

第五十四章-ExeCryptor v2.2.50.a-Part2

本章我们将编写一个脚本来修复 ExeCryptor 的 IAT。可能很多童鞋一听到脚本这个词就有一种头皮发麻的感觉。(嘿嘿),因为一般来说人家编写好的脚本通常都比较长,看起来非常复杂的样子,当然会感觉到头皮发麻撒。但是我这里换一种方法,我不是直接拿一个现成的脚本给大家,而是带着大家从零开始编写这个脚本,逐步添砖加瓦,这样大家接受起来就容易的多。

我的想法很简单,就是遍历 IAT,看哪一个 IAT 项被重定向了,如果被重定向了的话,就修复之,我们首先来构建这个脚本的基本框架。首先我们要定义一个变量,用它来作为 IAT 项的指针。

var table

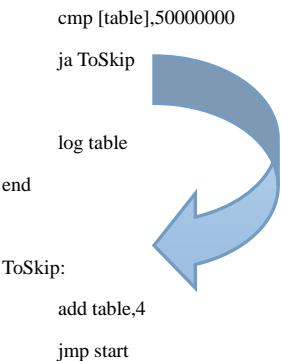
首先我们将 IAT 的起始地址保存到该 table 变量中

mov table,460818

这里我们就将 table 这个变量初始化完毕了,接下来我们需要遍历整个 IAT,判断其中哪些项被重定向了,如果被重定向了,就修复之。如果没有被重定向,就继续遍历下一项。

脚本的基本框架就是这样的:

start:



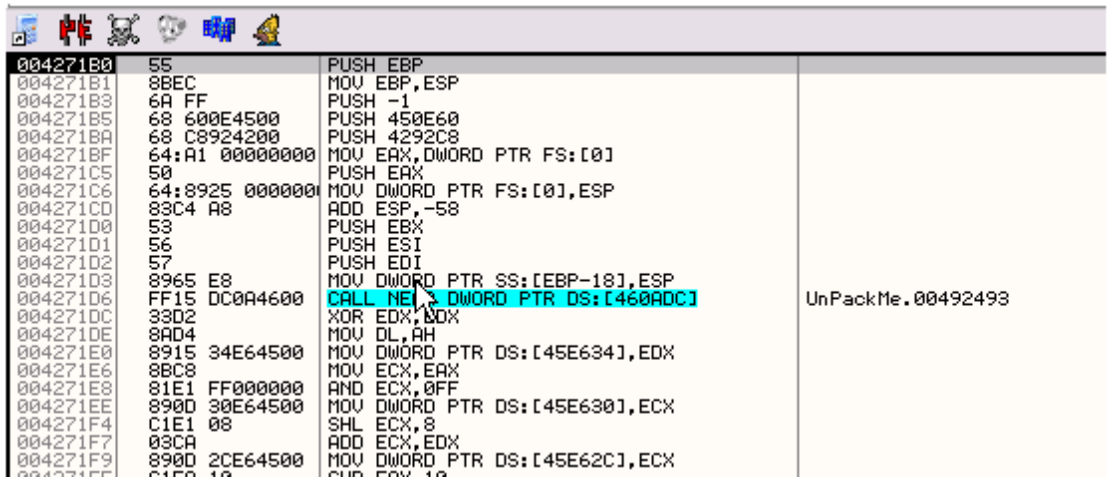
这是一个基本框架-遍历 IAT 中的每一项,依次判断 IAT 项中的值是否大于 50000000(一般来说大于 50000000 的话,就属于是 DLL 中的地址,即为正常的 API 函数地址,没有被重定向)如果大于 50000000 的话,就直接跳过该项,继续遍历下一项。

如果是小于等于 50000000 的话,说明是被重定向过的,则进行下面的操作:

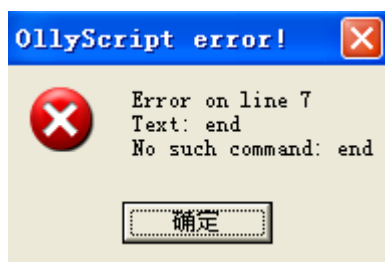
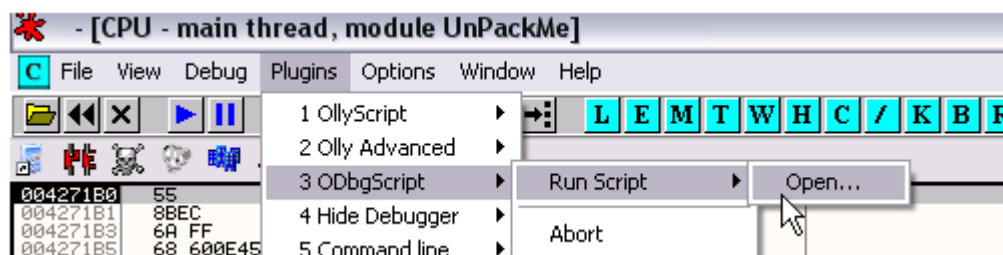
log table

