



实验4-2

PE导入表机制实验

利用Patch指令的方式实现EPO

- 实验三中我们尝试替换入口点5字节为JMP，获得执行权
- 杀毒软件检测入口点的指令就能识别出来这种替换
- 若知道一条指令起始位置，用JMP替换？有一些问题：
 - 1) 你怎么知道这是一条指令的？
 - 2) 你怎么知道它从哪来开始？
- 如果把数据区当成指令，EIP从不会指向那里，病毒也不可能执行。
- 如果从指令的中间开始Patch（修改），EIP也不会指向那里，而且还会毁坏指令，让程序飞了。

最简单的函数调用指令Patch

- 普通函数调用指令：call指令，机器码E8 xx xx xx xx
- 仅仅查找E8来进行指令Patch？
 - 会有太多匹配，还可能是数据
 - 某些call不一定会被执行
- 结论：选择执行可能性高的指令来进行Patch

Patch指令的核心问题与解决思路

■ 核心问题：

1. 找到一条指令，并知道其起始边界
2. 保证指令必然执行（否则patch后病毒代码也不会必然执行）

■ 解决思路：

- 分析一些正常程序常用的**API函数调用语句**，
- 如kernel32.dll中的WriteFile函数
- 对导入函数调用指令进行Patch
- 这些指令的可靠度很高
- 可以确保大概率被执行

- 现在，我们来学习**导入函数**的执行机制：
- 通过导入表相关机制，了解间接调用指令，然后进行Patch，从而获得执行机会
- 基本上，恶意代码获得执行的机会都是利用某种间接性，比如这里的**导入表相关机制**

实验的目的是要了解Windows下的导入表机制

程序如何调用某个DLL的导出API

- 函数调用一般采用Call + 相对偏移 (**直接跳转**)
- 机器码为: E8 xx xx xx xx, 其中, xx xx xx xx是目标地址相对Call指令后面那条指令的偏移 (类似JMP指令)

```
15:    fun(1, 2);  
00301228 6A 02          push    2  
0030122A 6A 01          push    1  
0030122C E8 F2 FD FF FF  call    @ILT+30(fun) (00301023)
```

