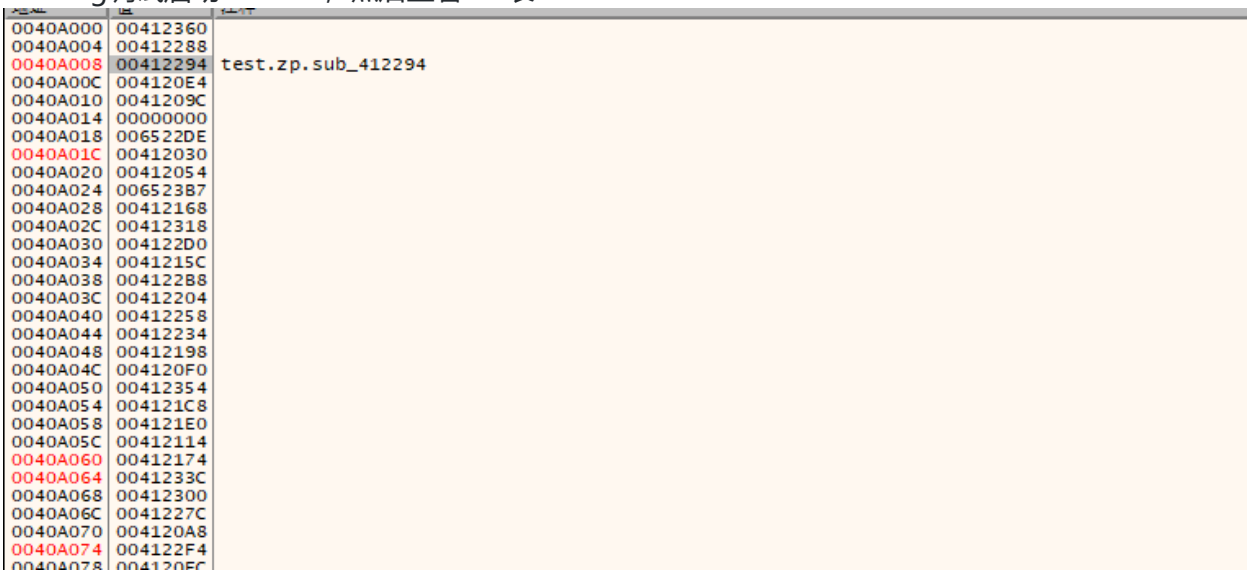
zp1.6脱壳手记

zp壳百度网盘链接，密码：脱壳手记

保护手法说明

IAT保护

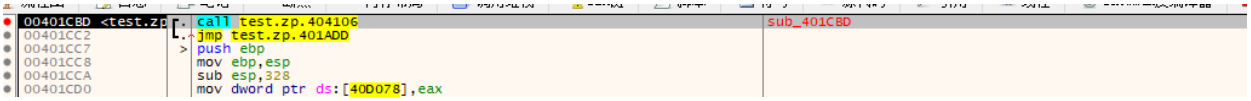
X64dbg调试启动test.exe，然后查看IAT表



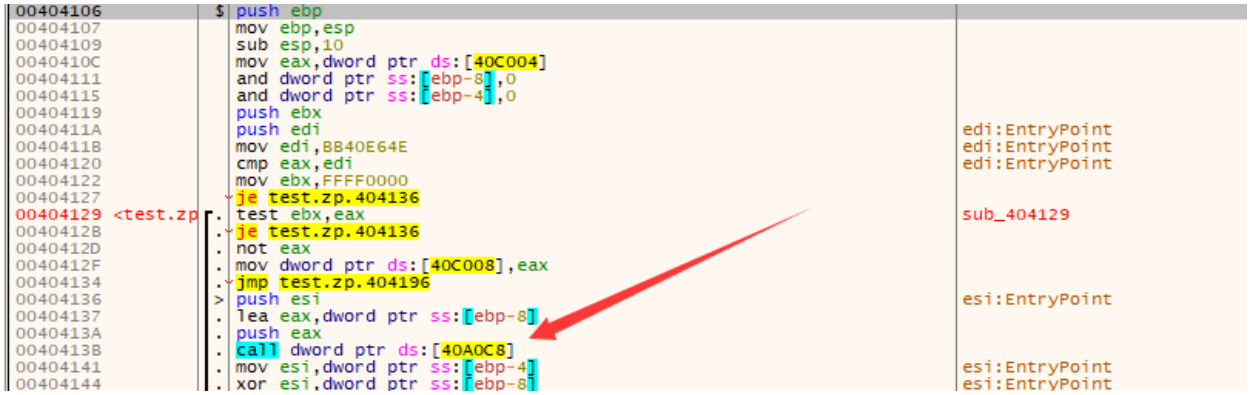
不出意外，这里被改过了。IAT的保护手法有三种，第一种是IAT混淆，第二种是API模拟，第三种是把API代码抽出来，第三种方式其实还有待商榷。

IAT混淆

定位到入口点



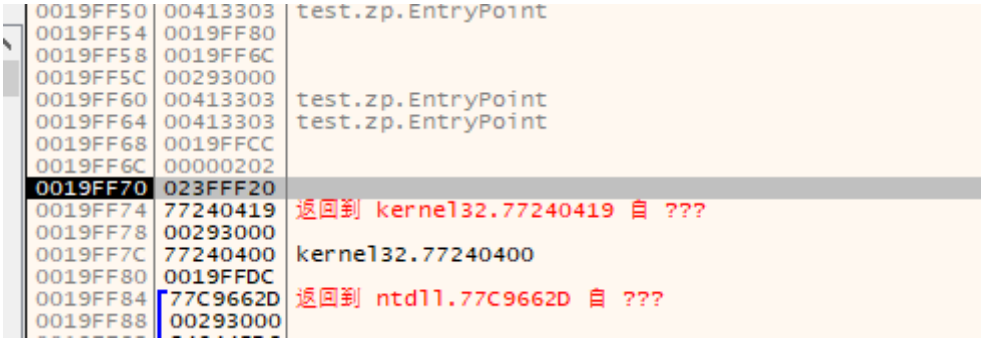
进到第一个call里面，跟进第一个调用API的地方，



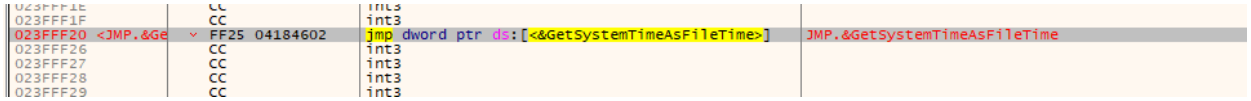
这里，可以看到IAT被混淆过了。将这里的jmp的地址设为新eip，然后单步步过跟踪，直到ret的位置

```
004121A4   push 573555DA
004121A9   jmp test.zp.41200C
0041200C   jmp 21AB0B8
021AB0B8   pushfd
021AB0B9   pushad
021AB0BA   push dword ptr ss:[esp+24]
021AB0BE   call 21A9CC8
021AB0C3   popad
021AB0C4   popfd
021AB0C5   ret
```

在这里，取出栈顶的值0x23fff20



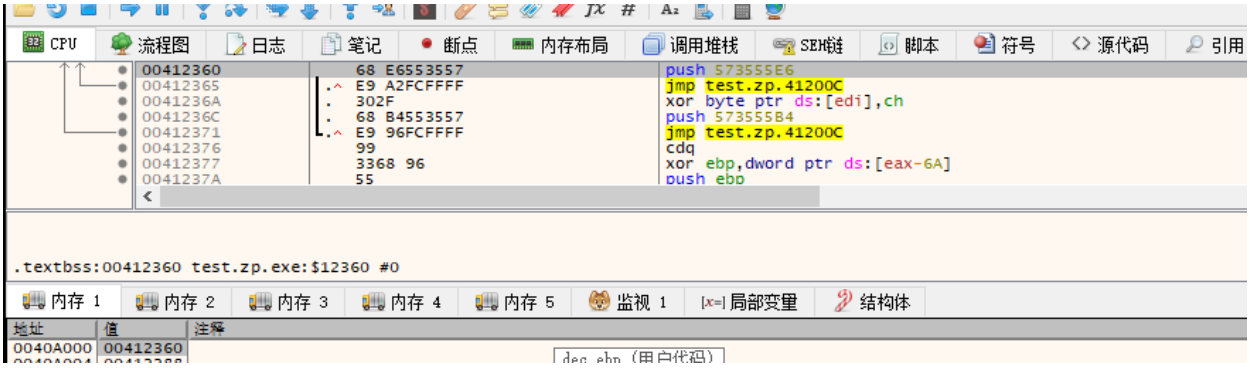
然后在反汇编窗口中转到这个值，



这里可以看到，通过一个jmp跳转到了API的地址。
到此，第一种IAT的混淆方式就分析完了。

API模拟

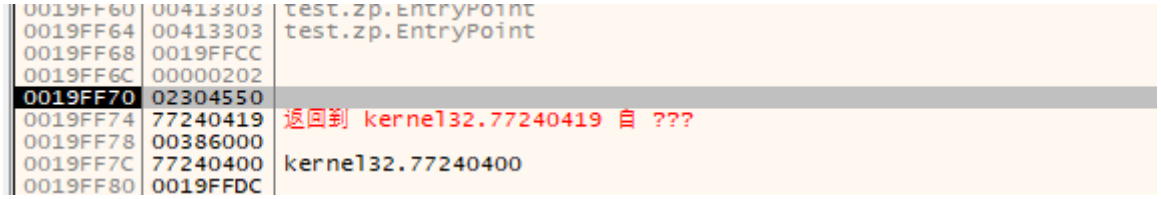
在IAT的混淆中，有一部分ret的地址并非是API的地址，而是一个堆地址，这是壳自己把系统库dll(kernel32, user32, gdi32)载入到内存中，然后这个地址就是壳载入的dll在内存中的地址。这种手法称之为api模拟。
定位IAT表，在反汇编窗口中转到IAT表的第一项，