end

这里的话我们就需要对重定向的项进行修复了,要对其修复的话,首先我们要知道其实际要调用的 API 函数是什么,这里为了简单起见,我们只测试第一个待修复的项,所以在定位到实际要调用的 API 函数以后,直接退出循环。



我们首先断到 OEP 处(如何断在 OEP 处,上一章已经给大家介绍过了),然后执行该脚本看看效果。



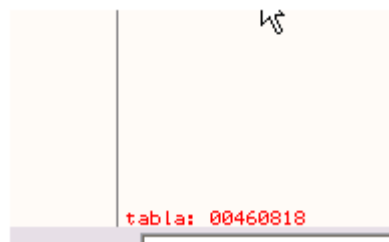
我们可以看到提示第 7 行有一个不识别的 end 命令,呵呵,写错了,应该是 ret 命令才对,我们将它改过来。

```
var table
    mov table,460818
start:
    cmp [table],50000000
    ja ToSkip
    log table
    ret
ToSkip:
    add table,4
    jmp start
```

这里我们已经改过来了,再次执行该脚本看看效果。



我们可以看到这次没有报错,并且第一个重定向的 IAT 项的地址成功被记录到日志中了。



现在我们需要获取重定向的 IAT 项的值了,所以这里我们再定义 content 变量用于临时保存重定向的 IAT 项的值。

```
var table
```

```
var content

mov table,460818

start:

cmp [table],50000000

ja ToSkip:

log table

mov content,[table]

log content

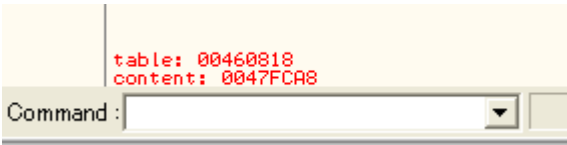
ret

ToSkip:

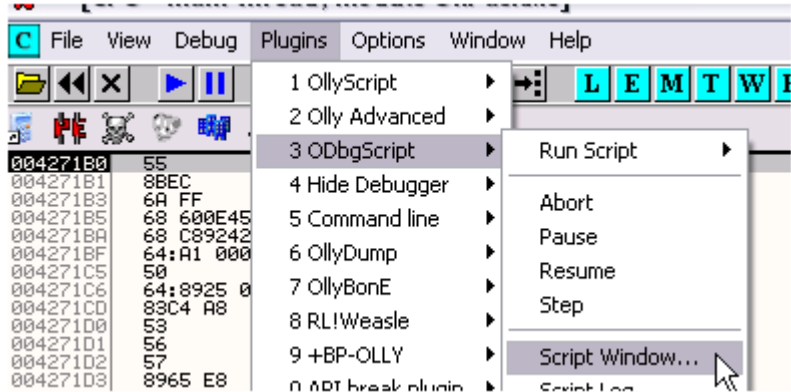
add table,4

jmp start
```

这里添加了 content 这个变量以后,我们就可以把重定向过的 IAT 项的地址以及值都记录到日志中了。



我们可以看到执行完该脚本以后,日志中记录了 460818 这个 IAT 项中的值为 47FCA8,接下来我们需要将 ret 命令去掉,循环遍历所有项。不知道大家知不知道其实 ODbgScript 这个插件是可以对脚本进行单步跟踪的,以便脚本出错的时候方便我们调试。我们来看一看这个功能如何使用:



我们选中菜单项 Plugins-ODbgScript-Script Window,打开脚本跟踪窗口。

Script Execution				
Line	Command	Result	EIP	Values <---
	C:\Script.txt			
1	var table			
2	var content			
3	mov table,460818			
4	start:			
5	cmp [table],50000000			
6	ja ToSkip			
7	log table			
8	mov content,[table]			
9	log content			
10	ret			
11	ToSkip:			
12	add table,4			
13	jmp start			