在机器码中, E8后为F2 FD FF FF

如果是绝对地址, 这是一个高地址的内核空间, 这基本是不可能的

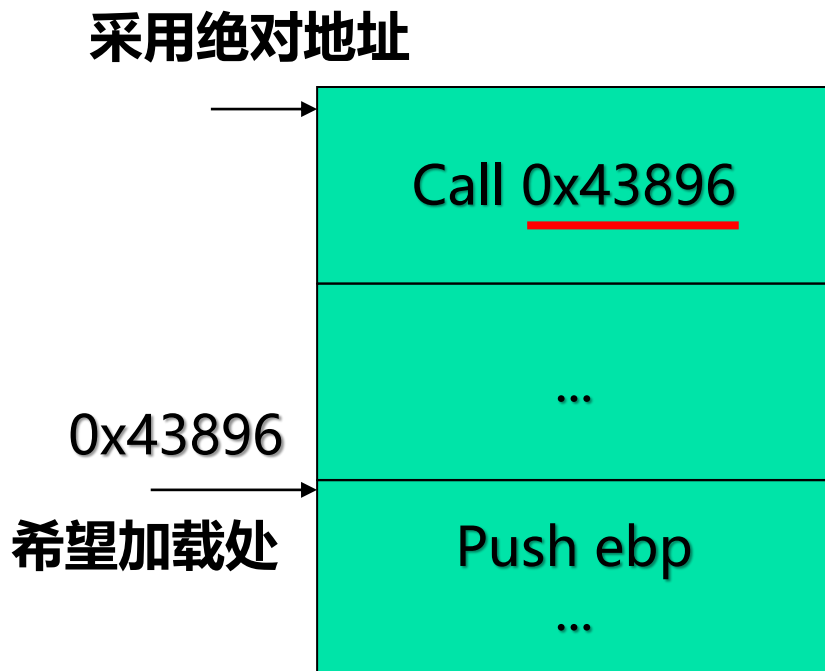
因此, 猜测是值为负数的偏移量说明调用的fun函数在前面

似乎Call指令调用的绝对地址00301023, 但这是VC工具帮我们解析出来的结果, 再来看看机器码

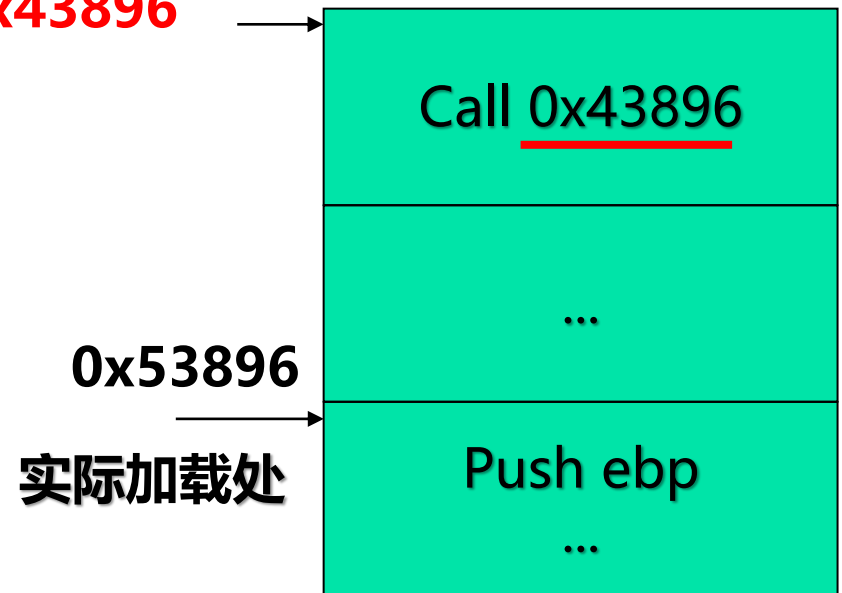
思考，为什么Call要用相对偏移

- 在前面，我们已经知道了加载可能出现与约定地址不吻合的问题如果如果Call指令采用了绝对地址，那么这条Call指令就会失效。

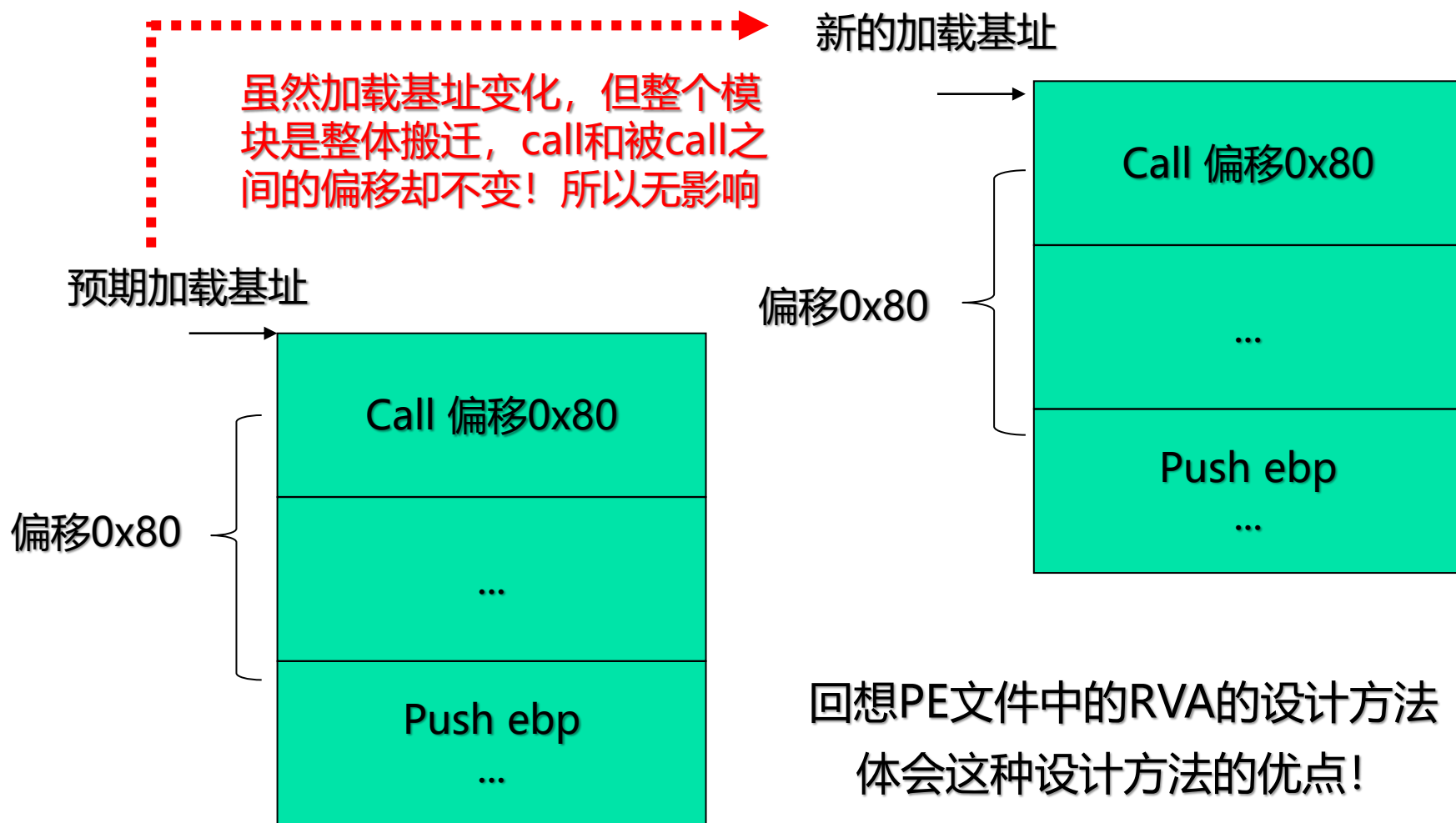
实际加载处



目标指令的新地址是0x53896
而call指令中地址还是
0x43896