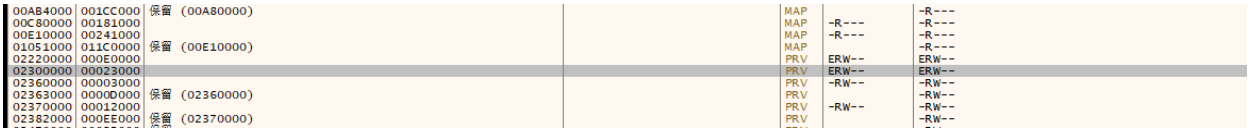
这里明显看出来IAT所对应的位置不是API的地址，而是一个jmp，将这里设为eip，然后单步步过跟踪，

```
00412360    push 573555E6
00412365    jmp test.zp.41200C
0041200C    jmp 62B0B8
0062B0B8    pushfd
0062B0B9    pushad
0062B0BA    push dword ptr ss:[esp+24]
0062B0BE    call 629CC8
0062B0C3    popad
0062B0C4    popfd
0062B0C5    ret
```

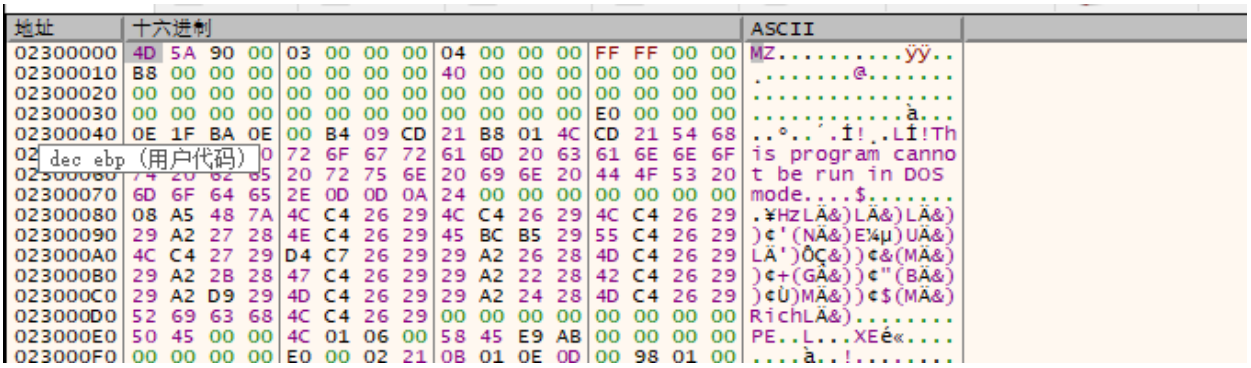
这个时候看堆栈的栈顶，地址为0x02304550



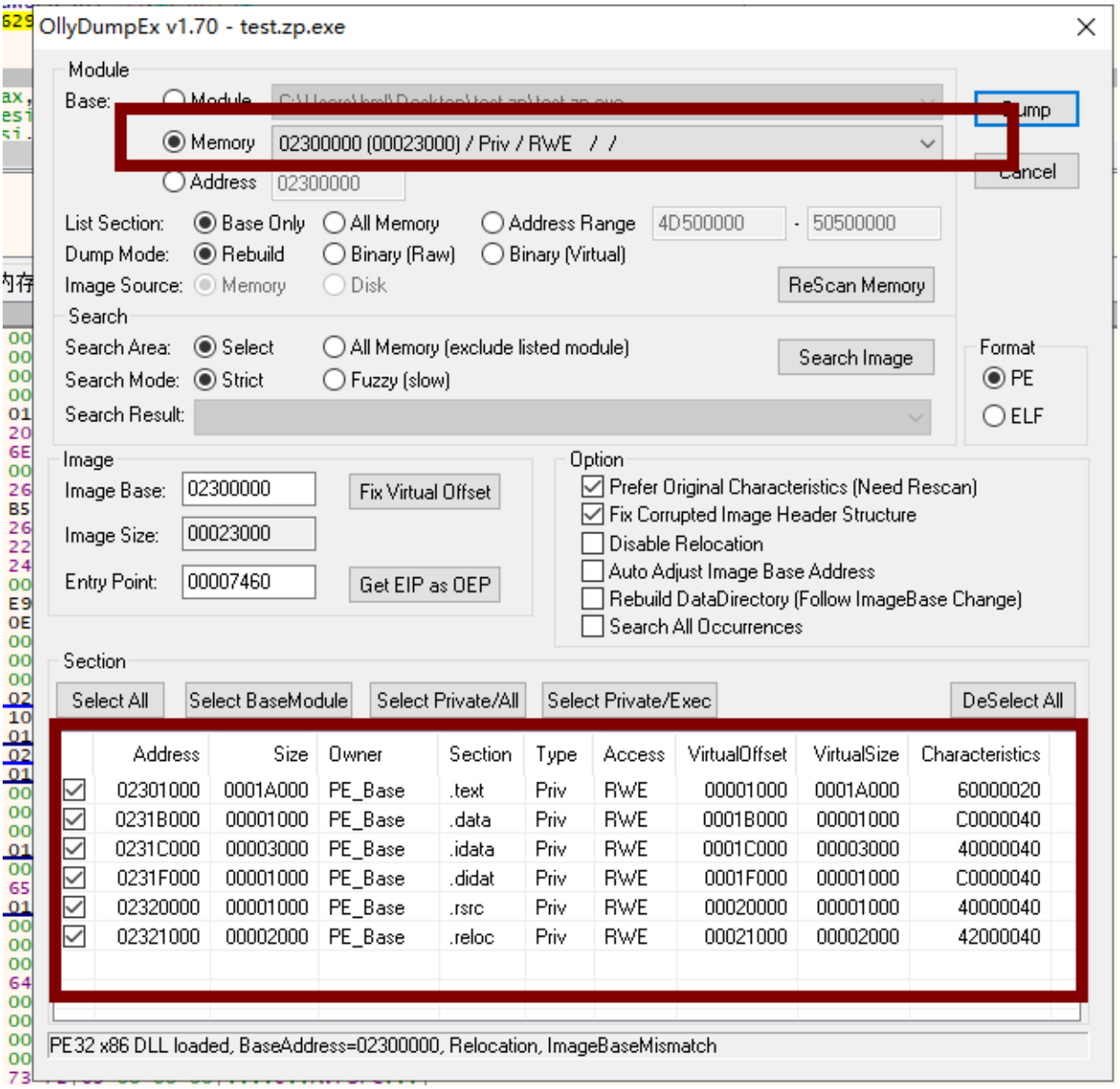
在栈上这个地址右键转到内存布局，然后在内存布局中查看地址所在的内存范围，



然后在0x02300000上右键，在内存窗口中转到，查看内存中首地址处的内容：



这里明显看出是个PE文件，然后使用x64dbg的插件OllyDumpex，选择memory，找到这个地址范围，下面会识别出这个dll的区块信息。

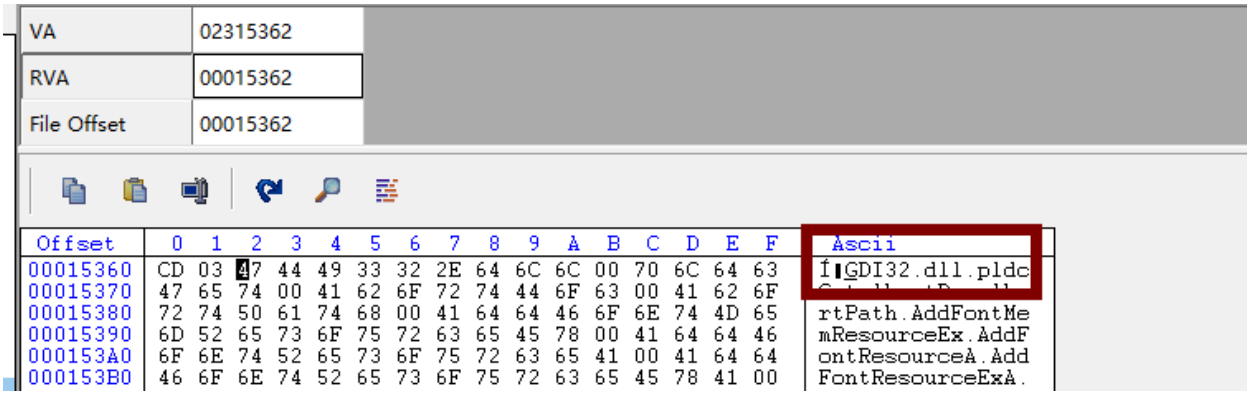


然后点击dump按钮，保存文件为somesys.dll，接着使用CFF打开这个dll

Member	Offset	Size	Value
Characteristics	00012D10	Dword	00000000
TimeDateStamp	00012D14	Dword	ABE94558
MajorVersion	00012D18	Word	0000
MinorVersion	00012D1A	Word	0000
Name	00012D1C	Dword	00015362
Base	00012D20	Dword	000003E8
NumberOfFunctions	00012D24	Dword	000003E9
NumberOfNames	00012D28	Dword	000003C1

