

# 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

0040A000	00412360	
0040A004	00412288	
0040A008	00412294	test.zp.sub_412294
0040A00C	004120E4	
0040A010	0041209C	
0040A014	00000000	
0040A018	006522DE	
0040A01C	00412030	
0040A020	00412054	
0040A024	006523B7	
0040A028	00412168	
0040A02C	00412318	
0040A030	004122D0	
0040A034	0041215C	
0040A038	00412288	
0040A03C	00412204	
0040A040	00412258	
0040A044	00412234	
0040A048	00412198	
0040A04C	004120F0	
0040A050	00412354	
0040A054	004121C8	
0040A058	004121E0	
0040A05C	00412114	
0040A060	00412174	
0040A064	0041233C	
0040A068	00412300	
0040A06C	0041227C	
0040A070	004120A8	
0040A074	004122F4	
0040A078	004120FC	

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

00401C8D	<test.zp	call test.zp.404106	sub_401C8D
00401CC2		jmp test.zp.401ADD	
00401CC7		push ebp	
00401CC8		mov ebp,esp	
00401CCA		sub esp,328	
00401CD0		mov dword ptr ds:[40D078],eax	

进到第一个call里面，跟进第一个调用API的地方，

00404106	\$	push ebp	
00404107		mov ebp,esp	
00404109		sub esp,10	
0040410C		mov eax,dword ptr ds:[40C004]	
00404111		and dword ptr ss:[ebp-8],0	
00404115		and dword ptr ss:[ebp-4],0	
00404119		push ebx	
0040411A		push edi	
00404118		mov edi,8B40E64E	edi:EntryPoint
00404120		cmp eax,edi	edi:EntryPoint
00404122		mov ebx,FFFF0000	edi:EntryPoint
00404127		je test.zp.404136	
00404129	<test.zp	test ebx,eax	sub_404129
00404128		je test.zp.404136	
0040412D		not eax	
0040412F		mov dword ptr ds:[40C008],eax	
00404134		jmp test.zp.404196	
00404136		push esi	esi:EntryPoint
00404137		lea eax,dword ptr ss:[ebp-8]	
0040413A		push eax	
00404138		call dword ptr ds:[40A0C8]	
00404141		mov esi,dword ptr ss:[ebp-4]	esi:EntryPoint
00404144		xor esi,dword ptr ss:[ebp-8]	esi:EntryPoint

004121A4	push 573555DA
004121A9	jmp test.zp.41200C
004121AE	bound ebx,qword ptr ds:[ebx+68]

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

0019FF50	00413303	test.zp.EntryPoint
0019FF54	0019FF80	
0019FF58	0019FF6C	
0019FF5C	00293000	
0019FF60	00413303	test.zp.EntryPoint
0019FF64	00413303	test.zp.EntryPoint
0019FF68	0019FFCC	
0019FF6C	00000202	
0019FF70	023FFF20	
0019FF74	77240419	返回到 kernel32.77240419 自 ???
0019FF78	00293000	
0019FF7C	77240400	kernel32.77240400
0019FF80	0019FFDC	
0019FF84	77C9662D	返回到 ntdll.77C9662D 自 ???
0019FF88	00293000	

然后在反汇编窗口中转到这个值，

023FFF1E	CC	int3	
023FFF1F	CC	int3	
023FFF20	<JMP,&Ge	FF25 04184602	jmp dword ptr ds:[<&GetSystemTimeAsFileTime>] JMP,&GetSystemTimeAsFileTime
023FFF26	CC	int3	
023FFF27	CC	int3	
023FFF28	CC	int3	
023FFF29	CC	int3	

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，

00412360	68 E6553557	push 573555E6
00412365	E9 A2FCFFFF	jmp test.zp.41200C
0041236A	302F	xor byte ptr ds:[edi],ch
0041236C	68 84553557	push 573555B4
00412371	E9 96FCFFFF	jmp test.zp.41200C
00412376	99	cdq
00412377	3368 96	xor ebp,dword ptr ds:[eax-6A]
0041237A	55	push ebp

.textbss:00412360 test.zp.exe:\$12360 #0		
--	--	--

内存 1	内存 2	内存 3	内存 4	内存 5	监视 1	[x=] 局部变量	结构体
------	------	------	------	------	------	-----------	-----

地址	值	注释
0040A000	00412360	[dec ebp (用户代码)]

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

0019FF60	00413303	test.zp.EntryPoint
0019FF64	00413303	test.zp.EntryPoint
0019FF68	0019FFCC	
0019FF6C	00000202	
0019FF70	02304550	
0019FF74	77240419	返回到 kernel32.77240419 自 ???
0019FF78	00386000	
0019FF7C	77240400	kernel32.77240400
0019FF80	0019FFDC	