这里我们可以看到单步跟踪的快捷键为 S,给指定行设置断点的快捷键为 F2,有了这两个命令就足够我们对脚本进行跟踪排错了。

下面我们按 S 键单步跟踪看看效果如何。

Script Execution				
Line	Command	Result	EIP	Values <---
1	C:\Script.txt			
2	var table		004271B0	
3	var content		004271B0	
4	mov table,460818			
5	start:			
6	cmp [table],50000000		004271B0	47FCA8?60818
7	ja ToSkip		004271B0	
8	log table			
9	mov content,[table]			
10	log content			
11	ret			
12	ToSkip:			
13	add table,4			
14	jmp start			

这里我们可以看到 EIP 这一列会显示当前行 EIP 的值为多少,Values <---这一列的话,如果当前行有取值操作的话,就会以 值《内存地址 的方式显示出来,例如:这里显示的是 47FCAB << 460818。我们也可以通过给指定行设置断点,然后单击 Resume 菜单项来执行多条命令,然后停在断点处。好了,现在我们关闭这个脚本窗口,继续给我们的脚本添加新的内容。

现在我们需要给它添加一个条件,当满足该条件时就停止执行该脚本并退出,这里该条件应该是遍历完整个 IAT,即超出了 IAT 的范围,table 指针大于 460F28 时就退出。

```
var table
var content
    mov table,460818
start:
    cmp table,460F28
    ja final
    cmp [table],50000000
    ja ToSkip
    log table
    mov content,[table]
    log content
    ret
ToSkip:
    add table,4
    jmp start
final:
    ret
```

这里判断的条件我们就添加好了,当遍历到 IAT 尾部的时候,我们就跳转到 final 标签处,结束脚本的执行。

下面我们就可以来修复重定向的 IAT 项了,前面我们在定位到第一个重定向的 IAT 项时,是将其地址和值都记录到日志中,然后 ret 退出脚本,这里我们将这个 ret 命令去掉。

```
var table
var content
    mov table,460818
start:
    cmp table,460F28
    ja final
    cmp [table],50000000
```

```

ja ToSkip
log table
mov content,[table]
log content
ToSkip:
add table,4
jmp start
final:
ret

```

这里我们就将 log content 后面 ret 命令去掉了,这样处理之后,在记录完重定向 IAT 项的地址以及值以后,会到达 ToSkip 标签处继续遍历后面的 IAT 项,重复上面的过程,直到遍历完整个 IAT 为止,我们来看看该脚本执行的效果。



我们来看看日志信息。

Address	Message
	table: 00460818
	content: 0047FCA8
	table: 0046081C ASCII "AoH"
	content: 00486F41
	table: 00460820
	content: 00480098
	table: 00460824
	content: 0047A4F9
	table: 00460828
	content: 0048D981
	table: 0046082C
	content: 00000000
	table: 00460838
	content: 00000000
	table: 00460970
	content: 00000000
	table: 00460974
	content: 004889B6
	table: 00460978 ASCII " H"
	content: 004858A9
	table: 0046097C
	content: 0048D82F
	table: 00460980
	content: 0046C28C
	table: 00460984
	content: 004789D0
	table: 00460988
	content: 0048D400
	table: 0046098C
	content: 00490116
	table: 00460990
	content: 00491487
	table: 00460994
	content: 0046F629
	table: 00460998
	content: 0047E634

这里我们会发现有一些 IAT 项的值为零,我们知道,这些零是分隔符,并没有实质性的作用,所以我们还需要添加相应的命令跳过这些分隔符。

```

var table
var content

    mov table,460818

start:

    cmp table,460F28

    ja final

    cmp [table],50000000

    ja ToSkip

    mov content,[table]

    cmp content,0

    je ToSkip

    log content

    log table

ToSkip:

    add table,4

    jmp start

final:

    ret

```

这里我们添加了两条命令跳过值为零的 IAT 项,我们再次执行该脚本看看效果。

Address	Message
	content: 0047FCA8
	table: 00460818
	content: 00486F41
	table: 0046081C ASCII "AoH"
	content: 00480098
	table: 00460820
	content: 0047A4F9
	table: 00460824
	content: 00480981
	table: 00460828
	content: 004889B6
	table: 00460974
	content: 004858A9
	table: 00460978 ASCII " H"
	content: 0048082F
	table: 0046097C
	content: 0046C28C
	table: 00460980
	content: 004789D0
	table: 00460984
	content: 00480400
	table: 00460988
	content: 00490116
	table: 0046098C
	content: 004914B7
	table: 00460990
	content: 0046F629
	table: 00460994
	content: 0047E634
	table: 00460998
	content: 0047C5F3
	table: 0046099C
	content: 00482308
	table: 004609A0
	content: 00483086
	table: 004609A4
	content: 00480D45
	table: 004609A8
	content: 00477A69
	table: 004609AC ASCII "izG"
	content: 0048D17B
	table: 004609B0
	content: 00492469
	table: 004609B4
	content: 0048C1E9
	table: 004609B8

