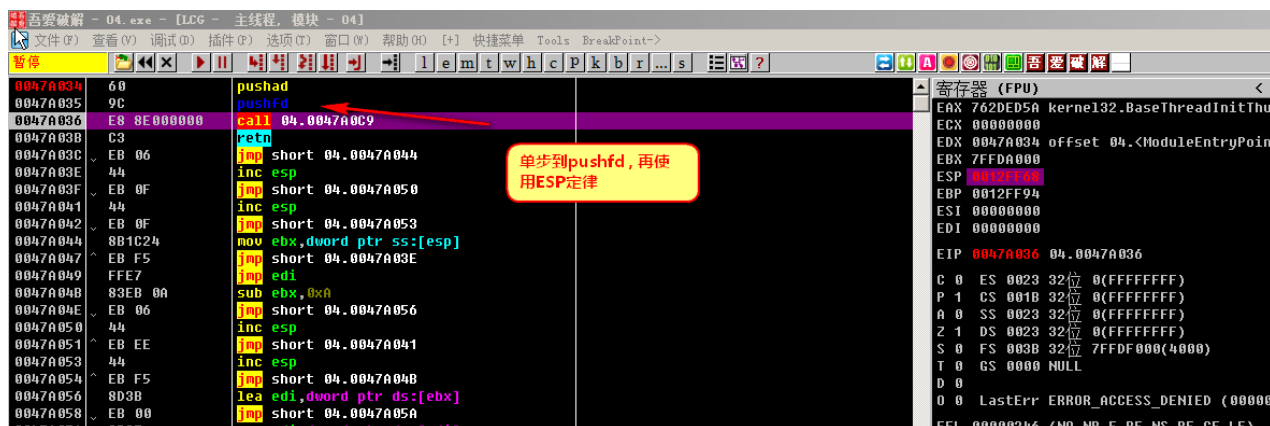
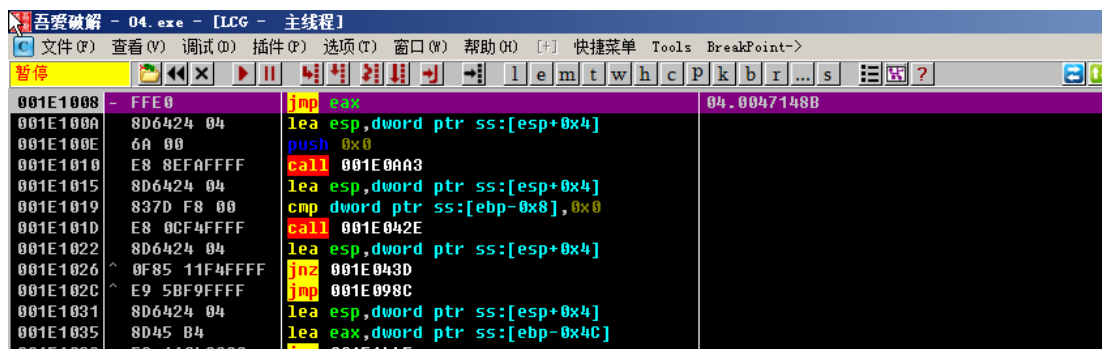