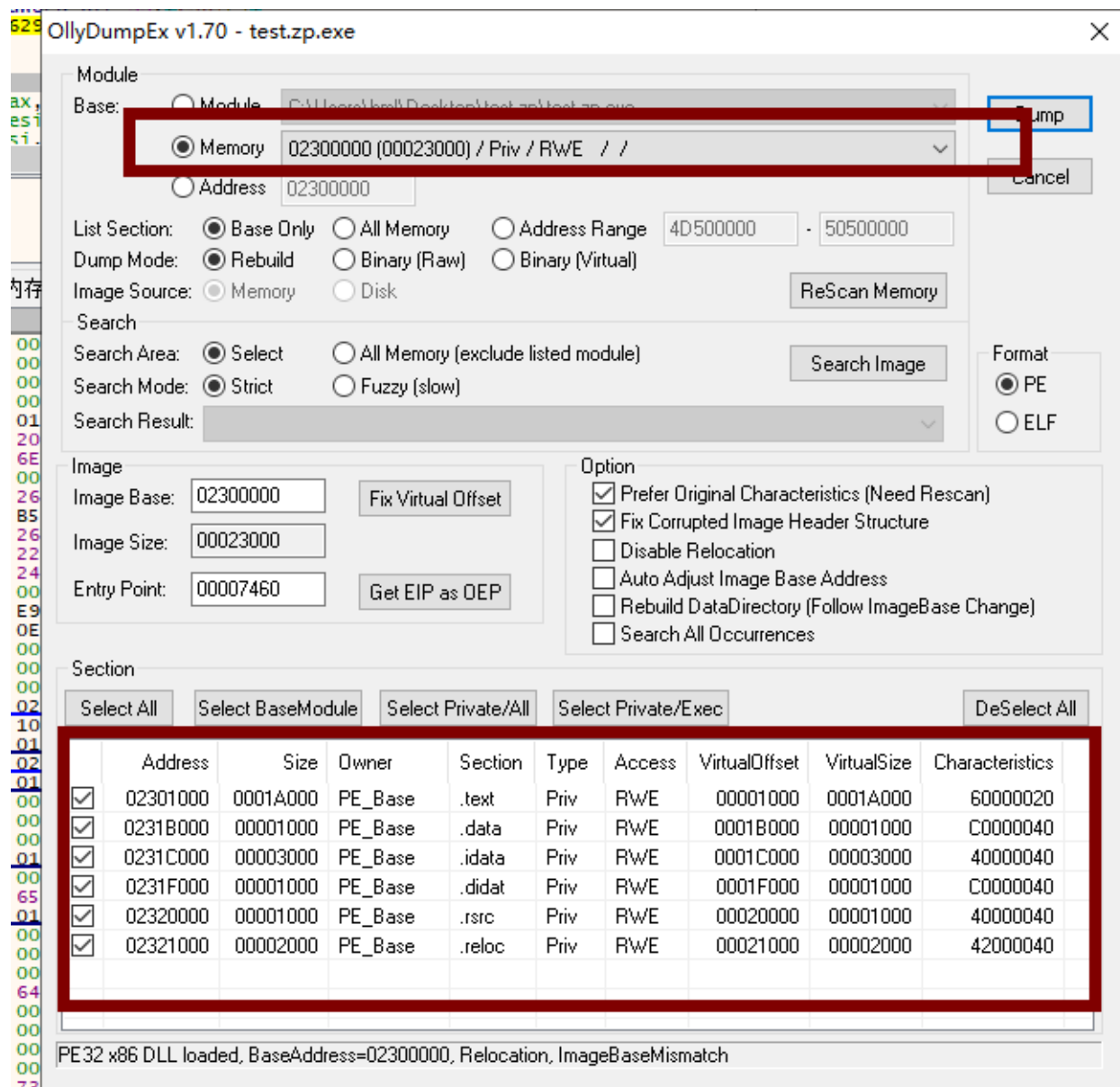
在栈上这个地址右键转到内存布局，然后在内存布局中查看地址所在的内存范围，

00A84000	001CC000	保留 (00A80000)		MAP	-R---	-R---
00C80000	00181000			MAP	-R---	
00E10000	00241000			MAP	-R---	
01051000	011C0000	保留 (00E10000)		MAP	-R---	
02220000	000E0000			PRV	ERW--	ERW--
02300000	00023000			PRV	ERW--	ERW--
02360000	00003000			PRV	-RW--	-RW--
02363000	00000000	保留 (02360000)		PRV	-RW--	-RW--
02370000	00012000			PRV	-RW--	-RW--
02382000	000EE000	保留 (02370000)		PRV	-RW--	-RW--