Ordinal	Function RVA	Name Ordinal	Name RVA	Name
(nFunctions)	Dword	Word	Dword	szAnsi
0000046C	00008B90	0084	00015C87	D3DKMTCreateKeyedMutex2
0000046D	00008BA0	0085	00015C9F	D3DKMTCreateKeyedMutex
0000046E	00008BB0	0086	00015CB6	D3DKMTCreateOutputDupl

这里可以看出，dll的导出表很全，感谢作者没有把这个dll的导入表给废掉啊，省了很多事，直接拿导出表的name字段，去看看dll的名字，

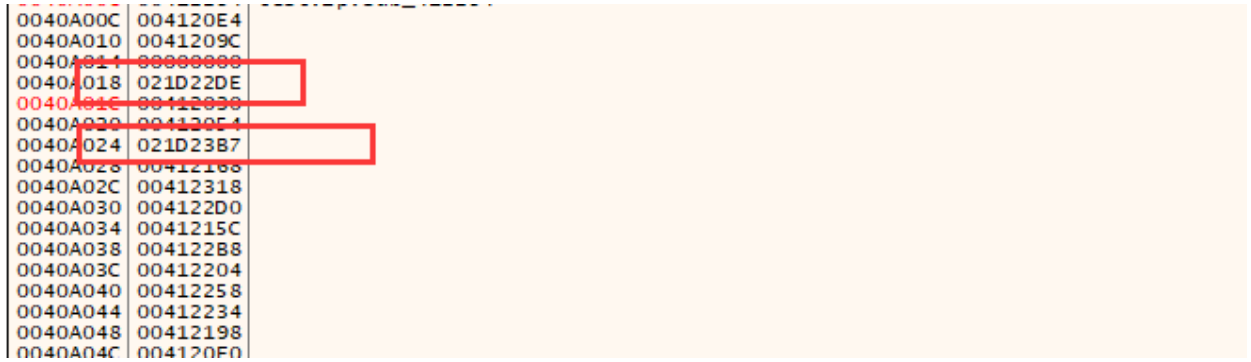


到这里，第二种保护手法就浮出水面了。

API代码抽离

这个API代码抽离，是我的猜测。

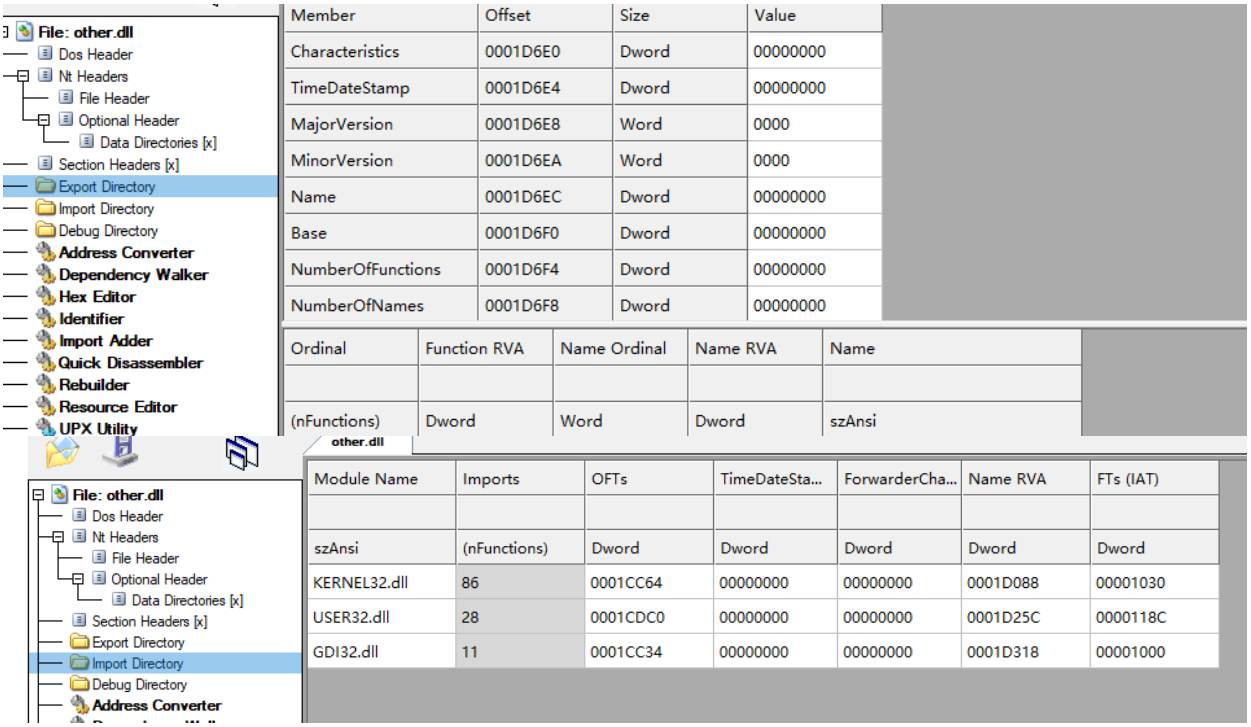
重新在内存窗口打开IAT表，



在这里的地址基本上都是0x0040开头的，但是有几个地址却不是的，它们是堆地址，如上图所圈出来的0x021d22de，在反汇编窗口中转到，

这里可以看到，这个地址处是一段位于堆中的代码，这段代码其实也是一个PE文件中一段代码，使用上面的手法将这个PE文件dump出来，使用CFF查看发现是个dll，

Member	Offset	Size	Value	Section
Export Directory RVA	00000158	Dword	0001D6E0	.text
Export Directory Size	0000015C	Dword	000005FB	
Import Directory RVA	00000160	Dword	0001CBE4	.text
Import Directory Size	00000164	Dword	00000050	



这里可以看到，虽然数据目录中还有导出表的信息，但是导出表已经被毁掉了，只剩下导入表。

这里整理一下思路，正常程序的导入表应该是连续的，从某个dll中导入的函数应该是在放在一个数组中，中间连续不断，直到遇到下一个其它dll的导入函数，不同dll的导入函数数组中间使用至少一个0隔断。而从第三种保护手法来看，一个导出函数的数组中，出现了两个dll的导入函数，这明显是处理过的。