我们可以看到所有的待修复的 IAT 项的地址以及内容都被打印出来了,下面我们来尝试修复这些项。

接下来我们要做的就是定位到重定向的 IAT 项实际要调用的 API,我们可以利用内存写入断点来定位。当断下来的时候我们将 API 函数的地址填充到对应的 IAT 项中,接下来我们不让其去调用实际要调用的 API 函数,而是让其继续遍历 IAT 中的下一项。

脚本中设置内存写入断点用到的是 BPWM 这个命令,具体用法如下:

BPWM 地址,大小

在指定地址处,设置一个内存写入断点。”大小”是指内存中的字节大小。

例子:

BPWM 401000, FF

通过这个命令我们就可以对指定内存单元设置内存写入断点了。

mov eip,content

bpwm table,4

这里我们首先将 EIP 指向重定向后函数的入口点,接着对待修复的 IAT 项设置内存写入断点。

接下来通过 COB 命令继续往下执行。

COB 命令的具体用法如下:

COB

发生中断后,让脚本继续执行(移除 EOB 指令)

例子:

COB

mov eip,content

bpwm table,4

cob ToRepair

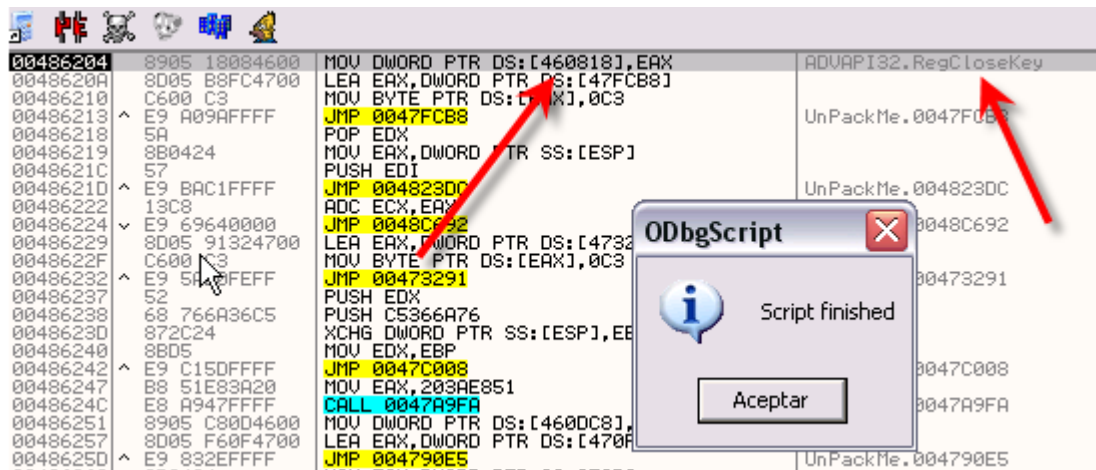
run

ToRepair:

ret

这段脚本的意思就是说对待修复的 IAT 项设置内存写入断点,接着运行起来,当发生中断的时候就跳转 ToRepair 标签处。

这里 ToRepair 标签处,我只添加了一个 ret 命令,只是为了测试是否能成功断在第一个待修复的 IAT 项被写入的地方。



我们可以看到执行完该脚本以后,正好断在了向第一个待修复的 IAT 项写入 API 函数地址的指令处,下面我们可以通过 STI 命令单步执行这一行,看看第一个 IAT 项被修复后的效果。

STL 命令的用法如下:

STI

相当于在 OD 中按 F7,单步步入

例子:

sti

mov eip,content

bpwm table,4

cob ToRepair

run

ToRepair

sti

ret

好,我们执行该修改后的脚本,执行完毕以后,可以看到第一个重定向的 IAT 项的值成功被修复了。

The screenshot shows the OllyDbg interface with assembly code loaded. An ODbgScript dialog box is open, indicating the script has finished. The assembly code includes various instructions like MOV, LEA, MOV, JMP, ADC, POP, AND, ADD, OR, XCHG, IMUL, CMP, JE, PUSH, POP, ROL, and ADD. The hex dump at the bottom shows memory addresses and their corresponding hex and ASCII values. A red arrow points to the first entry in the hex dump table.