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：

地址	十六进制	ASCII
02300000	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZ.....ÿÿ..
02300010	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00	.....@.....
02300020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
02300030	00 00 00 00 00 00 00 00 00 00 00 00 E0 00 00 00	.....à....
02300040	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	.°.!.!LITh
02300050	72 6F 67 72 61 6D 20 63 61 6E 6E 6F is program canno	
02300060	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20 t be run in DOS	
02300070	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 mode...\$.....	
02300080	08 A5 48 7A 4C C4 26 29 4C C4 26 29 4C C4 26 29 .%HzL&)&)L&)&	
02300090	29 A2 27 28 4E C4 26 29 45 BC B5 29 55 C4 26 29 )e"(NA&)E4p)U&)	
023000A0	4C C4 27 29 D4 C7 26 29 29 A2 26 28 4D C4 26 29 LÂ')ôÇ&))e&(M&)&	
023000B0	29 A2 28 28 47 C4 26 29 29 A2 22 28 42 C4 26 29 )e+(GA&))e"(BA&)	
023000C0	29 A2 D9 29 4D C4 26 29 29 A2 24 28 4D C4 26 29 )cU)MA&))e\$(M&)&	
023000D0	52 69 63 68 4C C4 26 29 00 00 00 00 00 00 00 RichL&)...	
023000E0	50 45 00 00 4C 01 06 00 58 45 E9 AB 00 00 00 PE...L...XEé«....	
023000F0	00 00 00 00 E0 00 02 21 08 01 0E 0D 00 98 01 00	.....â...!

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
0000046F	00008BC0	0087	00015CD0	D3DKMTCreateOverlay

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，

VA	02315362
RVA	00015362
File Offset	00015362

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Ascii
00015360	CD	03	47	44	49	33	32	2E	64	6C	6C	00	70	6C	64	63	i\GDI32.dll.pldc
00015370	47	65	74	00	41	62	6F	72	74	44	6F	63	00	41	62	6F	64 6F 6E 74 4D 65
00015380	72	74	50	61	74	68	00	41	64	64	46	6F	6E	74	4D	65	rtPath.AddFontMe
00015390	6D	52	65	73	6F	75	72	63	65	45	78	00	41	64	64	46	mResourceEx.AddF
000153A0	6F	6E	74	52	65	73	6F	75	72	63	65	41	00	41	64	64	ontResourceA.Add
000153B0	46	6F	6E	74	52	65	73	6F	75	72	63	65	45	78	41	00	FontResourceExA.

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，

0040A00C	004120E4
0040A010	0041209C
0040A014	88888888
0040A018	021D22DE
0040A01E	88412038
0040A020	004120C4
0040A024	021D23B7
0040A028	00412188
0040A02C	00412318
0040A030	004122D0
0040A034	0041215C
0040A038	004122B8
0040A03C	00412204
0040A040	00412258
0040A044	00412234
0040A048	00412198
0040A04C	004120F0

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，

021D22D7	3E	push esi	esi:EntryPoint
021D22DA	5D	pop ebp	
021D22DB	C2 1800	ret 18	
021D22DE	56	push esi	esi:EntryPoint
021D22DF	FF7424 08	push dword ptr ss:[esp+8]	
021D22E3	FF15 1C901C02	call dword ptr ds:[&C10seHandle]	
021D22E9	8BF0	mov esi,eax	esi:EntryPoint
021D22EB	A1 E8311D02	mov eax,dword ptr ds:[21D31E8]	
021D22F0	8038 00	cmp byte ptr ds:[eax],0	
021D22F3	74 0E	jz 21D2303	
021D22F5	85F6	test esi,esi	esi:EntryPoint
021D22F7	74 0A	jz 21D2303	
021D22F9	FF7424 08	push dword ptr ss:[esp+8]	
021D22FD	E8 FDFEFFFF	call 21D21FF	
021D2302	59	pop ecx	ecx:EntryPoint
021D2303	8BC6	mov eax,esi	esi:EntryPoint
021D2305	5E	pop esi	esi:EntryPoint
021D2306	C2 0400	ret 4	
021D2309	56	push esi	esi:EntryPoint
021D230A	FF7424 10	push dword ptr ss:[esp+10]	
021D230E	FF7424 10	push dword ptr ss:[esp+10]	
021D2312	FF7424 10	push dword ptr ss:[esp+10]	
021D2316	FF15 18901C02	call dword ptr ds:[&OpenFile]	
021D231C	8BF0	mov esi,eax	esi:EntryPoint