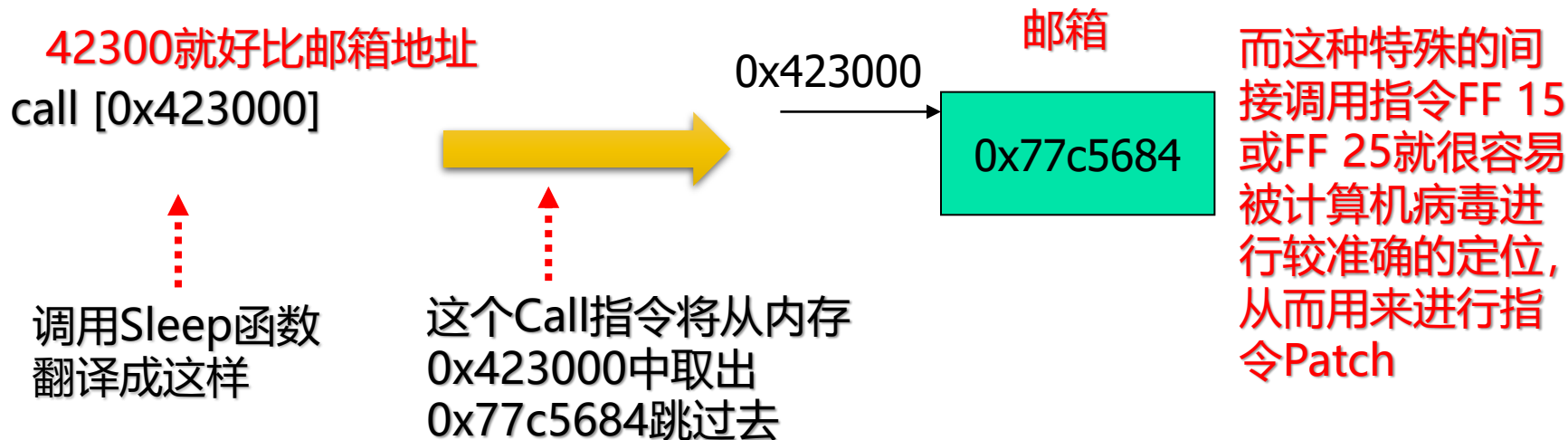


但是，采用相对偏移Call就不会出问题



相对偏移的局限

- Call 相对偏移的优点只能运用到同一个模块中 (DLL, EXE), 如果跨越了模块, 调用DLL中的函数时, 并不知道该模块会加载到哪里, 因此, 是不可能事先算出转跳偏移的, 那怎么办呢?
- 这是在编译器和系统协助下, 利用**导入表机制**完成了这个工作, 0x423000属于**调用模块**的导入表, 在调用模块的地址空间, 里面放的内容是**被调用模块**中的函数地址, 这就是间接调用的思想
- 简单说, 就像为每个被调用的DLL函数都设定一个邮箱, 系统加载了DLL后, 利用**导出表机制**获得函数的地址 (这个我们前面已经学习过了), 然后将函数地址放到各自的邮箱中, 这样, 需要调某函数时, 就从邮箱里拿到这个地址