Address	Hex dump	ASCII
00460818	F0 6B DA 77 41 6F 48 00 98 00 48 00 F9 A4 47 00	-k rWAcH.ü.H.~ãG.
00460828	81 09 48 00 00 00 00 00 CF 65 C3 58 D8 03 C4 58	ü#H.....deXi-X
00460838	00 00 00 00 D4 6A EF 77 86 95 EF 77 89 6A EF 77ëj'wfö'wëj'w
00460848	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00L.....

我们可以看到第一个待修复的 IAT 项已经被修复了,下面我们要做的就是继续定位下一个待修复的 IAT 项,并且确保不执行当前 IAT 项实际要调用的 API 函数,以免出错。


```

004271B0 Single step event at 004271B0
74CC0000 Module C:\WINDOWS\system32\oledlg.dll
74CC1000 Code size in header is 00010400, extending to size of s
00401000 Set virtual address to ollybone module for NX remove
content: 0047FCA8
table: 00460818
00486204 Memory breakpoint when writing to [00460818]
content: 00486F41
table: 0046081C ; ASCII "AoH"
003A003B Exception 00000006
content: 00480098
table: 00460820
7C9112B4 Access violation when reading [00000001]
003A003B Exception 00000006
content: 0047A4F9
table: 00460824
00481732 Access violation when reading [00000000]
0012FFCA Access violation when reading [00000006]
0012FFCA Access violation when reading [00000006]
0012FFCA Debugged program was unable to process exception
Process terminated, exit code C0000005 (-1073741819.)
Command

```

这里我们可以看到在脚本执行的过程中,产生了异常,出错了。我们在编写脚本的过程中应该充分考虑到这一点-哪些地方会产生异常,我们需要检查当前是否处于我们预想的位置,这里我们不修改任何东西,直接用 ODbgScript 插件调试看看哪里出错了。

004271B0	55	PUSH EBP		
004271B1	8BEC	MOV ERP,ESP		
Script Execution				
Line	Command	Result	EIP	Values <---
10	cmp content,0			
11	je ToSkip			
12	log content			
13	log table			
14	mov eip,content			
15	bpwm table,4			
16	cob ToRepair			
17	run			
18	ToRepair:			
19	sti			
20	ToSkip:			
21	add table,4			
22	jmp start			
23	final:			
24	ret			
004271F4	C1E1 08	SHL ECX,8		
004271F7	03CA	ADD ECX,EDX		

我们在第 19 行处按 F2 键设置一个断点,执行这个 sti 命令以后,第一个待修复的 IAT 项就会被修复,我们开始执行该脚本看看效果。

004271B0	55	PUSH EBP		
004271B1	8BEC	MOV ERP,ESP		
Script Execution				
Line	Command	Result	EIP	Values <---
10	cmp content,			
11	je ToSkip			
12	log content			
13	log table			
14	mov eip,cont			
15	bpwm table,			
16	cob ToRepair			
17	run			
18	ToRepair:			
19	sti			
20	ToSkip:			
21	add table,4			
22	jmp start			
23	final:			
24	ret			
004271F4	C1E1 08			
004271F7	03CA	ADD ECX,EDX		

我们单击鼠标右键选择 Resume,恢复脚本的执行,就可以看到断在 sti 这条命令处了。

Line	Command	Result	EIP	Values <---
1	var table		004271B0	
2	var content			
3	mov table,460818		004271B0	
4	start:			
5	cmp table,460F28		004271B0	460818
6	ja final			
7	cmp [table],50000000			47FCB8?60818
8	ja ToSkip			
9	mov content,[table]			47FCB8?60818
10	cmp content,0			47FCB8
11	je ToSkip			
12	log content			47FCB8
13	log table			460818
14	mov eip,content			47FCB8
15	bpwm table,4			460818
16	cob ToRepair			
17	run		004271B0	
18	ToRepair:			
19	sti		00486204	
20	ToSkip:			
21	add table,4			
22	jmp start			
23	final:			
24	ret			