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	

File: other.dll

Dos Header

Nt Headers

File Header

Optional Header

Data Directories [x]

Section Headers [x]

Export Directory

Import Directory

Debug Directory

Address Converter

Dependency Walker

Hex Editor

Identifier

Import Adder

Quick Disassembler

Rebuilder

Resource Editor

UPX Utility

Member	Offset	Size	Value
Characteristics	0001D6E0	Dword	00000000
TimeDateStamp	0001D6E4	Dword	00000000
MajorVersion	0001D6E8	Word	0000
MinorVersion	0001D6EA	Word	0000
Name	0001D6EC	Dword	00000000
Base	0001D6F0	Dword	00000000
NumberOfFunctions	0001D6F4	Dword	00000000
NumberOfNames	0001D6F8	Dword	00000000

Ordinal	Function RVA	Name Ordinal	Name RVA	Name
(nFunctions)	Dword	Word	Dword	szAnsi

other.dll

Module Name	Imports	OFTs	TimeDateSta...	ForwarderCha...	Name RVA	FTs (IAT)
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
KERNEL32.dll	86	0001CC64	00000000	00000000	0001D088	00001030
USER32.dll	28	0001CDC0	00000000	00000000	0001D25C	0000118C
GDI32.dll	11	0001CC34	00000000	00000000	0001D318	00001000

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，

00401CBD	<test.zp	E8 44240000	call test.zp.404106	Sub_401CBD
00401CC2	E9 16FEFFFF	jmp test.zp.401ADD		
00401CC7	> 55	push ebp		
00401CC8	RRFC	mov ehb.esb		

接着会遇到一个call 0x403f10，进到里面

00401ADD	> 6A 60	push 60		
00401ADF	68 50B44000	push test.zp.40B450		
00401AE4	E8 27240000	call <test.zp.sub_403F10>		
00401AE9	8365 FC 00	and dword ptr ss:[ebp-4],0		
00401AED	8D45 90	lea eax,dword ptr ss:[ebp-70]		
00401AF0	50	push eax		
00401AF1	FF15 40A04000	call dword ptr ds:[40A040]		
00401AF7	C745 FC FFFFFFFF	mov dword ptr ss:[ebp-4],FFFFFFF		
00401AFE	BF 94000000	mov edi,94		
00401B03	57	push edi		

会发现这个call会跳到一个堆地址处，

00403F10	<test.zp	E9 9FE1E501	jmp 2262084	sub_403F10
00403F15	1C 8B	sub al,8B		
00403F17	4B	dec eax		
00403F18	6C	insb		

而这个地址处的代码是被混淆过的。

02262084	68 69002502	push 2250069	2250069:"hp?e"	
02262089	C3	ret		
0226208A	CF	hretd		
0226208B	68 0F0E2502	push 22500EF		
022620C0	68 7B594000	push test.zp.40597B		
022620C5	68 6B714000	push test.zp.40716B		
022620CA	C3	ret		
022620CB	68 550E2502	push 2250E55		
022620D0	E9 C00A1AFE	jmp test.zp.402B95		
022620D5	0E	push cs		
022620D6	40	inc eax		
022620D7	68 F30E2502	push 2250EF3		
022620DC	68 9E2D4000	push test.zp.402D9E		
022620E1	68 9E2D4000	push test.zp.402D9E		
022620E6	C3	ret		
022620E7	D4 68	aam 68		
022620F9	14 0F	adc al,F		

