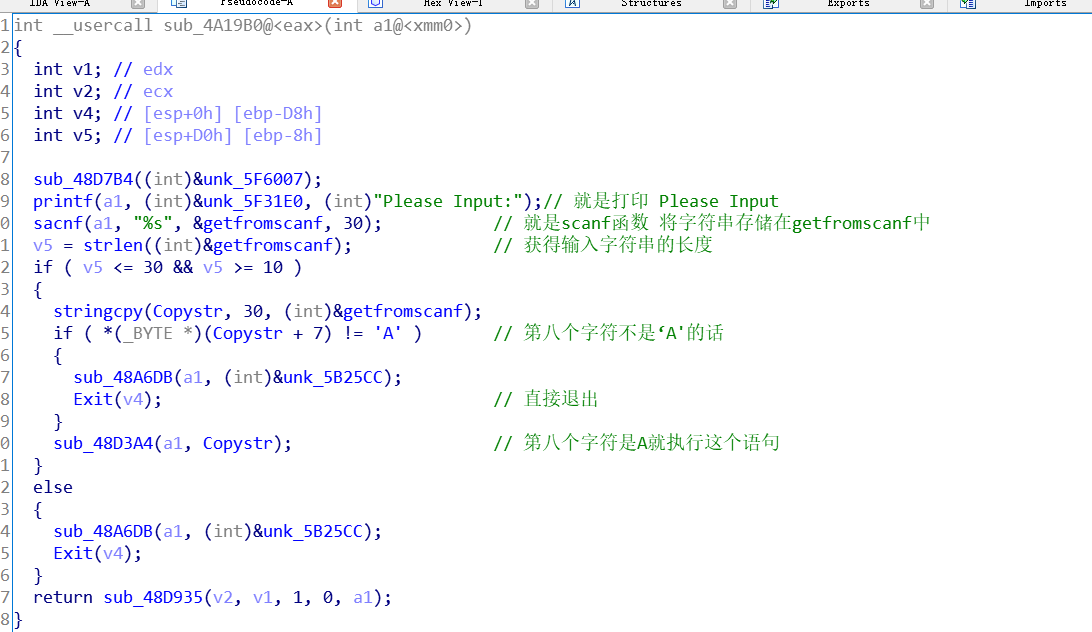


图3.2-9

伪代码中一些函数的解释都在注释中了，同时在函数跟踪过程中将一些函数进行了替换。

1. 以变量“dword\_5F3088”为线索，跟踪“Flag”相关的信息；

在我的伪代码替换中，Copystr变量就是dword\_5F3088，这个变量就是用户输入字符串经过stringcpy函数拷贝后得到的字符串。对伪代码中的函数进行分析：

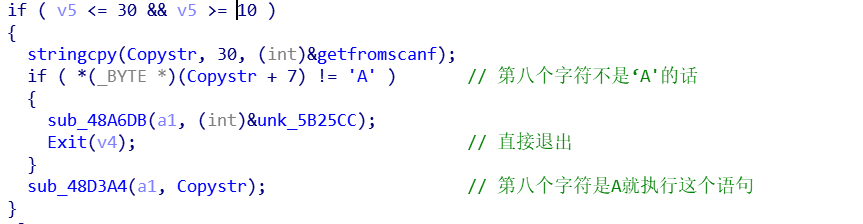


图3.2-10

首先判断输入字符串的长度是否在10~30之间，而后判断输入字符串的第8个字符是否是’A’。如果不是’A’，那么经过两个函数直接退出；否则经过sub\_48D3A4进行处理Copystr，进入sub\_48D3A4函数后，发现这个函数就是对输入的字符串进行加密(简单的异或操作)，