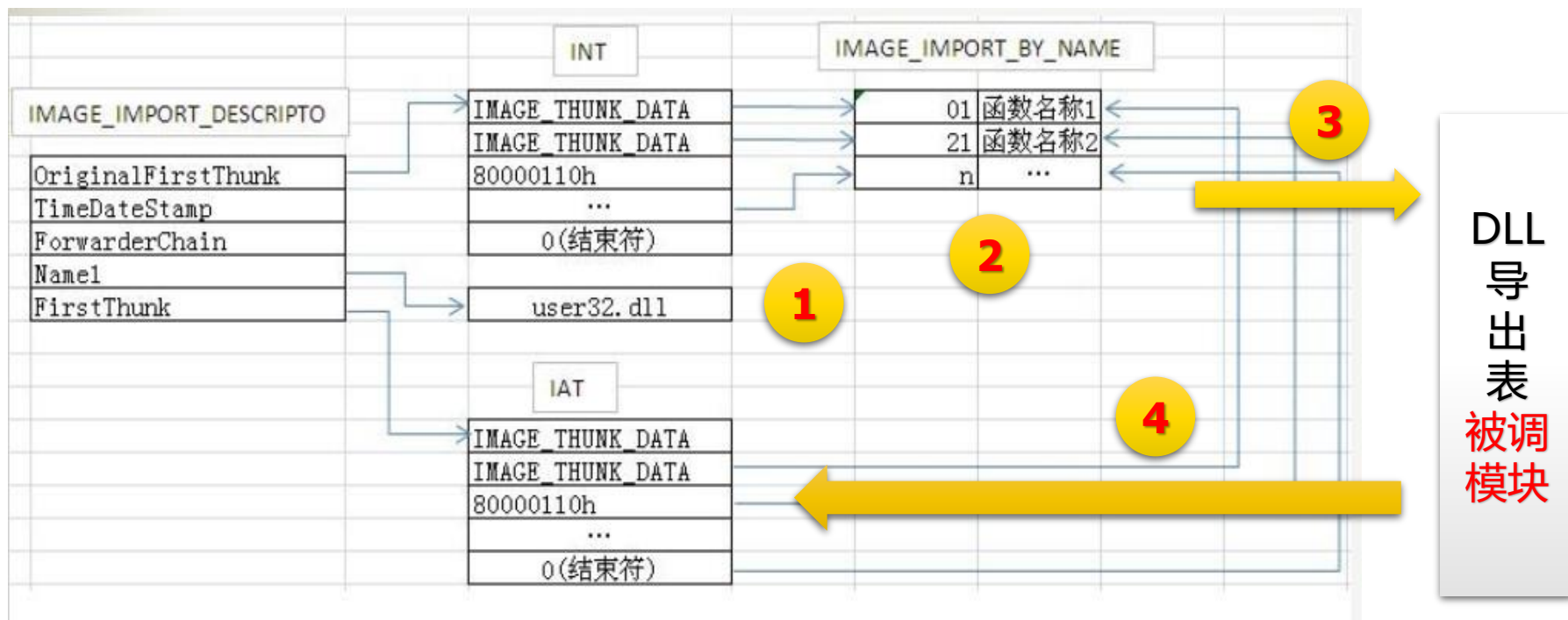
实验

分析Windows下的导入表机制

系统加载时调用模块的导入表初始化算法

IAT初始化算法:

- 1 从导入表目录获取IMAGE_IMPORT_DESCRIPTOR表入口，获取该表的当前项，**每项代表一个被引用的DLL**，从其中DLL名RVA段可获取DLL名，下面就引入此DLL
- 2 系统遍历**编译时生成的**每个IAT（如果有INT则判断INT，当预先绑定时，只能遍历INT），比如第2项，其中存储的值是IMAGE_IMPORT_BY_NAME中**对应项的RVA**
- 3 找到该项，获得其函数名串，以\0结尾，通过对应**DLL导出表**找到相关函数的加载地址
- 4 **然后将其放入IAT第2项**（此时，IAT表项的值才变为了函数的加载地址）
- 5 如此遍历IAT，将所有项都填入对应函数入口地址
- 6 遍历IMAGE_IMPORT_DESCRIPTOR表，对所有DLL都做2~4步处理。