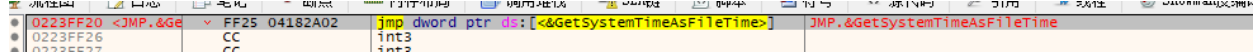
## 修复

# IAT混淆修复

了解了IAT的保护方式，修复思路就很简单了。

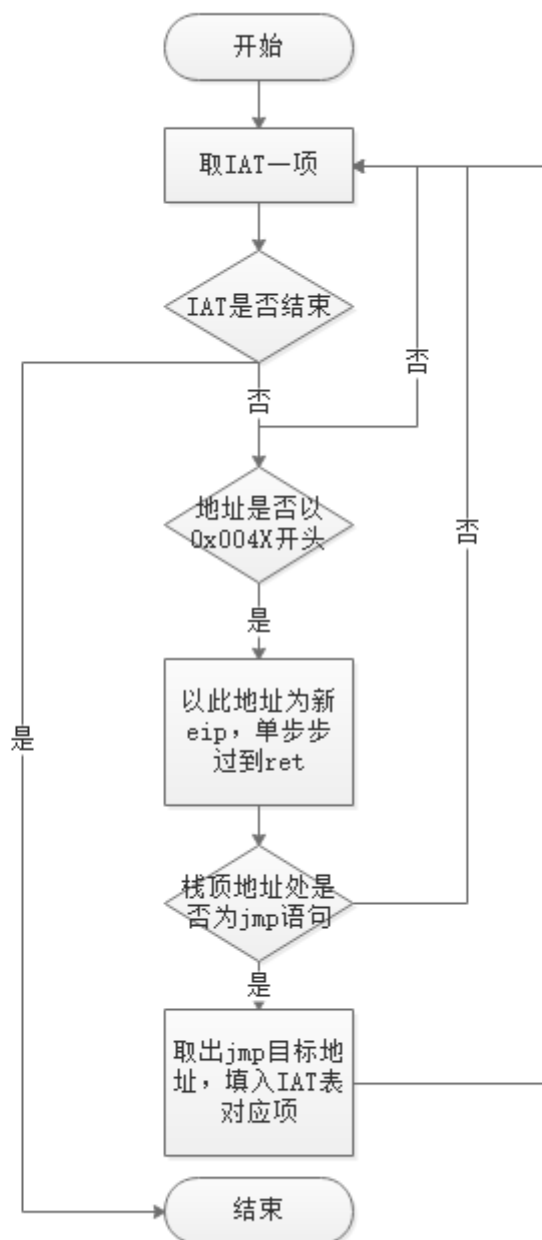
## 第一种方式修复

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。

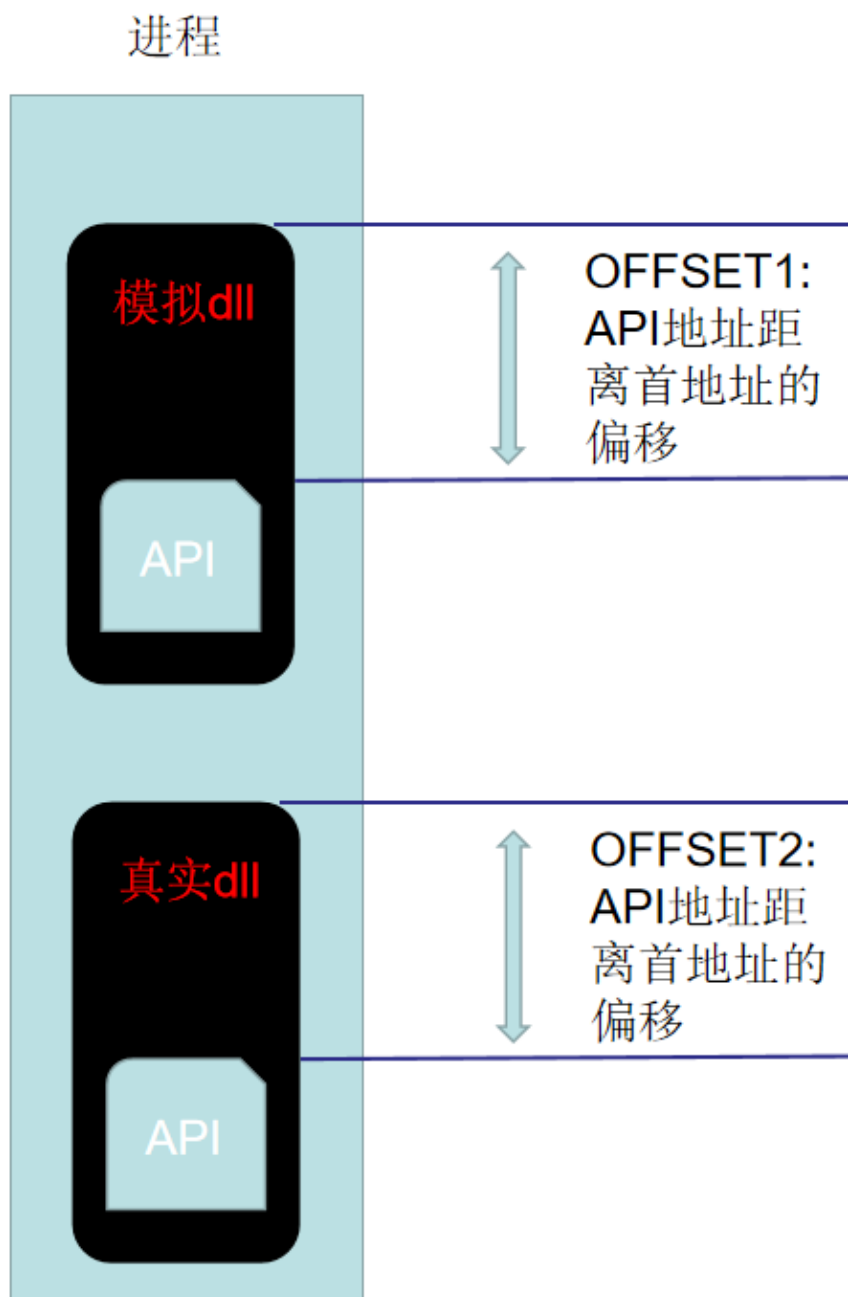
所以，我这里选择写个x64dbg的python脚本，脚本会扫描IAT表，取出IAT表中以0x004XX开头的每一项，并以其为新eip，然后单步运行到ret，取出栈顶地址，并判断是否为jmp语句，如果是则将jmp的目标地址即API地址填入对应的IAT表中。