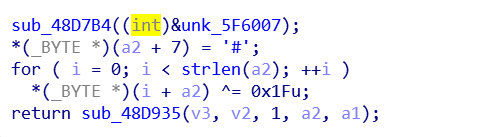


图3.2-11

随后对该变量进行交叉引用查询，发现sub\_49DC80函数中对Copystr的引用与flag相关，于是进入该函数跟踪。

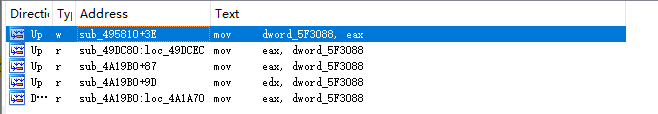


图3.2-12

在该函数中发现了对另外一个变量(字符串)的加密处理，同时将变量和Copystr进行了比较，这里就已经可以明确的猜想这个a2变量就是flag的引用。

而后看到a2和Copystr的比较通过后，会执行sub\_488B4AA函数两次，不过一次参数为111，即ASCII码为‘o’，而另一个参数107即为‘k’ 。这时基本可以确定a2就是flag了，flag在进行和0x1Cu异或后再和加密后的输入字符串比较，如果比较一致那么就输出‘ok’。

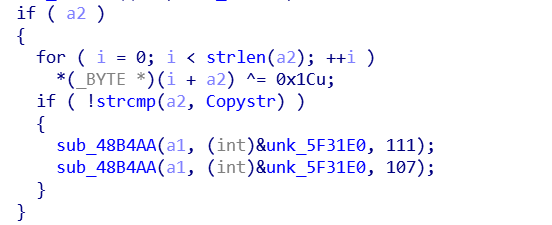


图3.2-13

于是不断地对调用a2变量的函数使用交叉引用功能，最终查找到了flag，flag的初始值就是“invalid argument”。





图3.2-14

1. 尝试恢复“Flag”。

如果仅仅只是简单的输入flag的初始值显然也是不能成功的夺旗，在跟踪过程中我们发现了两个加密（简单的异或操作）。输入字符串在经过加密后与flag加密后的字符串进行对比，同时两者的加密是不同的显然就不能简单的直接使用flag 的初始值来夺旗。

这里就可以编写脚本程序来帮助我们解决问题，直接将flag初始值经过

两次异或，并将第八位‘#‘改成’A‘即可恢复Flag，介于对于C语言更加熟练，所以没有使用IDA的脚本功能，而是编写C程序来解密，对应的代码与结果如下所示：

|  |
| --- |
| #define \_CRT\_SECURE\_NO\_WARNINGS  #include<bits/stdc++.h>  using namespace std;  int main() {  const char\* a = "invalid argument";  int l = strlen(a);  char Flag[100];  memset(Flag, 0, sizeof(Flag));  for (int i = 0; i < l; i++) {  Flag[i] = (\*(a + i)) ^ 0x1Cu;  cout << Flag[i];  }  cout << endl;  for (int i = 0; i < l; i++) {  Flag[i] = Flag[i] ^ 0x1Fu;  cout << Flag[i];  }  cout << endl;  Flag[7] = 'A';  cout << Flag << endl;  return 0;  } |

解密结果如下图所示：



图3.2-15

测试结果如下图所示：



图3.2-16

# 总结

本次的实验我觉得非常有趣！这也是第一次接触CTF相关的东西吧，以前都只是看别人去比赛做题，第一次做逆向是真的觉得很有意思。

实验刚开始做的时候，很认真地去跟踪每一个函数，但这样却陷入函数的调用深渊中，有的函数就是不断的调用一些系统函数，但实际上没有太大的实质性意义，对于解析Flag没有太大帮助。在完成这次实验后倒是总结了一些经验，不要深于某一个函数或者某一块难以理解的代码，不要去转牛角尖。要会利用工具的各项功能，一开始没有使用交叉引用，就不停的找源flag字符串，甚至于在Strings窗口中一个个的比对。而后发现这个PE必须需要将输入的字符串（可能经过某种处理）与Flag进行比对才能判断是否正确，所以必须会有对输入字符串变量的调用。使用交叉引用后，发现了一个函数有进行字符串比较操作，同时其中一个字符串变量地址就是输入的字符串地址。于是对这个a2形参变量的调用函数进行交叉引用，然后找到了这个flag的实际地址，就是“Invalid argument”。最后通过编写解密程序完成了实验。

本次实验收获了许多，让我对PE文件的结构有了一定的了解，学会了逆向工具IDA Pro的一些使用方法，以及如何简单地利用工具去进行逆向分析。这次实验让我对逆向分析有了更深的认识以及更浓厚的兴趣，知道了一些基础的逆向分析程序的思路，希望以后能熟练使用这门课学到的知识和技能去做一些更具有挑战性的分析。