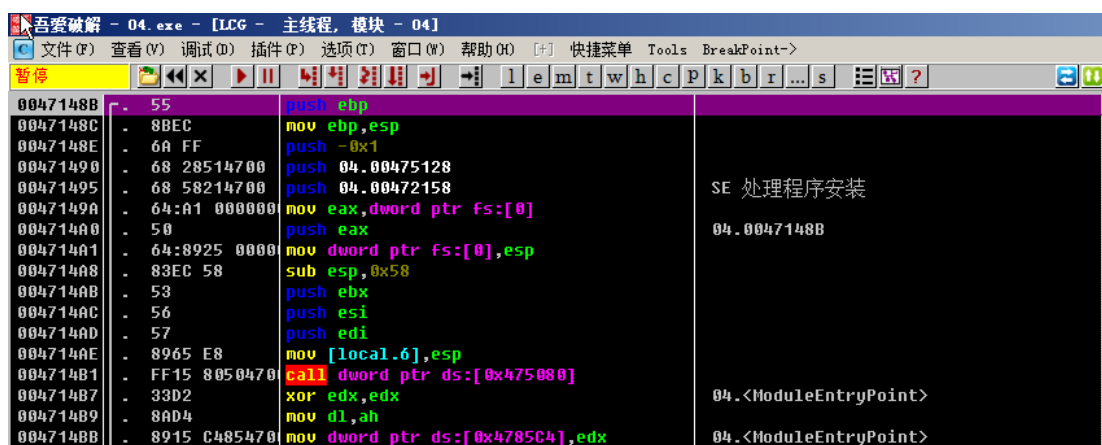
# 第一步,寻找 OEP



ESP 硬件断点命中后,单步几次,就到了 OEP

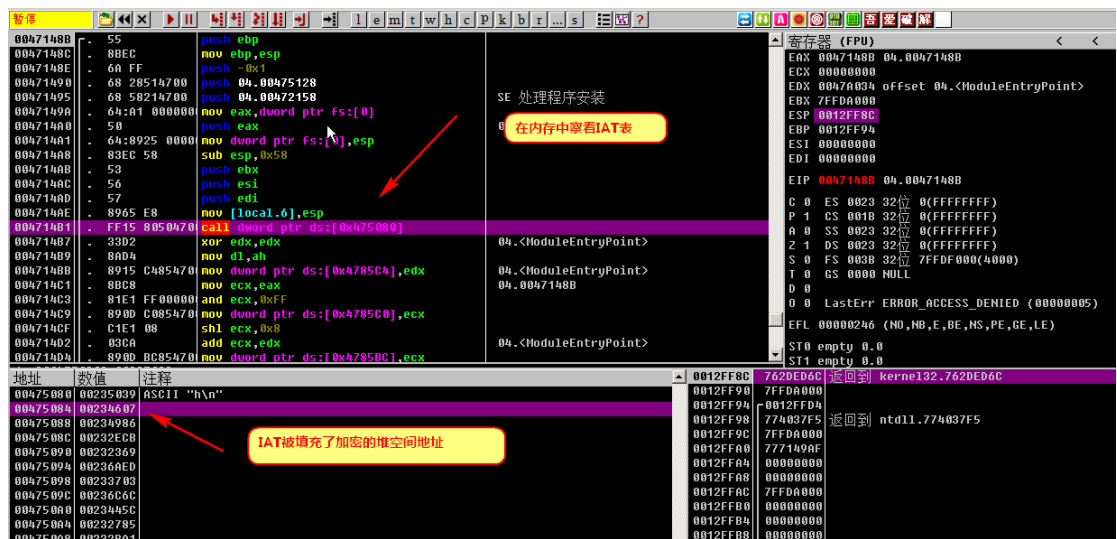


**0047148B** 地址处,就是 OEP



通过查看 OEP 处第一个 CALL 函数 `call dword ptr ds:[0x475080]`, 一般程序中引用的函数地址保存在 IAT 表中, 而这里的汇编指令是从 0x475080, 取 4 字节内容, 再 call 的。

说明 0x475080 地址就是 IAT 表的地址, 它里面保存的是函数地址。



在内存中查看 0x475080 IAT 表中的数据，发现被填充的全是 23xxxx 开头的堆内存地址，而不是正常函数开头的地址，比如 7xxxxxxx。所以 IAT 被加密了

## 第二步,找到填充 IAT 的位置

在 IAT 表的位置，下硬件写入断点，第一次命中的是在 ntdll 模块，直接 F9，等第二次命中，来到 001E0895 就是在填充 IAT，我们的思路是，再它填充完以后，再次用原始函数地址覆盖 IAT，等于它没做加密。

所以我们需要记录，填充 IAT 指令的下一行指令的地址 001E0897，但是这个地址，是在申请的堆空间里面的一个地址，所以我们只需要记录偏移地址 0897 就行了，而 0x1E0000 是每次运行程序都会动态申请得到的一个地址，我们写脚本的时候需要动态获取这个地址，再加 0897 这个偏移，就可以每次都能准确获取到填充 IAT 后面的位置。

```
001E0895      8902      mov dword ptr ds:[edx],eax
001E0897      E8 39000000 call 001E08D5
```

Debugger window showing assembly code and registers.