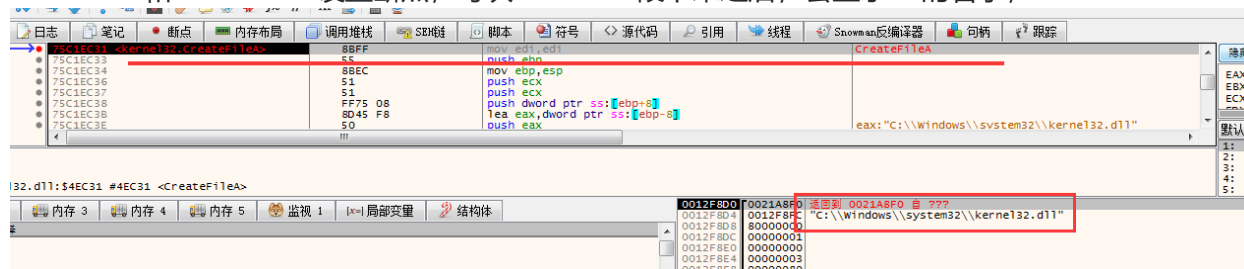
## 第二种方式修复

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)
symbolList = vectorGetSymbolInfoList(symbolInfoList)
for i in symbolList:
    #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))

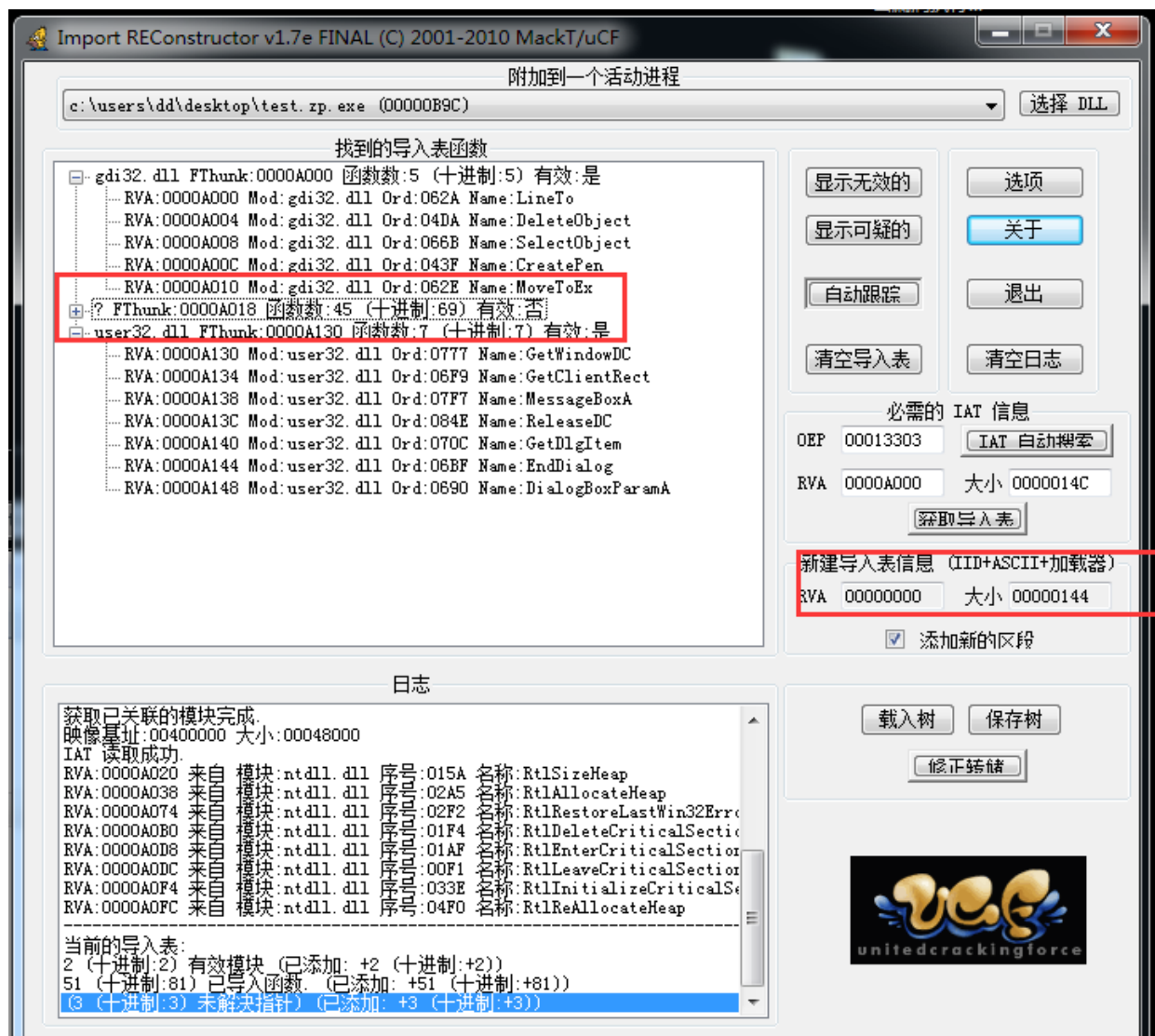
    codeOfBegin = ReadByte(espValue)
    print("Byte: %02x " % codeOfBegin)
    if codeOfBegin == 0xff:
        apiAddr = ReadDword(espValue + 2)
        print("JumpAddr: %08x ApiAddr: %08x" % (espValue + 2, apiAddr))
        apiAddr = ReadDword(apiAddr)
        print("apiAddr: %08x ApiAddr: %08x" % (espValue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