1. 观察加载前的导入表情况

在PE文件的可选头的数据目录中找到导入表的RVA

cloudmusic.exe				
IMAGE_DOS_HEADER		RVA	Data	Description
MS-DOS Stub Program		00000184	00000070	Size
IMAGE_NT_HEADERS		00000188	00066740	RVA
Signature		0000018C	0000008C	Size
IMAGE_FILE_HEADER		00000190	0006F000	RVA
IMAGE_OPTIONAL_HEADER		00000194	00011168	Size
IMAGE_SECTION_HEADER .text		00000198	00000000	RVA
		0000019C	00000000	Size

找到该导入表所在节，本例为.rdata节

cloudmusic.exe				
IMAGE_DOS_HEADER		RVA	Data	Description
MS-DOS Stub Program		00000228	2E 72 64 61	Name
IMAGE_NT_HEADERS		0000022C	74 61 00 00	
Signature		00000230	0001E7BA	Virtual Size
IMAGE_FILE_HEADER		00000234	00049000	RVA
IMAGE_OPTIONAL_HEADER		00000238	0001E800	Size of Raw Data
IMAGE_SECTION_HEADER .text		0000023C	00047800	Pointer to Raw Data
IMAGE_SECTION_HEADER .rdata		00000240	00000000	Pointer to Relocations
IMAGE_SECTION_HEADER .data		00000244	00000000	Pointer to Line Numbers
IMAGE_SECTION_HEADER .tls		00000248	0000	Number of Relocations
IMAGE_SECTION_HEADER .rsrc		0000024A	0000	Number of Line Numbers
		0000024C	40000040	Characteristics

2. 观察INT和IAT表

IMAGE_SECTION_HEADER .tls
IMAGE_SECTION_HEADER .rsrc
IMAGE_SECTION_HEADER .reloc
SECTION .text
SECTION .rdata
 IMPORT Address Table
 IMAGE_DEBUG_DIRECTORY
 DELAY_IMPORT_DLL_Names
 IMAGE_LOAD_CONFIG_DIRECTORY
 IMAGE_TLS_DIRECTORY
 IMAGE_DEBUG_TYPE_CODEVIEW
 IMAGE_DEBUG_TYPE_
 DELAY_IMPORT_Descriptors

RVA	Data	Description	Value
00049000	000674C8	Hint/Name RVA	02A6 SetEntriesInAcW
00049004	000674AE	Hint/Name RVA	006C ConvertSidToStringSidW
00049008	0006749C	Hint/Name RVA	02C1 SetThreadToken
0004900C	00067484	Hint/Name RVA	007C CreateProcessAsUserW
00049010	0006746C	Hint/Name RVA	0197 LookupPrivilegeValueW
00049014	00067456	Hint/Name RVA	015A GetTokenInformation
00049018	0006744A	Hint/Name RVA	0107 EqualSid
0004901C	00067436	Hint/Name RVA	00DF DuplicateTokenEx
00049020	00067424	Hint/Name RVA	00DE DuplicateToken
00049024	0006740C	Hint/Name RVA	007F CreateRestrictedToken
00049028	000673F6	Hint/Name RVA	0083 CreateWellKnownSid
0004902C	000673EC	Hint/Name RVA	0076 CopySid

IAT

内容一致

SECTION .text
SECTION .rdata
 IMPORT Address Table
 IMAGE_DEBUG_DIRECTORY
 DELAY_IMPORT_DLL_Names
 IMAGE_LOAD_CONFIG_DIRECTORY
 IMAGE_TLS_DIRECTORY
 IMAGE_DEBUG_TYPE_CODEVIEW
 IMAGE_DEBUG_TYPE_
 DELAY_IMPORT_Descriptors
 DELAY_IMPORT_Name_Table
 DELAY_IMPORT_Hints/Names
 IMAGE_EXPORT_DIRECTORY
 EXPORT Address Table
 EXPORT Name Pointer Table
 EXPORT Ordinal Table
 EXPORT Names
 IMPORT Directory Table
 IMPORT Name Table