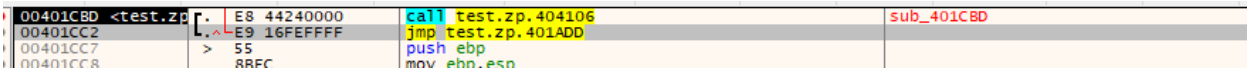
对这里，我的猜测是zp把api的代码给抽出来，自己实现了，但这只是猜测。

比较奇怪的是，我在这几个堆地址处设置了断点，但是自程序始终，这些断点都没有来。所以最后我根据直觉把这几个地址的填成了kernel32的api的地址。

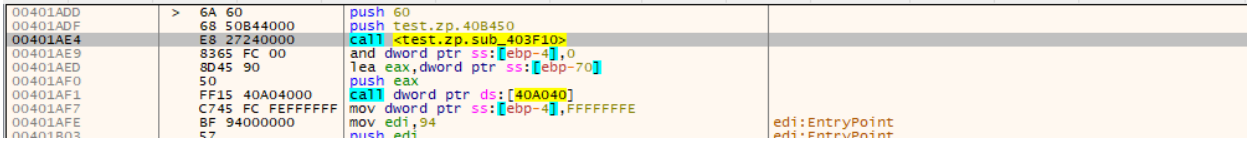
代码抽离混淆

除了对导入表混淆过之外，zp还把原exe的部分代码抽出来，混淆后放在了堆内存中执行。

定位到OEP，然后跟踪进入call下面的jmp，

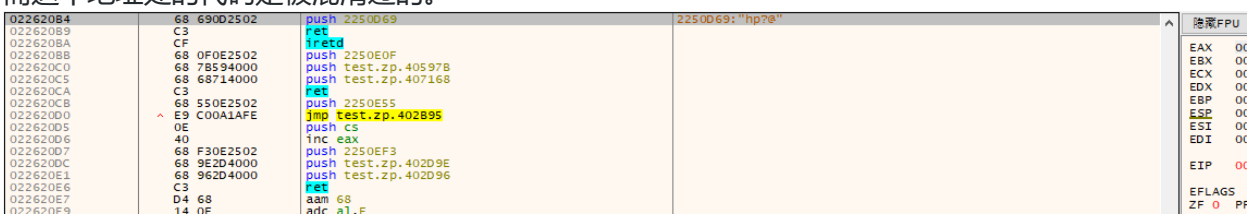


接着会遇到一个call 0x403f10，进到里面



会发现这个call会跳到一个堆地址处，

而这个地址处的代码是被混淆过的。



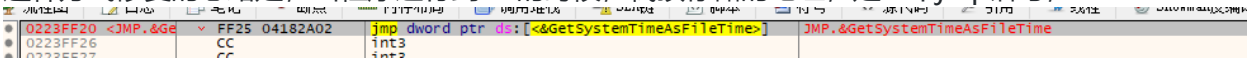
修复

IAT混淆修复

了解了IAT的保护方式，修复思路就很简单了。

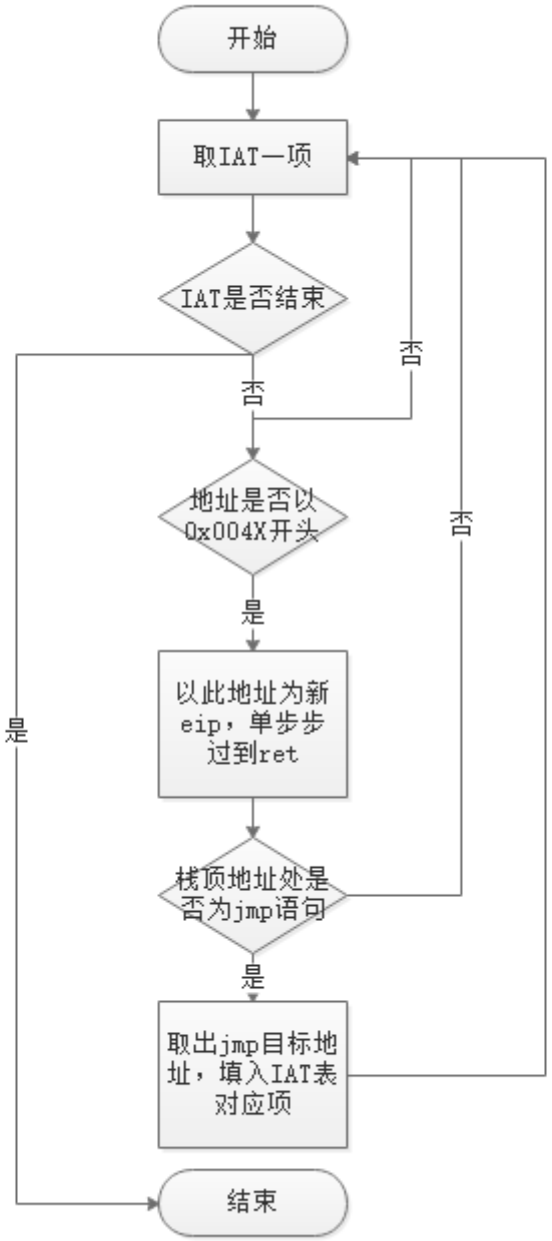
第一种方式修复

这种方式修复的思路是，当程序运行到ret的时候，栈顶存储的地址，是一个jmp语句，



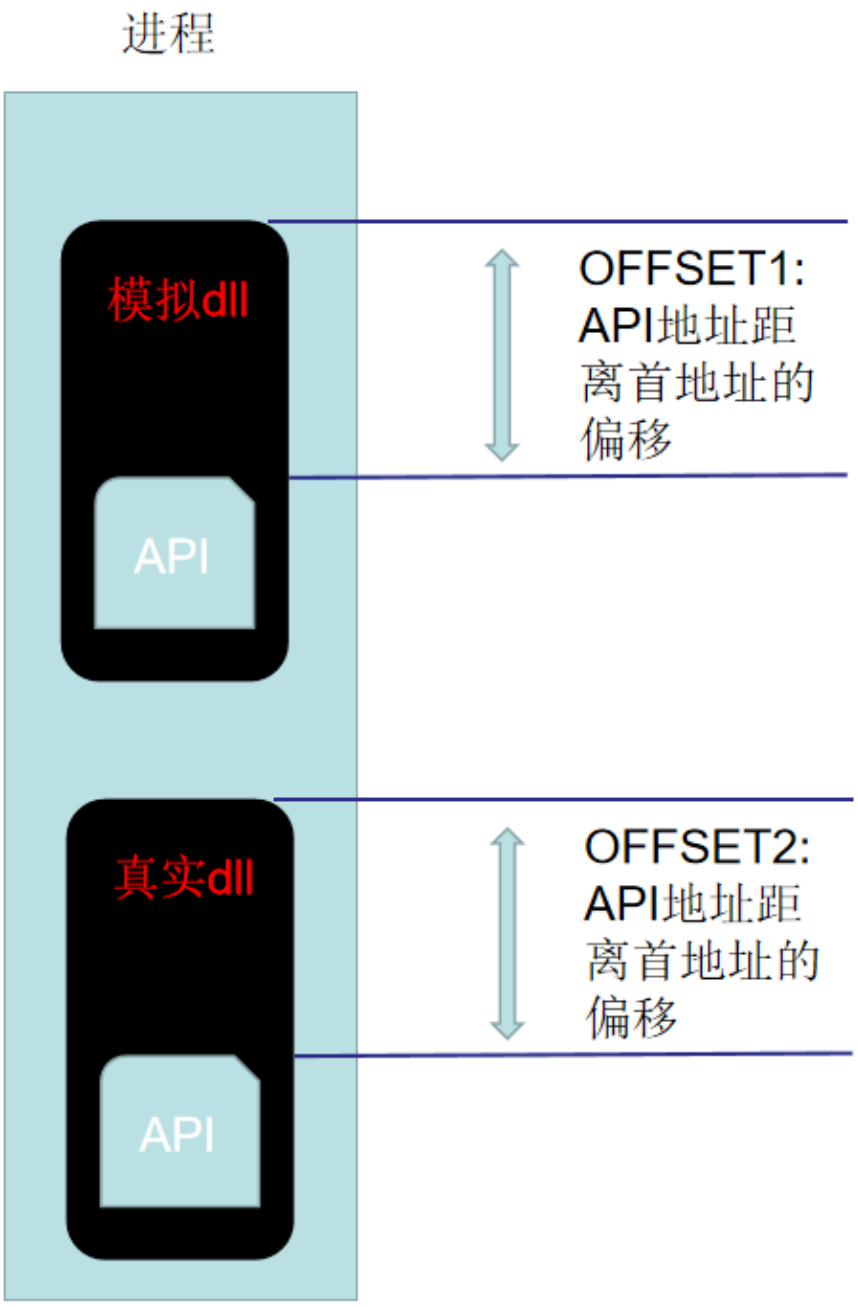
可以通过jmp语句获得真正API的地址。

所以，我这里选择写个x64dbg的python脚本，脚本会扫描IAT表，取出IAT表中以0x004XX开头的每一项，并以其为新eip，然后单步运行到ret，取出栈顶地址，并判断是否为jmp语句，如果是则将jmp的目标地址即API地址填入对应的IAT表中。