    addrToHandle = addrToHandle + 4

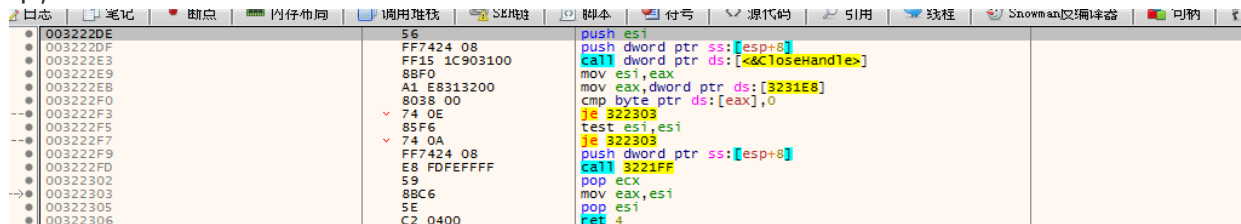
```

Address	Disassembly	Comment
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	75C1F6C8	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	75C12DD0	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.GetCPIInfo
0040A0E4	75C1DADB	kernel32.GetACP
0040A0E8	75C1461A	kernel32.GetOEMCP
0040A0EC	75C1C446	kernel32.Sleep

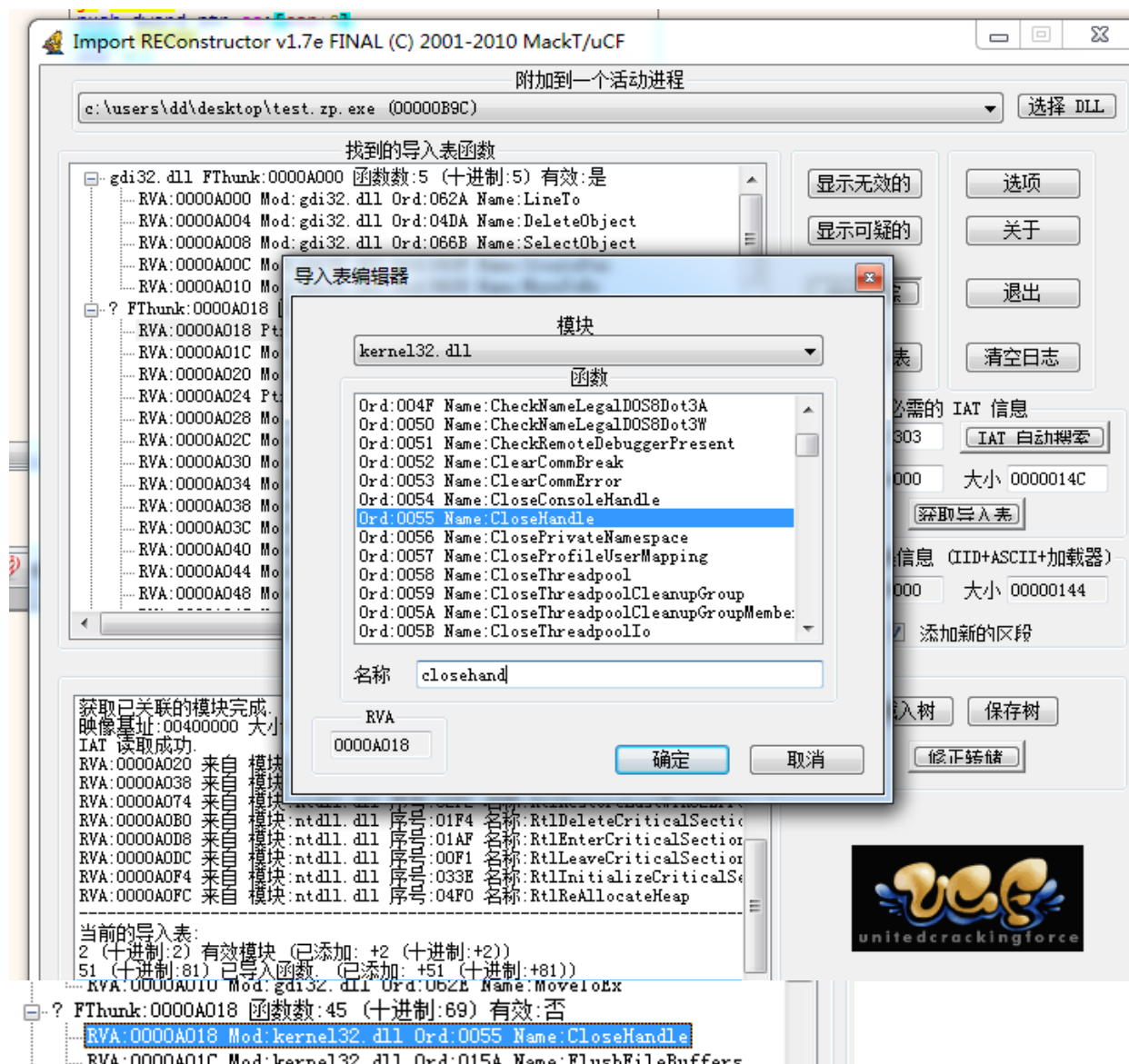
代码跑完之后，修复的IAT可见上图。