RVA	Data	Description	Value
000667C0	000674C8	Hint/Name RVA	02A6 SetEntriesInAcW
000667D0	000674AE	Hint/Name RVA	006C ConvertSidToStringSidW
000667D4	0006749C	Hint/Name RVA	02C1 SetThreadToken
000667D8	00067484	Hint/Name RVA	007C CreateProcessAsUserW
000667DC	0006746C	Hint/Name RVA	0197 LookupPrivilegeValueW
000667E0	00067456	Hint/Name RVA	015A GetTokenInformation
000667E4	0006744A	Hint/Name RVA	0107 EqualSid
000667E8	00067436	Hint/Name RVA	00DF DuplicateTokenEx
000667EC	00067424	Hint/Name RVA	00DE DuplicateToken
000667F0	0006740C	Hint/Name RVA	007F CreateRestrictedToken
000667F4	000673F6	Hint/Name RVA	0083 CreateWellKnownSid
000667F8	000673EC	Hint/Name RVA	0076 CopySid
000667FC	000673DC	Hint/Name RVA	0261 RegOpenKeyExW
00066800	000673C0	Hint/Name RVA	0249 RegDisablePredefinedCache
00066804	000673B0	Hint/Name RVA	0290 RevertToSelf
00066808	0006739E	Hint/Name RVA	0239 RegCreateKeyExW
0006680C	00067390	Hint/Name RVA	0230 RegCloseKey
00066810	0006737C	Hint/Name RVA	02F1 SystemFunction036

INT

3. 看看表中的RVA是否指向了函数名

IMAGE_SECTION_HEADER .tls	^	RVA	Data	Description	Value
IMAGE_SECTION_HEADER .rsrc		00049000	000674C8	Hint/Name RVA	02A6 SetEntriesInAclW
IMAGE_SECTION_HEADER .reloc		00049004	000674AE	Hint/Name RVA	006C ConvertSidToStringSidW
SECTION .text		00049008	0006749C	Hint/Name RVA	02C1 SetThreadToken
SECTION .rdata		0004900C	00067484	Hint/Name RVA	007C CreateProcessAsUserW
IMPORT Address Table		00049010	0006746C	Hint/Name RVA	0197 LookupPrivilegeValueW
IMAGE_DEBUG_DIRECTORY		00049014	00067456	Hint/Name RVA	015A GetTokenInformation
DELAY_IMPORT_DLL_Names		00049018	0006744A	Hint/Name RVA	0107 EqualSid
IMAGE_LOAD_CONFIG_DIRECTORY		0004901C	00067436	Hint/Name RVA	00DF DuplicateTokenEx
IMAGE_TLS_DIRECTORY		00049020	00067424	Hint/Name RVA	00DE DuplicateToken
IMAGE_DEBUG_TYPE_CODEVIEW		00049024	0006740C	Hint/Name RVA	007F CreateRestrictedToken
IMAGE_DEBUG_TYPE_		00049028	000673F6	Hint/Name RVA	0083 CreateWellKnownSid
DELAY_IMPORT_Descriptors		0004902C	000673EC	Hint/Name RVA	0076 CopySid

找到学号末尾对应的项，例如第一个项000674C8，换算文件偏移000674C8 – 49000 + 47800=65CC8

IMAGE_NT_HEADERS	00000230	0001E7BA	Virtual Size
Signature	00000234	00049000	RVA
IMAGE_FILE_HEADER	00000238	0001E800	Size of Raw Data
IMAGE_OPTIONAL_HEADER	0000023C	00047800	Pointer to Raw Data
IMAGE_SECTION_HEADER .text	00000240	00000000	Pointer to Relocations
IMAGE_SECTION_HEADER .rdata	00000244	00000000	Pointer to Line Numbers
IMAGE_SECTION_HEADER .data	00000248	0000	Number of Relocations

用UE/C32ASM定位到文件65CC8的位置，可见其确实指向了相应的函数名（序号+函数名）

```
00065cc0h: 6E 67 53 69 64 57 00 00 A6 02 53 65 74 45 6E 74 ; ngSidW...SetEnt
00065cd0h: 72 69 65 73 49 6E 41 63 6C 57 00 00 4E 01 47 65 ; riesInAclW...N.Ge
00065ce0h: 74 53 65 63 75 72 69 74 79 49 6E 66 6F 00 41 44 ; tSecurityInfo.AD
00065cf0h: 56 41 50 49 33 32 2E 64 6C 6C 00 00 C3 00 53 48 ; VAPI32.dll...?SH
```


4. 看看加载后INT表

现在我们看见加载前INT和IAT的内容是一致的，这是编译器完成的
那么加载后IAT的内容是什么呢？这个是操作系统的加载器完成的，**我们需要去看看加载在内存中的IAT表**