虽然在调试脚本过程中我们可以精确看到 EIP 的值,但是在脚本中我们还是要对其进行检查。

00486204	8905 18084600	MOV DWORD PTR DS:[460818],EAX	ADVAPI32.RegCloseKey
0048620A	8D05 B8FC4700	LEA EAX,DWORD PTR DS:[47FCB8]	
00486210	C600 C3	MOV BYTE PTR DS:[EAX],0C3	
00486213	E9 A09AFFFF	JMP 0047FCB8	UnPackMe.0047FCB8
00486218	5A	POP EDX	
00486219	8B0424	MOV EAX,DWORD PTR SS:[ESP]	
0048621C	57	PUSH EDI	
0048621D	E9 BAC1FFFF	JMP 004823DC	

我们按 S 键执行该 sti 命令的话,第一个待修复的 IAT 项就会被修复。

00486204	8905 18084600	MOV DWORD PTR DS:[460818],EAX	
0048620A	8D05 B8FC4700	LEA EAX,DWORD PTR DS:[47FCB8]	
00486210	C600 C3	MOV BYTE PTR DS:[EAX],0C3	
00486213	E9 A09AFFFF	JMP 0047FCB8	
00486218	5A	POP EDX	
00486219	8B0424	MOV EAX,DWORD PTR SS:[ESP]	
0048621C	57	PUSH EDI	
0048621D	E9 BAC1FFFF	JMP 004823DC	
00486222	13C8	ADC ECX,EAX	
00486224	E9 69640000	JMP 0048C692	
00486229	8D05 91324700	LEA EAX,DWORD PTR DS:[473291]	
0048622F	C600 C3	MOV BYTE PTR DS:[EAX],0C3	
00486232	E9 5AD0FEFF	JMP 00473291	
00486237	52	PUSH EDX	
00486238	68 766A36C5	PUSH C5366A76	
0048623D	872C24	XCHG DWORD PTR SS:[ESP],EBP	
00486240	8BD5	MOV EDX,EBP	
00486242	E9 C15DFFFF	JMP 0047C008	
00486247	B8 51E83A20	MOV EAX,203AE851	
0048624C	E8 A947FFFF	CALL 0047A9FA	
00486251	8905 C80D4600	MOV DWORD PTR DS:[460DC8],EAX	
00486257	8D05 F60F4700	LEA EAX,DWORD PTR DS:[470FF6]	
0048625C	E9 832EFFFF	JMP 004790E5	
00486262	8B0424	MOV EAX,DWORD PTR SS:[ESP]	
DS:[00460818]=77DA6BF0 (ADVAPI32.RegCloseKey)			
Address	Hex dump	ASCII	
00460818	F0 6B DA 77 41 6F 48 00 98 00 48 00 F9 A4 47 00	-k rWA	
00460828	81 D9 48 00 00 00 00 00 CF 65 C3 58 D8 03 C4 58	u^H..	

按道理来说,接下来将继续修复第二个待修复的 IAT 项,我们按 S 键继续跟踪,看看会发生什么。

Script Execution				
Line	Command	Result	EIP	Values <---
1	var table		004271B0	
2	var content			
3	mov table,460818		004271B0	
4	start:			
5	cmp table,460F28		0048620A	46081C,460818
6	ja final		0048620A	
7	cmp [table],50000000		004271B0	47FCA8?60818
8	ja ToSkip			
9	mov content,[table]			47FCA8?60818
10	cmp content,0			47FCA8
11	je ToSkip			
12	log content			47FCA8
13	log table			460818
14	mov eip,content			47FCA8
15	bpwm table,4			460818
16	cob ToRepair			
17	run		004271B0	
18	ToRepair:			
19	sti		00486204	
20	ToSkip:			
21	add table,4		0048620A	460818
22	jmp start		0048620A	
23	final:			
24	ret			

这里我们可以看到 table 这个变量(该变量实际上是 IAT 的指针)已经指向 IAT 中的下一项了,我们继续跟踪。

00479B7A=UnPackMe.00479B7A			
Address	Hex dump	ASCII	
00460818	F0 6B DA 77 41 6F 48 00 98 00 48 00 F9 A4 47 00	-k rWAoH.ü.H.~RG.	
00460828	81 D9 48 00 00 00 00 00 CF 65 C3 58 D8 03 C4 58	ü-H....deXi-X	
00460838	00 00 00 00 04 6A EF 77 66 95 EF 77 89 6A EF 77	...ëj'wfö'wëj'w	
00460848	F3 AD EF 77 ED D9 EF 77 99 88 EF 77 C0 B5 EF 77	%i'wY'w0i'wLä'w	
00460858	2A 7D EF 77 B2 7C EF 77 77 53 F2 77 1E C9 F1 77	*J'wü!'wWS=wAF:w	
00460868	0C BC EF 77 52 D4 EF 77 FA 8D EF 77 F1 DD EF 77	.ä'wRE'wL'w±i'w	