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，



## 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。

这里的修复思路是把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
    */

```

```

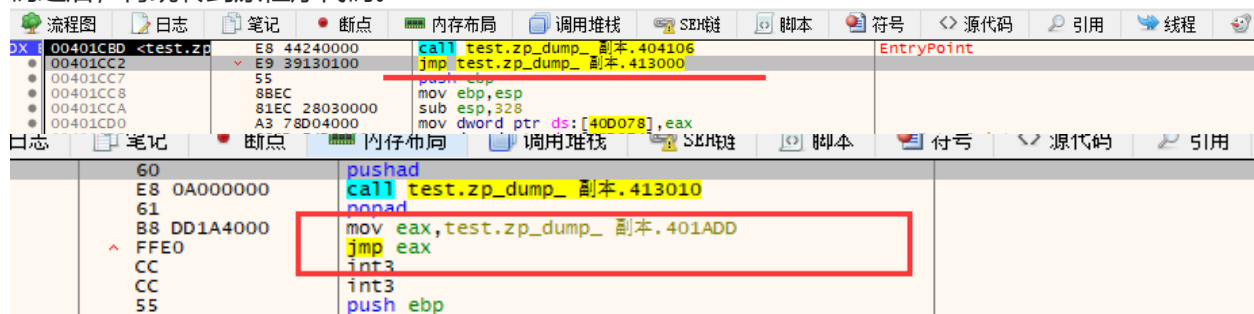
for (int i = 0; i < sizeof(dis) / sizeof(dis[0]); ++i)
{
    HANDLE hFile = pfnCreateFileA((LPCTSTR)dis[i].m_szName, GENERIC_READ |
    GENERIC_WRITE, FILE_SHARE_READ, NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

    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。