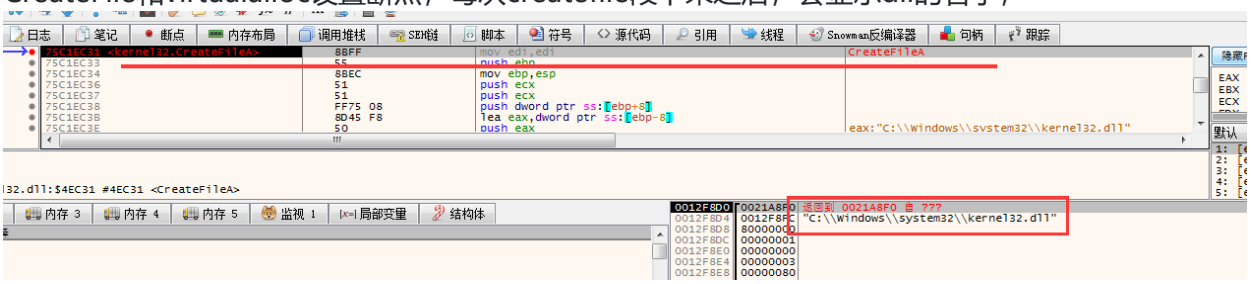
第二种方式修复

第二种方式是API模拟，在这种方式中，模拟API的在模拟dll中距离模块基址的偏移和真实API在真实DLL中距离首地址的偏移值是一样的。



如上图，Offset1和offset2的值是一样的。所以只要计算出offset1，然后加上真实dll的模块基址就是真正API的地址。

这里需要找到模拟dll在内存中的首地址和模拟dll与真实dll的对应关系，这个很好找，只要在CreateFile和virtualalloc设置断点，每次createfile段下来之后，会显示dll的名字，



然后继续运行，紧接着virtualalloc会段下来，而virutalalloc返回的地址就是前面createfile打开的真实dll的名字。

第三种修复方式

第三种保护方式，因为设置的断点并没有来，无法推测其用途，所以我最后只是凭直觉填了几个API的名字。可能，这只是个demo，第三种保护方式只有3个API。我分别填写了CreateFileA，CloseHanlde和SetFilePoint。

修复代码

这里的修复代码使用的是x64dbg的python插件，写的是python脚本。整体的流程是先获取模拟dll的地址和模拟dll与真实dll的对应关系，然后读取IAT表，判断修复类型，分别进行修复。

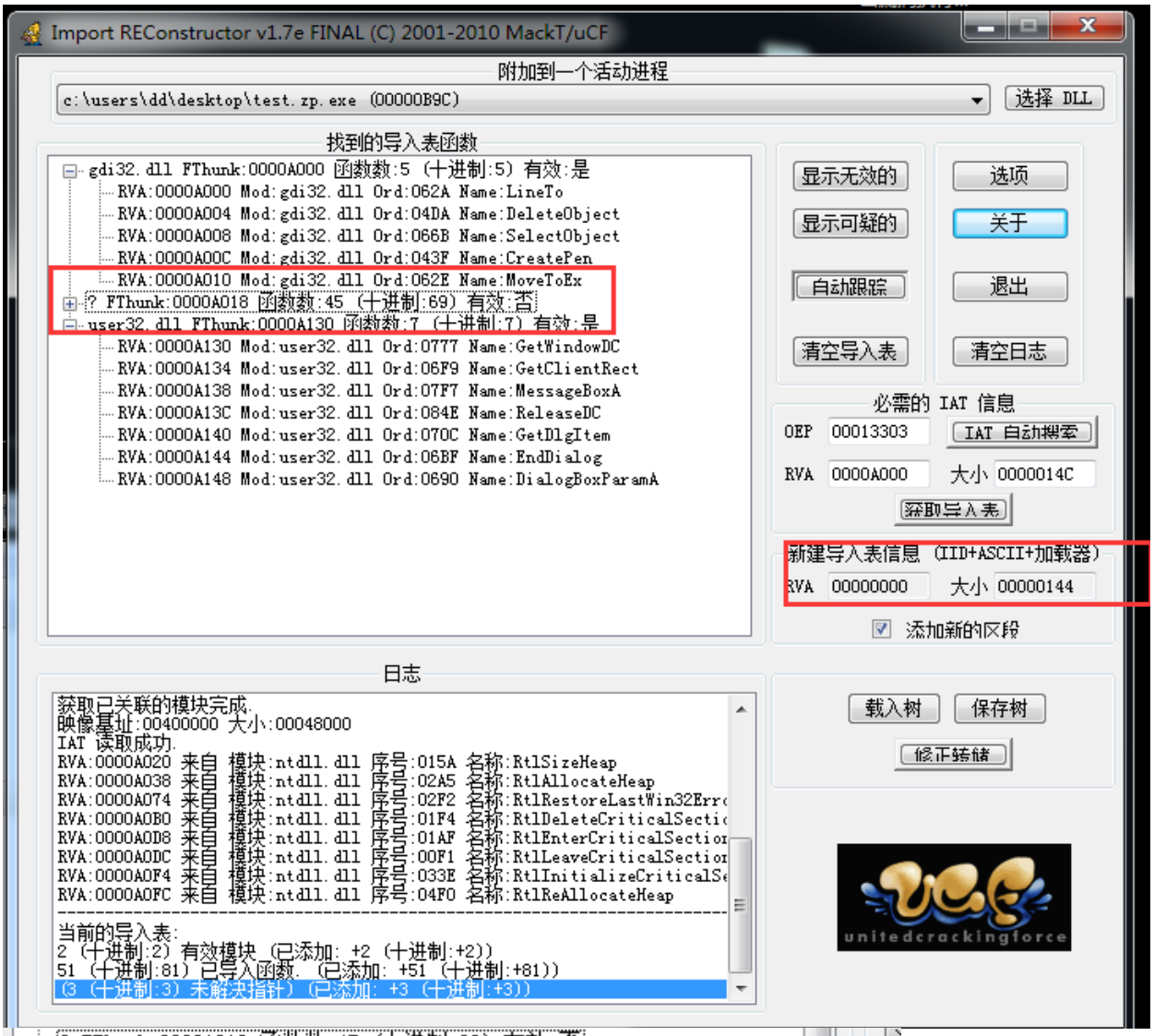
```
import x64dbgpy
from x64dbgpy.pluginsdk.x64dbg import *

addrOfKerbaseMo = 0
addrOfUserMo = 0
addrOfDGIMo = 0
def FixDllMonulate(espValue):
    addrOfKerbaseRe = BaseFromName("kernel32")
    sizeOfKerbase = 0xe0000

    addrOfUserRe = BaseFromName("user32")
    sizeOfUser = 0x00199000

    addrOfGDIRe = BaseFromName("gdi32")
    sizeOfGDI = 0x00023000

    if espValue > addrOfKerbaseMo and espValue < addrOfKerbaseMo + sizeOfKerbase:
        return espValue - addrOfKerbaseMo + addrOfKerbaseRe


    if espValue > addrOfUserMo and espValue < addrOfUserMo + sizeOfUser:
        return espValue - addrOfUserMo + addrOfUserRe


    if espValue > addrOfDGIMo and espValue < addrOfDGIMo + sizeOfGDI:
        return espValue - addrOfDGIMo + addrOfGDIRe

    return  espValue

def RunToRet():
    StepIn()
    StepIn()
    StepIn()
    while(1):
        StepOver()
        code = ReadByte( GetEIP())
        if(code == 0xC3):
            break;
    print("RunToRet EIP: %08x" % GetEIP())
```