IMAGE_NT_HEADERS	00000123	00	Minor Linker Version
Signature	00000124	00047400	Size of Code
IMAGE_FILE_HEADER	00000128	00038600	Size of Initialized Data
IMAGE_OPTIONAL_HEADER	0000012C	00000000	Size of Uninitialized Data
IMAGE_SECTION_HEADER .text	00000130	0002D9EE	Address of Entry Point
IMAGE_SECTION_HEADER .rdata	00000134	00001000	Base of Code
IMAGE_SECTION_HEADER .data	00000138	00049000	Base of Data
IMAGE_SECTION_HEADER .tls	0000013C	00400000	Image Base
IMAGE_SECTION_HEADER .rsrc	00000140	00001000	Section Alignment

首先，PE文件的ImageBase告诉我们预加载的地址是00400000
但是，实际的加载地址呢？我们可以用OD来查看一下

4. 看看加载后INT表

我们用OD来打开可执行程序，查看其入口点的实际地址为00BBD9EE

=实际基址 + 入口点RVA

CPU - 主线程, 模块 - cloudmus		
地址	HEX 数据	反汇编
00BBD9EE	\$ E8 299B0000	CALL cloudmus.00BC751C
00BBD9F3	E9 7FFEFFFF	JMP cloudmus.00BBD877
00BBD9F8	CC	INT3
00BBD9F9	CC	INT3
00BBD9FA	CC	INT3
00BBD9FB	CC	INT3
00BBD9FC	CC	INT3
00BBD9FD	CC	INT3
00BBD9FE	CC	INT3
00BBD9FF	CC	INT3
00BBDAA0	\$ 57	PUSH EDI
00BBDAA1	56	PUSH ESI

00BBD9EE – 0002D9EE =
00B90000

可以看见，实际加载的基址和我们预期基址是不同的

因此，用此值和PE文件的AddressOfEntryPoint相减，我们就能知道实际的加载基址

cloudmusic.exe
IMAGE_DOS_HEADER
MS-DOS Stub Program
IMAGE_NT_HEADERS
Signature
IMAGE_FILE_HEADER
IMAGE_OPTIONAL_HEADER
IMAGE_SECTION_HEADER .text

	RVA	Data	Description
	00000120	010B	Magic
	00000122	0C	Major Linker Version
	00000123	00	Minor Linker Version
	00000124	00047400	Size of Code
	00000128	00038600	Size of Initialized Data
	0000012C	00000000	Size of Uninitialized Data
	00000130	0002D9EE	Address of Entry Point

4. 看看加载后IAT表

接下来，我们就要去内存里面看看加载后的IAT了

IMAGE_SECTION_HEADER .rsrc
IMAGE_SECTION_HEADER .reloc
SECTION .text
SECTION .rdata
 IMPORT Address Table
IMAGE_DEBUG_DIRECTORY
DELAY_IMPORT_DLL_NAMES
IMAGE_LOAD_CONFIG_DIRECTORY

RVA	Data	Description	Value
00049000	000674C8	Hint/Name RVA	02A6 SetEntriesInAclW
00049004	000674AE	Hint/Name RVA	006C ConvertSidToStringSidW
00049008	0006749C	Hint/Name RVA	02C1 SetThreadToken
0004900C	00067484	Hint/Name RVA	007C CreateProcessAsUserW
00049010	0006746C	Hint/Name RVA	0197 LookupPrivilegeValueW
00049014	00067456	Hint/Name RVA	015A GetTokenInformation
00049018	0006744A	Hint/Name RVA	0107 EqualSid