**Assembly Code:**

```

001E0888 E8 04FCFFFF call 001E0561
001E088D 8B45 DC mov eax,dword ptr ss:[ebp-0x24]
001E0890 E9 07FCFFFF jmp 001E056C
001E0895 8902 mov dword ptr ds:[edx],eax
001E0897 E8 39000000 call 001E0805
001E089C 8B4E 08 mov ecx,dword ptr ds:[esi+0x8]
001E089F E8 06FBFFFF call 001E045A
001E08A4 83D7 F4 00 cmp dword ptr ss:[ebp-0xC],0x0
001E08A8 E9 97DFFFFF jmp 001E0644
001E08AD C745 AC 000000 mov dword ptr ss:[ebp-0x54],0x0
001E08B4 E9 59FBFFFF jmp 001E0412
001E08B9 58 pop eax
001E08BF E9 37000000 jmp 001E10F6
001E08C3 8B45 04 lea esp,dword ptr ss:[esp+0x4]
001E08C9 E8 E3030000 call 001E135F
001E08CE 6A 00 push 0x0
001E08D0 E8 5C070000 call 001E1031
001E08D5 8B45 04 lea esp,dword ptr ss:[esp+0x4]
001E08D9 58 pop eax
001E08DA E8 86010000 call 001E0A65
001E08DF 8B45 04 lea esp,dword ptr ss:[esp+0x4]
001E08E3 C2 0400 ret 0x4

```

**Registers (FPU):**

```

EAX 00235039 ASCII "h\n"
ECX 00235039 ASCII "h\n"
EDX 00000000 ASCII "9P#"
EBX FFB0F200
ESP 0012FE08 UNICODE "("
EBP 0012FF50
ESI 0039001C
EDI 00390024
EIP 001E0897

```

**Memory Dump:**

地址	数值	注释
00475080	00235039	ASCII "h\n"
00475084	00234607	
00475088	00234986	
0047508C	00232ECB	
00475090	00232369	
00475094	00236AED	
00475098	00233703	
0047509C	00236C6C	
004750A0	0023445C	
004750A4	00232785	

Debugger window showing assembly code and registers.

**Assembly Code:**

```

00475080 00235039 ASCII "h\n"
00475084 00234607
00475088 00234986
0047508C 00232ECB
00475090 00232369
00475094 00236AED
00475098 00233703
0047509C 00236C6C
004750A0 0023445C
004750A4 00232785

```

**Registers (FPU):**

```

EAX 00235039 ASCII "h\n"
ECX 00235039 ASCII "h\n"
EDX 00000000 ASCII "9P#"
EBX FFB0F200
ESP 0012FE08 UNICODE "("
EBP 0012FF50
ESI 0039001C
EDI 00390024
EIP 001E0897

```

**Memory Dump:**

地址	数值	注释
00475080	00235039	ASCII "h\n"
00475084	00234607	
00475088	00234986	
0047508C	00232ECB	
00475090	00232369	
00475094	00236AED	
00475098	00233703	
0047509C	00236C6C	
004750A0	0023445C	
004750A4	00232785	

**Context Menu:**

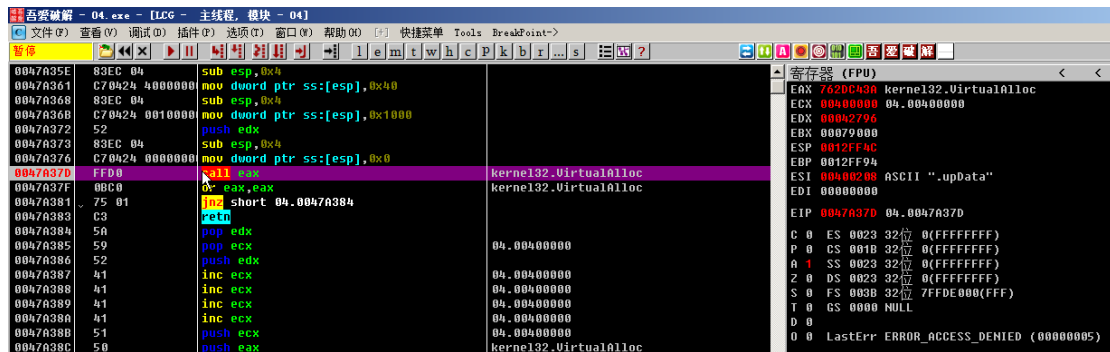
- 断点 (B)
- 查找 (S)
- 数据窗口中跟随
- 查找参考 (R) Ctrl+R
- 查看可执行文件 (E)
- 复制到可执行文件
- 转到
- Hex
- 文本
- 短型
- 长型
- 浮点
- 反汇编
- 指定
- CheckVmp
- 字符串
- 界面选项