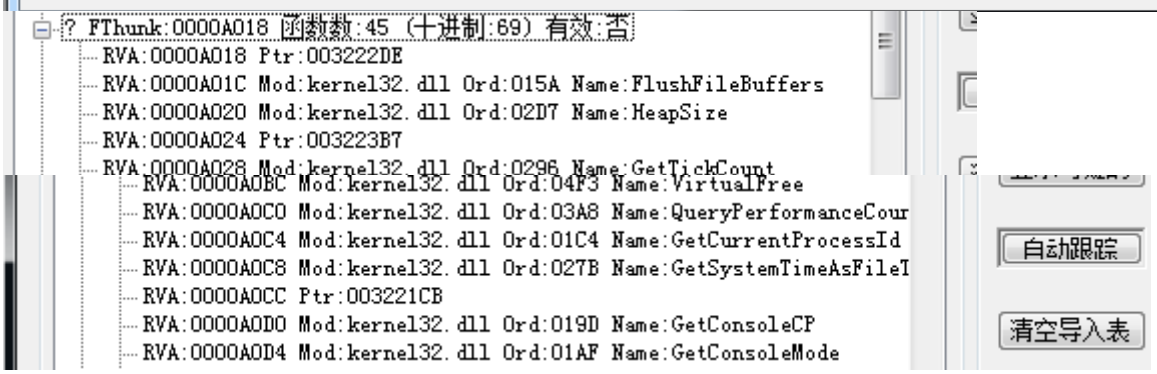
```
def RunToVirtualAllocRet():
    while(1):
        StepOver()
        code = ReadByte( GetEIP())
        if(code == 0xC2):
            break;
    print("RunToRet EIP: %08x" % GetEIP())

baseOfKernel = BaseFromName("kernel32")
print "baseOfKernel % 08X" % baseOfKernel
addrOfCreateFileA = 0
addrOfVirtualAlloc = 0

l = ListInfo()
Symbol_GetList(l)
symbolInfoList  = GetSymbolInfoList(l)
symbolLst = vectorGetSymbolInfoList(symbolInfoList)
for i in symbolLst:
    #print "va:%08X name:%s mod:%s"%(i.rva, i.name, i.mod)
    if (i.mod == "kernel32.dll"  and i.type == Export):
        if (i.name == "CreateFileA"):
            addrOfCreateFileA = i.rva + baseOfKernel
            print "va:%08X name:%s mod:%s"%(addrOfCreateFileA, i.name, i.mod)

        if (i.name == "VirtualAlloc"):
            addrOfVirtualAlloc = i.rva + baseOfKernel
            print "va:%08X name:%s mod:%s"%(addrOfVirtualAlloc, i.name, i.mod)


SetBreakpoint(addrOfCreateFileA)
Run()
Wait()
DeleteBreakpoint(addrOfCreateFileA)

i = 0
while(i < 3):
    SetBreakpoint(addrOfVirtualAlloc)
    Run()
    while(DbgIsRunning()):
        i = i
    print("pause")
    print "Wait EIP: %08x"%GetEIP()
    DeleteBreakpoint(addrOfVirtualAlloc)
    RunToVirtualAllocRet()
    if(i == 0):
        addrOfKerbaseMo = GetEAX()
        print "addrOfKerbaseMo %08X"%addrOfKerbaseMo
    if(i == 1):
        addrOfUserMo = GetEAX()
        print "addrOfUserMo %08X"%addrOfUserMo
    if(i == 2):
        addrOfDGIMo = GetEAX()
        print "addrOfDGIMo %08X"%addrOfDGIMo
    i = i+1

Run()
while(DbgIsRunning()):
```

```
    i = i

addrOfIATStart = 0x0040a000
addrOfIATEnd = 0x0040a14c

addrToHandle = addrOfIATStart
while(addrToHandle != addrOfIATEnd):
    addr = ReadDword(addrToHandle)
    if(addr == 0):
        addrToHandle = addrToHandle + 4
        continue
    print("addr: %08x " % addr)

    addrOfConfusionStart = 0x00410000
    addrOfConfusionEnd = 0x00420000
    if((addr < addrOfConfusionStart) or (addr > addrOfConfusionEnd)):
        print("befor FixDllMonulate addr: %08x " % addr)
        apiAddr = FixDllMonulate(addr)
        print("after FixDllMonulate addr: %08x ApiAddr: %08x" % (addr, apiAddr))
        WriteDword(addrToHandle, apiAddr)
        addrToHandle = addrToHandle + 4
        continue

    SetEIP(addr)
    print("EIP: %08x" % GetEIP())

    RunToRet()

    espAddr = GetESP()
    espValue = ReadDword(espAddr)
    print("EIP: %08x VAL: %08x" % (espAddr, espValue))

    codeOfBegin = ReadByte(espValue)
    print("Byte: %02x " % codeOfBegin)
    if codeOfBegin == 0xff:
        apiAddr = ReadDword(espValue + 2)
        print("JmpAddr: %08x ApiAddr: %08x" % (espValue + 2, apiAddr))
        apiAddr = ReadDword(apiAddr)
        print("apiAddr: %08x ApiAddr: %08x" % (espValue + 2, apiAddr))
        WriteDword(addrToHandle, apiAddr)
    else:
        apiAddr = FixDllMonulate(espValue)
        print("FixDllMonulate espValue: %08x ApiAddr: %08x" % (espValue, apiAddr))
        WriteDword(addrToHandle, apiAddr)

    addrToHandle = addrToHandle + 4
```

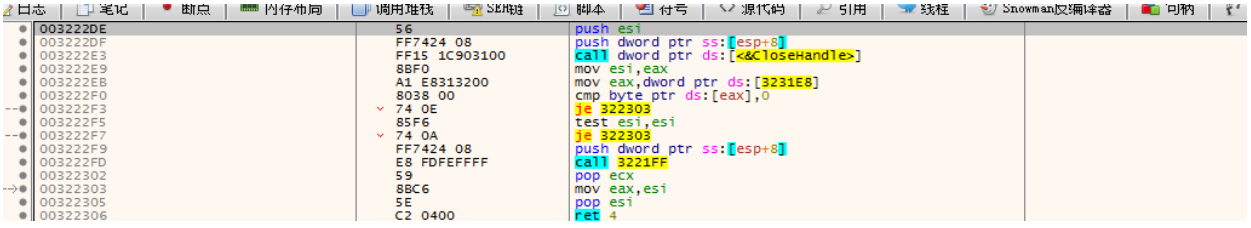
0040A000	7606F496	gdi32.LineTo
0040A004	76065F14	gdi32.DeleteObject
0040A008	76066640	gdi32.SelectObject
0040A00C	7606F3DE	gdi32.CreatePen
0040A010	76068B21	gdi32.MoveToEx
0040A014	00000000	
0040A018	003222DE	
0040A01C	75C0874F	kernel32.FlushFileBuffers
0040A020	777F99E0	ntdll.RtlSizeHeap
0040A024	003223B7	
0040A028	75C1C460	kernel32.GetTickCount
0040A02C	75C29107	kernel32.GetCommandLineA
0040A030	75C1C590	kernel32.HeapFree
0040A034	75C1DF50	kernel32.GetVersionExA
0040A038	777F296E	ntdll.RtlAllocateHeap
0040A03C	75C1FF05	kernel32.GetProcessHeap
0040A040	75BD1E10	kernel32.GetStartupInfoA
0040A044	75C12E0D	kernel32.TerminateProcess
0040A048	75C1D970	kernel32.GetCurrentProcess
0040A04C	75C308B9	kernel32.UnhandledExceptionFilter
0040A050	75C1F6CB	kernel32.SetUnhandledExceptionFilter
0040A054	75C17FF2	kernel32.IsDebuggerPresent
0040A058	75C1CE64	kernel32.GetProcAddress
0040A05C	75C1DAC3	kernel32.GetModuleHandleA
0040A060	75C1F930	kernel32.TlsGetValue
0040A064	75C1D9D4	kernel32.TlsAlloc
0040A068	75C1F953	kernel32.TlsSetValue
0040A06C	75C25489	kernel32.TlsFree
0040A070	75C1C580	kernel32.InterlockedIncrement
0040A074	777F2C93	ntdll.RtlRestoreLastWin32Error
0040A078	75C1C5E0	kernel32.GetCurrentThreadId
0040A07C	75C1CFB0	kernel32.GetLastError
0040A080	75C1C5B0	kernel32.InterlockedDecrement
0040A084	75C2BE5A	kernel32.ExitProcess
0040A088	75C2561E	kernel32.WriteFile
0040A08C	75C290EF	kernel32.GetStdHandle
0040A090	75C1D92A	kernel32.GetModuleFileNameA
0040A094	75C2CAEA	kernel32.FreeEnvironmentStringsA
0040A098	75C2CB02	kernel32.GetEnvironmentStrings
0040A09C	75C26DDC	kernel32.FreeEnvironmentStringsW
0040A0A0	75C1F0CA	kernel32.WideCharToMultiByte
0040A0A4	75C26DF4	kernel32.GetEnvironmentStringsW
0040A0A8	75C29119	kernel32.SetHandleCount
0040A0AC	75C26CE4	kernel32.GetFileType
0040A0B0	777F98B9	ntdll.RtlDeleteCriticalSection
0040A0B4	75C12DDD	kernel32.HeapDestroy
0040A0B8	75C1F144	kernel32.HeapCreate
0040A0BC	75C26D45	kernel32.VirtualFree
0040A0C0	75C1C5F2	kernel32.QueryPerformanceCounter
0040A0C4	75C1D985	kernel32.GetCurrentProcessId
0040A0C8	75C1D9E6	kernel32.GetSystemTimeAsFileTime
0040A0CC	003221CB	
0040A0D0	75C1C0F9	kernel32.GetConsoleCP
0040A0D4	75C2C388	kernel32.GetConsoleMode
0040A0D8	777E7290	ntdll.RtlEnterCriticalSection
0040A0DC	777E7250	ntdll.RtlLeaveCriticalSection
0040A0E0	75C290D7	kernel32.GetCPInfo
0040A0E4	75C1DADB	kernel32.GetACP
0040A0E8	75C1461A	kernel32.GetOEMCP
0040A0EC	75C1C446	kernel32.Sleep