接下来是修复第二个待修复的 IAT 项。

Script Execution				
Line	Command	Result	EIP	Values <---
1	var table		004271B0	
2	var content			
3	mov table,460818		004271B0	
4	start:			
5	cmp table,460F28		0048620A	46081C,460818
6	ja final			
7	cmp [table],50000000			486F41?6081C,47FCA8?60818
8	ja ToSkip			
9	mov content,[table]			486F41?6081C,47FCA8?60818
10	cmp content,0			486F41,47FCA8
11	je ToSkip			
12	log content			486F41,47FCA8
13	log table			46081C,460818
14	mov eip,content			486F41,47FCA8
15	bpwm table,4		0048620A	460818
16	cob ToRepair		004271B0	
17	run		004271B0	
18	ToRepair:			
19	sti		00486204	
20	ToSkip:			
21	add table,4		0048620A	460818
22	jmp start		0048620A	
23	final:			
24	ret			

我们跟踪到了这里,这里将对第二个待修复的 IAT 项设置内存写入断点,我们继续跟踪,我们可以看到当我们执行 run 命令的时候出错了。

Script Execution				
Line	Command	Result	EIP	Values <---
1	var table		004271B0	
2	var content		004271B0	
3	mov table,460818		004271B0	
4	start:			
5	cmp table,460F28		0048620A	46081C,460818
6	ja final			
7	cmp [table],50000000			486F41?6081C,47FCA8?60818
8	ja ToSkip			
9	mov content,[table]			486F41?6081C,47FCA8?60818
10	cmp content,0			486F41,47FCA8
11	je ToSkip			
12	log content			486F41,47FCA8
13	log table			46081C,460818
14	mov eip,content			486F41,47FCA8
15	bpwm table,4			46081C,460818
16	cob ToRepair			
17	run		0048620A	
18	ToRepair:			
19	sti		00486204	
20	ToSkip:			
21	add table,4		0048620A	460818
22	jmp start		0048620A	
23	final:			
24	ret			

Command:

Exception 00000006 - use Shift+F7/F8/F9 to pass exception to program

我们来看一看是哪里导致的错误。

Script Execution				
Line	Command	Result	EIP	Values <---
1	C:\Script.txt		004271B0	
2	var table		004271B0	
3	var content		004271B0	
4	mov table,460818		004271B0	
5	start:			
6	cmp table,460F28		0048620A	46081C,460818
7	ja final			
8	cmp [table],50000000			486F41?6081C,47FCA8?60818
9	ja ToSkip			
10	mov content,[table]			486F41?6081C,47FCA8?60818
11	cmp content,0			486F41,47FCA8
12	je ToSkip			
13	log content			486F41,47FCA8
14	log table			46081C,460818
15	mov eip,content			486F41,47FCA8
16	bpwm table,4			46081C,460818
17	cob ToRepair			
18	run		0048620A	
19	ToRepair:			
20	sti		00486204	
21	ToSkip:			
22	add table,4		0048620A	460818
23	jmp start		0048620A	
24	final:			
25	ret			

这里我们再次跟踪到了第 17 行的 run 命令处,我们单击鼠标右键选择 Abort 菜单项,终止脚本的执行,下面我们在 OD 中继续往下跟踪,看看是哪里导致的错误。

Address	Disassembly	Comment
00486F41	E8 342CFFFF	CALL 00479B7A
00486F46	E9 59FAFEFF	JMP 004769A4
00486F4B	81D5 77D1DE21	ADC EBP,21DED177
00486F51	E9 A0250000	JMP 004894F6
00486F56	13F0	ADC ESI,EAX
00486F58	5F	POP EDI
00486F59	23D7	AND EDX,EDI
00486F5B	03D0	ADD EBX,EBP
00486F5D	81C2 94EF1D87	ADD EDX,871DEF94
00486F63	81E2 22185F2A	AND EDX,2A5F1822
00486F69	81C2 23F8FB07	ADD EDX,D7FBF823
00486F6F	E9 F9B0FEFF	JMP 0047206D
00486F74	C3	RETN
00486F75	E9 82B5FFFF	JMP 0048252C
00486F7A	E8 65FCFEFF	CALL 00476BE4
00486F7F	E9 3248FFFF	JMP 0047B7B6
00486F84	CC	CMD ENT