**Hardware Write Menu:**

- Byte
- Word
- Dword

## 0x1E0000 怎么来的？

在入口单步跟，可以找到初始化壳代码相关的代码，最后由 VirtualAlloc 申请了一块堆空间，堆空间地址就是 0x1E0000，因为是动态申请的，所以可能每次运行程序，返回的申请的堆空间地址都是不同的，所以需要在申请完下一行指令的位置（0x47A37F）获取 EAX 的返回值。



```
0047A35E 83EC 04      sub esp,0x4
0047A361 C70424 40000000 mov dword ptr ss:[esp],0x40
0047A368 83EC 04      sub esp,0x4
0047A36B C70424 00100000 mov dword ptr ss:[esp],0x1000
0047A372 52          push edx
0047A375 83EC 04      sub esp,0x4
0047A378 C70424 00000000 mov dword ptr ss:[esp],0x0
0047A37D FFDB        call eax, kernel32.VirtualAlloc
0047A37F 00          or eax,eax
0047A381 75 01       jnz short 0x.0047A384
0047A383 C3          retn
0047A384 5A          pop edx
0047A385 59          pop ecx
0047A386 52          push edx
0047A387 41          inc ecx
0047A388 41          inc ecx
0047A389 41          inc ecx
0047A38A 41          inc ecx
0047A38B 51          push ecx
0047A38C 50          push eax, kernel32.VirtualAlloc
```

所以我们需要 在申请完以后，获取 EAX 中返回的堆空间地址 。

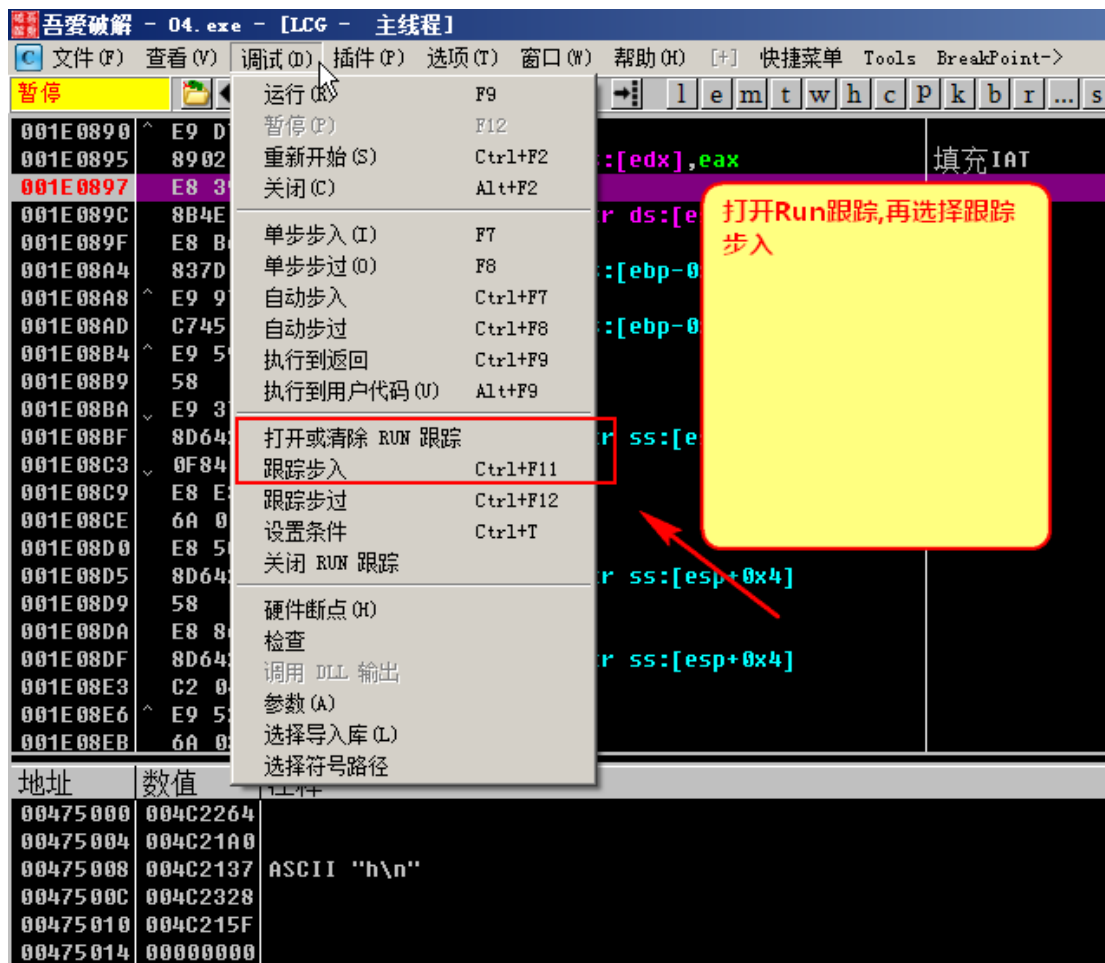
为啥要获取这个地址，因为壳真正的代码（填充 IAT、获取函数地址）就是在申请的空间执行的