代码跑完之后，修复的IAT可见上图。

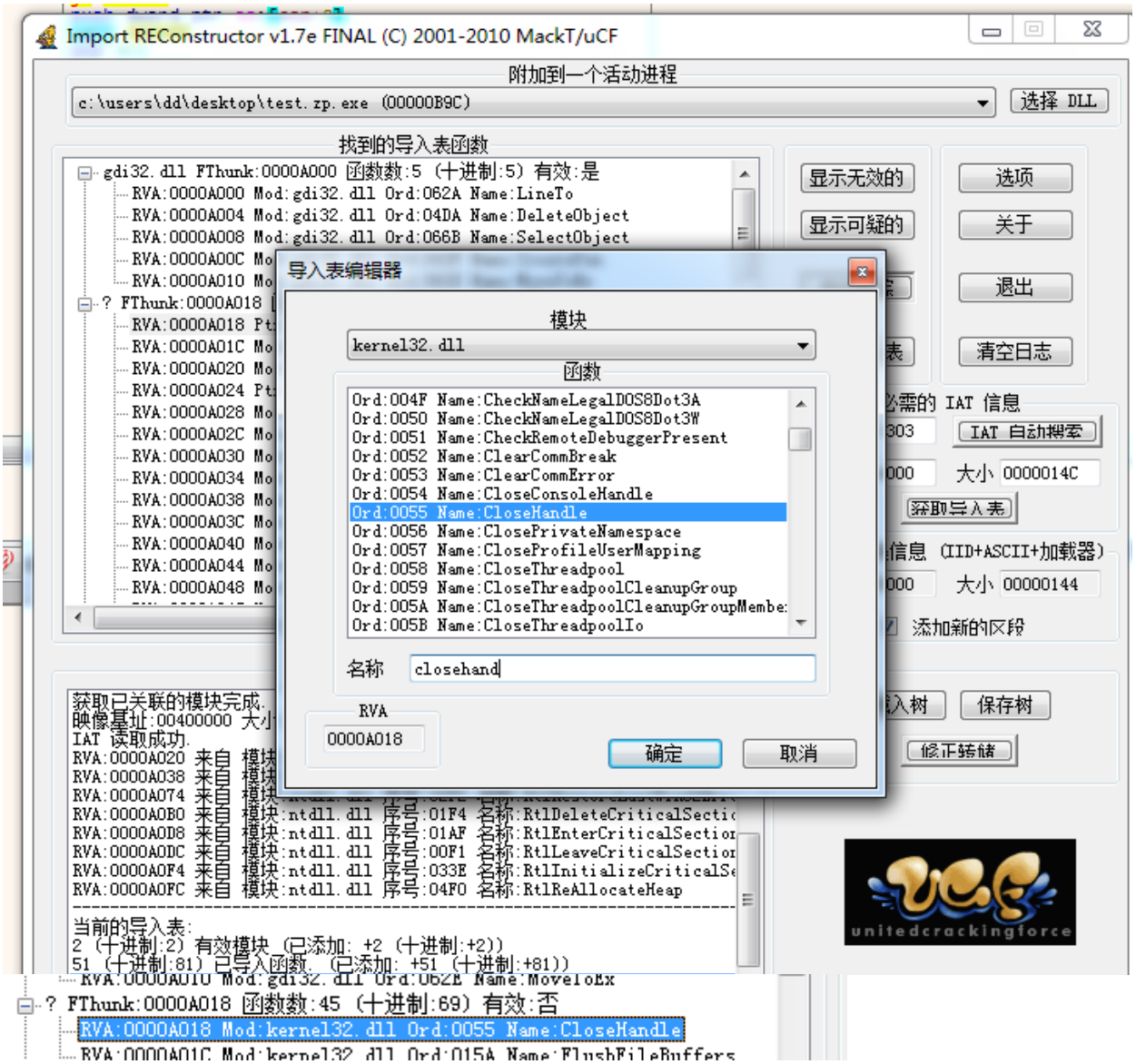




使用ImportRec打开，会看到ker32的3个api没有找到地址，这里其实可以设置断点，然后通过上下文来判断是什么东西，但是我设置的断点没有来，所以也就凭直觉随便填了API的名字，比如第一个，



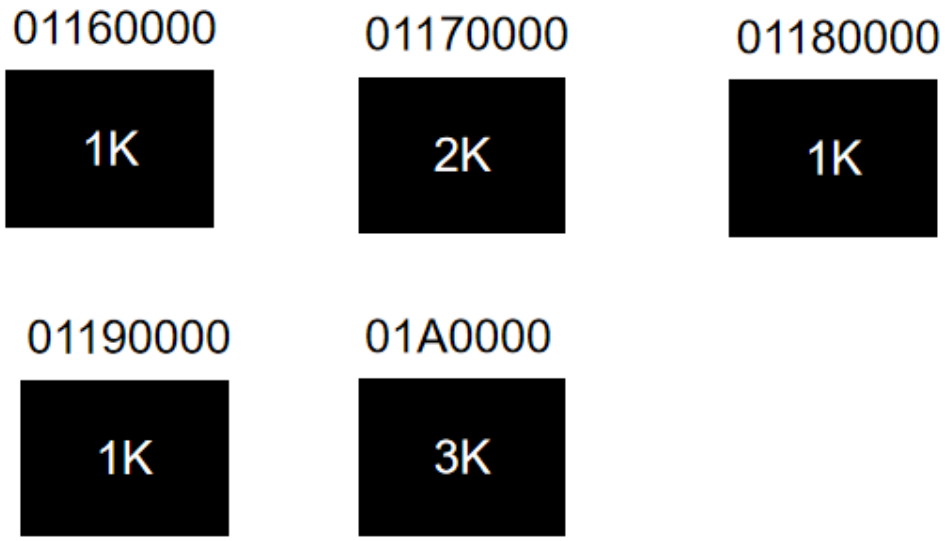
这里我看到调用了CLoseHandle，所以我就把第一个未识别的API的名字改成了CloseHandle，



其它的依次类推。

代码抽离混淆修复

zp将代码抽离并混淆后放在了诸多页面中，这些页面是不连续的。



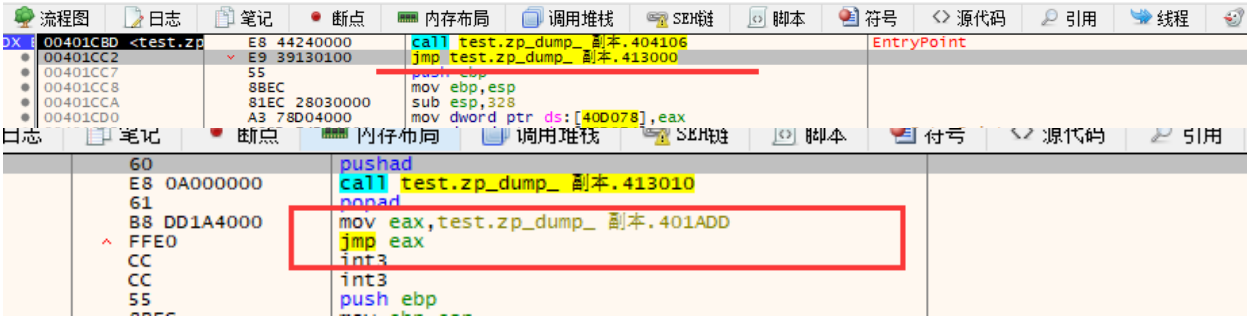
由上图可见，混淆代码分布在5个部分中。
这里的修复思路是将这些混淆代码所在的内存页dmp出来，然后做成PE的节，并修复其中的绝对地址。
这里我并没有去这么做，只是为了验证这个思路，给dmp程序加了个补丁。补丁的思路很简单，就是在程序启动的时候，申请对应的内存然后把dmp出来的内存页放到对应内存中，完了之后执行原来的主程序代码。补丁的代码见下面：

```
__declspec(naked) void  LoadData()
{
    __asm {
        pushad;
        call FixFunc;
        popad;
        mov eax, 0x00401ADD;
        jmp eax;
    }
}

void FixFunc()
{
    char szLoadLibraryA[] = { 'L', 'o', 'a', 'd', 'L', 'i', 'b', 'r', 'a', 'r', 'y', 'A',
'\0' };

    char szGetModuleHandle[] = SCA_SZ_GetModuleHandle;
    char szCreateDecompressor[] = SCA_SZ_CreateDecompressor;
    char szDecompress[] = SCA_SZ_Decompress;
    char szVirtualAlloc[] = SCA_SZ_VirtualAlloc;
    char szCreateFileA[] = SCA_SZ_CreateFileA;
    char szReadFile[] = SCA_SZ_ReadFile;
    char szCloseHandle[] = SCA_SZ_CloseHandle;

    PFN_LoadLibraryA pfnLoadLibraryA = (PFN_LoadLibraryA)MyGetProcAddress(GetKernel32(),
szLoadLibraryA);
```