在前面，我们已经知道IAT表第一项的RVA是00049000

那么它现在在内存中的VA就应该 = 实际基址 + RVA = B90000 + 49000 = BD9000

我们在OD中用内存窗体Ctrl+G定位到地址0x00BD9000，查看相应的内容

地址	HEX 数据	ASCII
00BD9000	70 FE ED 75 A0 D9 ED 75 B0 E9 ED 75 10 FD ED 75	p 清
00BD9010	10 D4 ED 75 70 DB ED 75 50 FC ED 75 D0 F9 ED 75	□ 皂 up
00BD9020	70 F0 ED 75 80 FF ED 75 D0 FC ED 75 00 E2 ED 75	p 疼 u €
00BD9030	80 DE ED 75 50 FE ED 75 90 F0 ED 75 20 E2 ED 75	€ 偷 u P
00BD9040	50 E1 ED 75 20 2A 74 74 00 BB ED 75 10 CD ED 75	P 汁 u * t
00BD9050	F0 F5 ED 75 70 3B EE 75 30 FF ED 75 30 3B EE 75	屎 群 p ; u
00BD9060	70 E1 ED 75 D0 40 EE 75 F0 30 EF 75 70 DF ED 75	p 汁 u 羴
00BD9070	F0 E4 ED 75 00 00 00 00 D0 3C 4F 75 E0 97 4E 75	炸 群 ...
00BD9080	60 1A 4E 75 50 97 4E 75 20 F5 4E 75 C0 98 4E 75	□ Nu P
00BD9090	00 4F 52 75 90 4B 52 75 20 F4 4E 75 30 31 4F 75	. ORu 低 F
00BD90A0	00 F1 4E 75 30 47 4E 75 20 43 4F 75 D0 EF 4E 75	. 温 u O G
00BD90B0	00 3C 4F 75 80 A4 4E 75 40 7D 4E 75 10 90 4E 75	. < Ou €
00BD90C0	30 F1 4E 75 30 4B 50 75 20 8F 4E 75 40 89 4E 75	O 温 u O K F
00BD90D0	D0 42 52 75 00 41 4F 75 30 93 4E 75 80 3C 4F 75	蟻 Ru. AC

可以看见，这就是我们加载后的IAT表内容，已经和PE文件中IAT表的内容不同了！！

那么它存放的是不是函数的入口地址呢？我们来验证一下


5. 验证加载后IAT表



RVA	Data	Description	Value
00049000	000674C8	Hint/Name RVA	02A6 SetEntriesInAclW
00049004	000674AE	Hint/Name RVA	006C ConvertSidToStringSidW
00049008	0006749C	Hint/Name RVA	02C1 SetThreadToken
0004900C	00067484	Hint/Name RVA	007C CreateProcessAsUserW

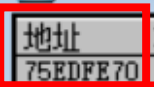
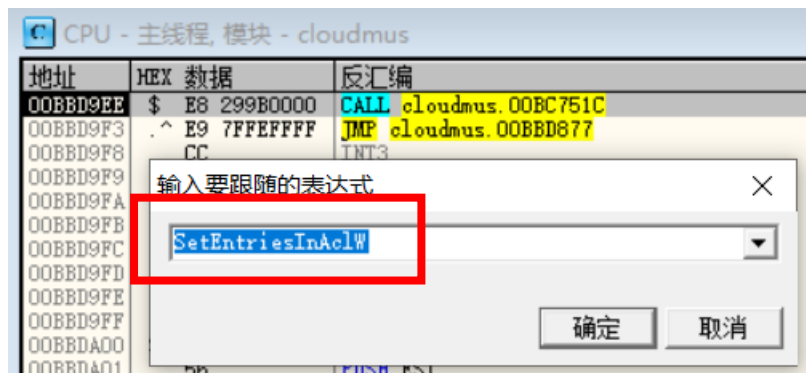
如果加载后的IAT表是函数地址：

那么前面4个字节 70 FE ED 75，就应该是对应函数SetEntriesInAclW的入口地址



地址	HEX 数据
00BD9000	70 FE ED 75 AO D9 ED 75 BO E9 ED 75 10 FD ED 75
00BD9010	10 D4 ED 75 70 DB ED 75 50 FC ED 75 D0 F9 ED 75
00BD9020	70 FO ED 75 80 FF ED 75 D0 FC ED 75 00 E2 ED 75
00BD9030	80 DB ED 75 50 FC ED 75 D0 FC ED 75 00 E2 ED 75

在OD的CPU窗口，按Ctrl+G输入函数名，可以看见函数的入口地址确实就是加载后IAT表的内容！（75 ED FE 70小端机填充方式）



地址	HEX 数据	反汇编
75EDFE70	8BFF	MOV EDI, EDI
75EDFE72	55	PUSH EBP
75EDFE73	8BEC	MOV EBP, ESP
75EDFE75	5D	POP EBP
75EDFE76	FF25 F421F375	JMP DWORD PTR DS:[75F321F4]
75EDFE7C	CC	INT3
75EDFE7D	CC	INT3
75EDFE7E	CC	INT3
75EDFE7F	CC	INT3
75EDFE80	CC	INT3

实验要求

选择一个可执行文件完成以上实验

建议选用notepad.exe或control.exe

要求：

1. 在PE文件找到加载前IAT表
2. 验证加载前IAT表存放的是指向函数名的RVA
3. 在内存中找到加载后IAT表
4. 验证加载后IAT表的内容为对应函数的入口地址