## 第三步，找到原始函数地址

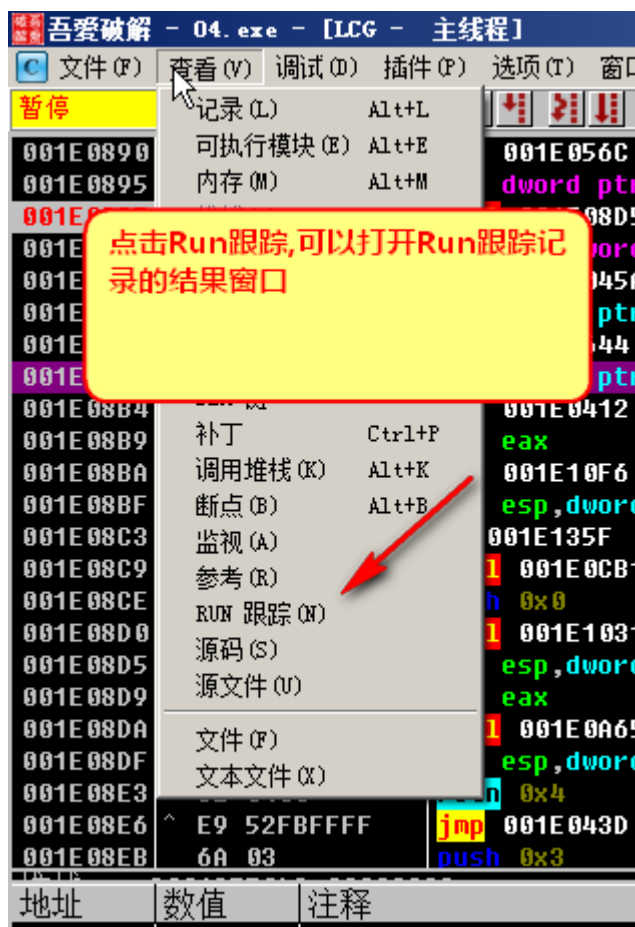
我们找到了 填充加密 IAT 的代码位置，说明获取原始函数 API 地址的代码，就在之前执行过的代码不远处了。

我们需要使用 Run 跟踪来帮助我们找到，获取原始函数地址的位置。

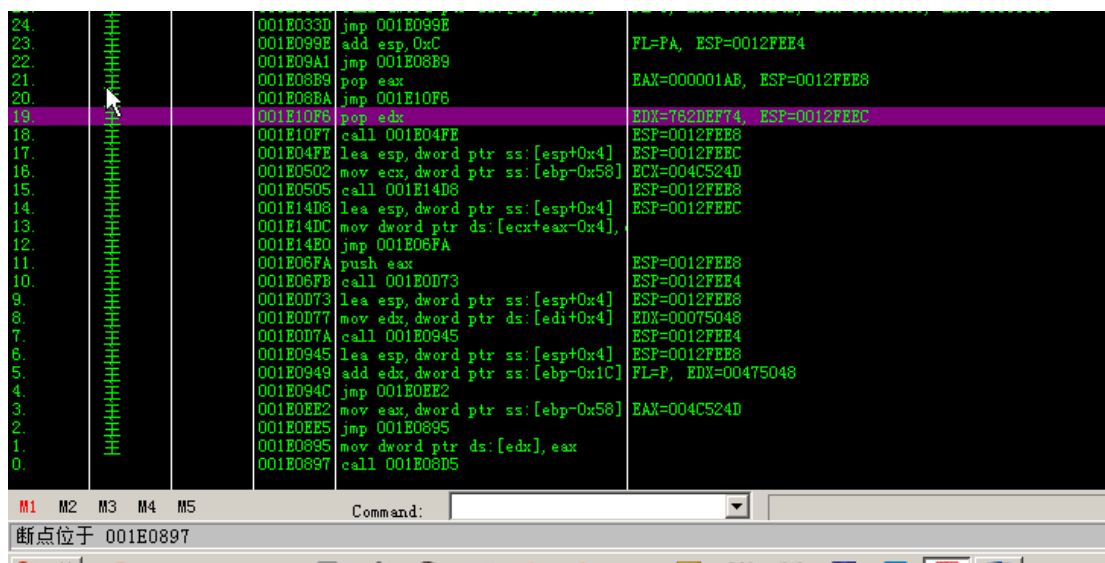
首先，我们需要用之前硬件断点的方法断到填充 IAT 的位置，再在这个位置下软件断点，然后 删 除 硬 件 断 点 。