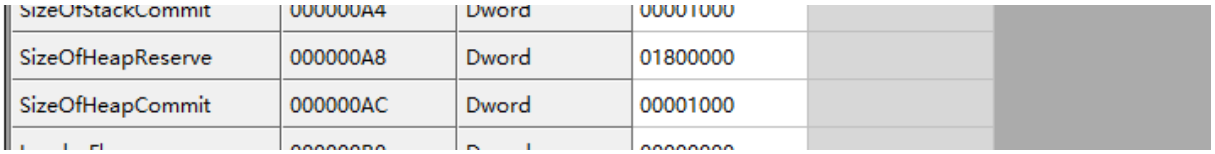
```
    PFN_MyGetProcAddress pfnGetProcAddress =
(PFN_MyGetProcAddress)MyGetProcAddress(GetKernel32(), szGetProcAddress);
    SCA_PFN_GetModuleHandle pfnGetModuleHandle =
(SCA_PFN_GetModuleHandle)pfnGetProcAddress(GetKernel32(), szGetModuleHandle);
    HMODULE hModCabinet = pfnLoadLibraryA(szCabinet);
    SCA_PFN_VirtualAlloc pfnVirtualAlloc =
(SCA_PFN_VirtualAlloc)pfnGetProcAddress(GetKernel32(), szVirtualAlloc);
    SCA_PFN_CreateFileA  pfnCreateFileA =
(SCA_PFN_CreateFileA)pfnGetProcAddress(GetKernel32(), szCreateFileA);
    SCA_PFN_ReadFile  pfnReadFile = (SCA_PFN_ReadFile)pfnGetProcAddress(GetKernel32(),
szReadFile);
    SCA_PFN_CloseHandle pfnCloseHandle =
(SCA_PFN_CloseHandle)pfnGetProcAddress(GetKernel32(), szCloseHandle);
    /*
    *  申请内存空间
    */
    DWORD dwAddrOfData = 0x01160000;
    DWORD dwSizeOfData = 0x50000;
    LPVOID pAddrOfData = pfnVirtualAlloc((LPVOID)dwAddrOfData, dwSizeOfData, MEM_RESERVE,
PAGE_EXECUTE_READWRITE);
    pAddrOfData = pfnVirtualAlloc((LPVOID)dwAddrOfData, dwSizeOfData, MEM_COMMIT,
PAGE_EXECUTE_READWRITE);
    if (pAddrOfData == NULL)
    {
        return;
    }

    DataInof dis[5] = {
        {
            0x01160000,
            0x1000,
            {'6', '1', '\0'}
        },
        {
            0x01170000,
            0x2000,
            { '7', '2', '\0' }
        },
        {
            0x01180000,
            0x1000,
            { '8', '1', '\0' }
        },
        {
            0x01190000,
            0x1000,
            { '9', '1', '\0' }
        },
        {
            0x011a0000,
            0x3000,
            { 'a', '3', '\0' }
        }
    };

    /*
    *  读取数据
    */
```

```
    for (int i = 0; i < sizeof(dis) / sizeof(dis[0]); ++i)
    {
        HANDLE hFile = pfnCreateFileA((LPCTSTR)dis[i].m_szName, GENERIC_READ |
GENERIC_WRITE, FILE_SHARE_READ, NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

        DWORD dwBytesToRead = 0;
        DWORD dwRet = pfnReadFile(hFile, (LPVOID)dis[i].m_dwAddr, dis[i].m_dwSize,
&dwBytesToRead, NULL);

        pfnCloseHandle(hFile);
    }
}
```

补丁的代码被我放在的原zp的壳代码的节，在程序启动的时候，跳转到我的补丁代码，执行完补丁代码之后，再跳转到原程序代码。



这里需要注意的是，我每次申请内存的时候，总是申请不到，所以我把PE文件中的堆保留给扩大了，



SizeOfStackCommit	000000A4	Dword	00001000
SizeOfHeapReserve	000000A8	Dword	01800000
SizeOfHeapCommit	000000AC	Dword	00001000

扩大了堆保留之后，我就可以申请到指定地址了。

代码抽离其它思路

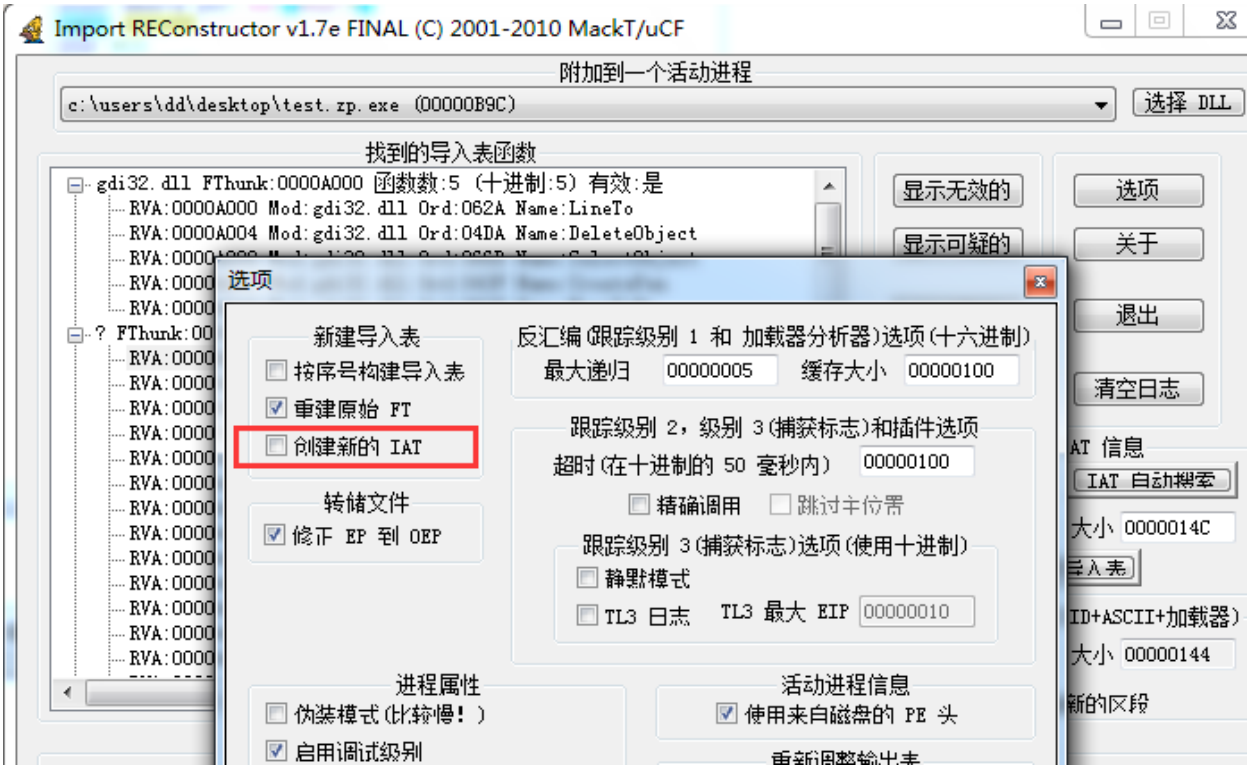
上面的方式肯定是最不好的方式，其实还有其它的思路，这里只谈思路，因为时间的关系，就不给出实现了。

- 思路1
  跟踪壳代码，找到抽离代码的位置，强转jmp，使其跳过代码抽离
- 思路2
  写IDA脚本，扫描代码，找到混淆的代码的部分，然后通过x64dbg的python脚本，单步跟踪，将代码记录下来，去混淆然后还原原来的代码。

个人觉得思路1其实是完美的，但是壳代码本身带有混淆，不好跟踪，所以思路2其实也是不错的。

其它注意

dmp出来的程序，在使用ImpRec修复导入表的时候，注意不要让ImpRec创建新的IAT表，不然抽离出来的代码中调用API的代码就废了。


Import REConstructor v1.7e FINAL (C) 2001-2010 MackT/uCF
附加到一个活动进程
c:\users\dd\desktop\test.zp.exe (00000B9C)
选择 DLL
找到的导入表函数
gdi32.dll FThunk:0000A000 函数数:5 (十进制:5) 有效:是
RVA:0000A000 Mod:gdi32.dll Ord:062A Name:LineTo
RVA:0000A004 Mod:gdi32.dll Ord:04DA Name:DeleteObject
显示无效的
显示可疑的
选项
关于
退出
清空日志
选项
新建导入表
按序号构建导入表
重建原始 FT
创建新的 IAT
转储文件
修正 EP 到 OEP
反汇编(跟踪级别 1 和 加载器分析器)选项(十六进制)
最大递归 00000005
缓存大小 00000100
跟踪级别 2，级别 3(捕获标志)和插件选项
超时(在十进制的 50 毫秒内) 00000100
精确调用
跳过半位置
跟踪级别 3(捕获标志)选项(使用十进制)
静默模式
TL3 日志
TL3 最大 EIP 00000010
进程属性
伪装模式(比较慢！)
启用调试级别
活动进程信息
使用来自磁盘的 PE 头
AT 信息
IAT 自动搜索
大小 0000014C
入表
ID+ASCII+加载器)
大小 00000144
新的区段