流程：

0.MFC人机交互

1.加载PE文件，读取关键信息到CPE对象

2.导入表信息以自定义结构体保存到CPE对象

3.dll载入内存并把pe关键信息写入dll结构体

4.pe重定位指向dll（记得恢复loadlibrary的重定位）

5.合并PE与dll区段（记得包含自定义导入表结构的大小）

6.修改OEP为dll的start函数

7.释放资源

关键细节：

1.自定义结构体保存导入表信息

经由文件头找到导入表结构体指针(PIMAGE\_IMPORT\_DESCRIPTOR)(20)

经由导入表结构体指针找到函数信息(4)，有序号和函数名两种形式

typedef struct \_MYIMPORT

{

DWORD m\_dwIATAddr;

DWORD m\_dwModNameRVA;

DWORD m\_dwFunNameRVA;

BOOL m\_bIsOrdinal;

DWORD m\_Ordinal;

}MYIMPORT, \*PMYIMPORT;

注意：PE文件中偏移信息是文件偏移，PE加载器正是依靠这些文件偏移信息把PE加载到内存的。

//#《1》当 IMAGE\_THUNK\_DATA 值的最高位为 1时，表示函数以序号方式输入，这时候低 31位被看作一个函数序号。当 IMAGE\_THUNK\_DATA 值的最高位为 0时，表示函数以字符串类型的函数名方式输入，这时双字的值是一个 RVA，指向一IMAGE\_IMPORT\_BY\_NAME 结构。

IMAGE\_IMPORT\_BY\_NAME STRUCT

Hint WORD ?

Name1 BYTE ?

IMAGE\_IMPORT\_BY\_NAME ENDS

可由如下取字符串：

PIMAGE\_IMPORT\_BY\_NAME pstcFunNam->Name

//#《2》记录自己对应模块的IAT地址m\_dwIATAddr，记录函数序号或者函数字符串偏移，注意在m\_pFunNameBuf函数名数组保存了函数名字符串

//#《3》把m\_pMyImport数组倒序保存，因为模块IAT地址已经保存了，所以只是在重定位的时候是由后向前进行填充的

//#《4》记录模块大小和函数大小，均是字符串形式。

m\_dwSizeOfModBuf = dwSizeOfModBuf;

m\_dwSizeOfFunBuf = dwSizeOfFunBuf;

2.解密IAT

//#《1》把函数地址放入g\_pfnVirtualAlloc new的内存，然后记录堆空间1地址

然后把堆空间1地址放入g\_pfnVirtualAlloc new的堆空间2，然后把堆空间2的地

址放入到IAT地址。 堆空间2的地址写入IAT地址可避免IAT地址和函数地址同时出

现，其实这里是邻近出现。

//#函数中的局部信息可以放到申请的内存中，并把内存地址赋值给全局变量进行访问，因为局部申请的内存还是存在的，只是地址找不到。因为C/C++是自己释放的，而Java和C#是自动回收的。

//《2》这里只是局部堆空间1复制到申请的堆空间2，又把堆空间2（其实和堆空间一模一样）地址写入到IAT地址。基本还是一次。

第一遍：

0 3 6 9 12 15

第二遍：

0 7 14 21

第三遍：

0 13 26 39

破解分析：

1. 找到IAT地址004020B0，由此可得IAT地址为00402000

ecx以指针数组形式索引形式往IAT地址写值

mov ecx,[eax+esi\*4]

且eax值是mov eax,[004169C0]，指针形式给出

1. 指针数组指针（4169C0）下断，看谁修改了该指针
2. 另辟蹊径，直接由明文知道min函数的地址。

针对性的查看IAT地址上min函数怎么恢复的，因为dll较多，而dlltest熟悉。

min函数IAT地址是4020a8。

这次把messagebox去掉，避免加入不必要的dll麻烦；

因为多个dll，不清楚什么时候开始往中间数组填充dlltest的函数地址；

注意：在OD中尽量使用地址进行下断，而不是某寄存器值作为指针被修改否下断，因<1>为存在不同运行程序时，某些内存指针变化（只要大小满足即可）。

为此可提出插入变化混淆代码来对抗基于地址的下断点。

<2>虽然数组的指针信息固定，可由地址进行下断；但是索引信息是固定的，可以借助基于地址下断后加上固定索引进而到达目标地址。（思考对抗方式，索引随机化思路）。

3》一种情况下得到指针数组指针为2400，已知该加密中数组索引没改变(28)，所以min1地址先存到了2400+28\*4=2400a0上。该地址下修改断。