打开 Run 跟踪窗口



跟踪一次循环代码后,察看 run 跟踪记录的指令信息,重点关注这次循环的过程中,寄存器发生的改变,从最后一条记录往前找,找寄存器中的值,像真实函数地址的指令,下面图中选中的一条指令中 EDX 的值,就很像一个函数地址



但是我们需要确认一下,所以到该指令的位置下断点确认,edx 果真就是个函数地址,

我们需要在获取 函数地址指令的下一行，获取 EDX 的值，所以需要记录偏移 **10F7**

```
001E10C4 E8 41FFFFFF call 001E100A
001E10C9 FF75 D8 push dword ptr ss:[ebp-0x28]
001E10CC E8 AE000000 call 001E117F
001E10D1 8D6424 04 lea esp,dword ptr ss:[esp+0x4]
001E10D5 57 push edi
001E10D6 E8 66FFFFFF call 001E0FA1
001E10DB 83C4 08 add esp,0x8
001E10DE E9 99F5FFFF jmp 001E067C
001E10E3 33C9 xor ecx,ecx
001E10E5 E9 31010000 jmp 001E121B
001E10EA 8D6424 04 lea esp,dword ptr ss:[esp+0x4]
001E10EE 8955 E0 mov dword ptr ss:[ebp-0x20],edx
001E10F1 E9 95F9FFFF jmp 001E0A8B
001E10F6 5A pop edx
001E10F7 E8 02FAFFFF call 001E04FE
001E10FC 8D6424 04 lea esp,dword ptr ss:[esp+0x4]
001E1100 8975 F0 mov dword ptr ss:[ebp-0x10],esi
001E1103 E8 C2FCFFFF call 001E0DCA
001E1108 8D6424 04 lea esp,dword ptr ss:[esp+0x4]
001E110C 8B7E 18 mov edi,dword ptr ds:[esi+0x18]
001E110F E8 9BF9FFFF call 001E0AAF
001E1114 68 00400000 push 0x40000
001E1119 E9 ED020000 jmp 001E140B
```

寄存器 (FPU)

```
EAX 0000041C
ECX 00000000
EDX 762D2B80 kernel32.HeapDestroy
EBX FFDF2C0
ESP 0012FEEC
EBP 0012FF58
ESI 004E001C
EDI 004E0474 UNICODE "嫉"
EIP 001E10F7
C 0 ES 0023 32位 0(FFFFFFFF)
P 1 CS 001B 32位 0(FFFFFFFF)
A 1 SS 0023 32位 0(FFFFFFFF)
Z 0 DS 0023 32位 0(FFFFFFFF)
S 0 FS 003B 32位 7FFDE000(FFF)
T 0 GS 0000 NULL
O 0 LastErr ERROR_ACCESS_DENIED (00000000)
EFL 00000216 (NO,NB,NE,A,NS,PE,GE,G)
ST0 empty 0.0
ST1 empty 0.0
```

地址 数值 注释

```
00474FF0 00000000
0012FEEC 00000000
0012FEF0 0000000F
```

## 第四步，编写 OD 脚本

通过前面几步的分析，我们所获取的数据如下：

**0047148B** == OEP

**0x47A37F** == 申请后的堆空间，里面是壳代码，返回值 EAX

**0x0897** == 填充 IAT 的下一行指令位置，我们需要用真实地址覆盖填充过的加密 IAT

**0x10F7** == 获取真实函数地址的下一行指令位置，真实地址保存在 EDX 中