这里我们处于第二个重定向 IAT 项所指向函数的入口处,我们继续跟踪。

00479B7A	870424	XCHG DWORD PTR SS:[ESP],EAX	ADVAPI32.RegCloseKey
00479B7D	58	POP EAX	
00479B7E	0F83 B2970000	JNB 00483336	UnPackMe.00483336
00479B84	8B05 94004800	MOV EAX,DWORD PTR DS:[480094]	
00479B8A	E9 DBEFFFF	JMP 00475A6C	UnPackMe.00475A6C

这里是将栈顶指针指向的内容与 EAX 的内容进行交换。

00483336	8B05 94004800	MOV EAX,DWORD PTR DS:[480094]	
0048333C	E8 955E0000	CALL 004891D6	
00483341	E9 5B78FFFF	JMP 0047ABA1	
00483346	C3	RETN	

这里是将 480094 这个内存单元的内容保存到 EAX 中,此时 480094 这个内存单元中内容为零,也就是说这条指令其实就是将 EAX 置零。

这里好像看不出什么端倪来。

既然是脚本在修复第二个 IAT 项的时候抛出的异常,那我们就来看看第二个 IAT 的项的调用处吧。

0043FA2E	FF15 1C084600	CALL NEAR DWORD PTR DS:[46081C]	UnPackMe.00486F41
0043FA34	85C0	TEST EAX,EAX	
0043FA36	75 6C	JNZ SHORT 0043FAA4	UnPackMe.0043FAA4
0043FA38	8D45 FC	LEA EAX,DWORD PTR SS:[EBP-4]	

这里我们通过单击鼠标右键选择 New origin here 将 EIP 指向该 CALL 处,接着手动给该 IAT 项设置一个内存写入断点,然后给返回地址处设置一个 BP 断点,执行以后我们会发现第二个 IAT 项的值并没有被修复,这也就是为什么会抛出异常的原因了。也就说我们的脚本逻辑上存在问题,第一个 IAT 项触发内存写入断点时断在了写入正确的 API 函数地址的指令处,但是第二项却没有。

好,这里我想到一个解决办法,虽然不是很正规,但是确实能解决这个问题。

我们对第一个待修复的 IAT 项即 460818 这一项设置内存写入断点,接着从第一个重定向后的函数入口处即 47FCA8 处开始跟踪(利用 OD 的自动跟踪功能跟踪),OD 自动跟踪大约要花 5 分钟左右的时间。部分跟踪指令序列如下:

```

00483C91 Main MOV AL,1 ; EAX=77DA6C01
00483C93 Main JMP 00483C78
00483C78 Main MOV BYTE PTR SS:[EBP-5],AL
00483C7B Main MOV AL,BYTE PTR SS:[EBP-5]
00483C7E Main POP ECX ; ECX=01000001, ESP=0012FE6C
00483C7F Main POP ECX ; ECX=77DA6C75, ESP=0012FE70
00483C80 Main POP EBP ; ESP=0012FE74, EBP=0012FE80
00483C81 Main RETN ; ESP=0012FE78
004833D5 Main TEST AL,AL ; FL=0
004833D7 Main JNZ 0047CC50
0047CC50 Main POP ECX ; ECX=00000001, ESP=0012FE7C
0047CC51 Main POP ECX ; ECX=77DA6C75, ESP=0012FE80
0047CC52 Main POP EBP ; ESP=0012FE84, EBP=0012FFB0
0047CC53 Main RETN ; ESP=0012FE88
0047691C Main MOV EAX,DWORD PTR SS:[EBP-C] ; EAX=77DA6BF0
0047691F Main MOV ESP,EBP ; ESP=0012FFB0
00476921 Main JMP 004765DC
004765DC Main JMP 0047F15C
0047F15C Main PUSH 47C9B5 ; ESP=0012FFAC
0047F161 Main JMP 00491A5F
00491A5F Main JMP 004737D4

```

004737D4 Main RETN ; ESP=0012FFB0

0047C9B5 Main POP EBP ; ESP=0012FFB4, EBP=0012FFF0

0047C9B6 Main RETN ; ESP=0012FFB8

0046E81D Main RETN ; ESP=0012FFBC

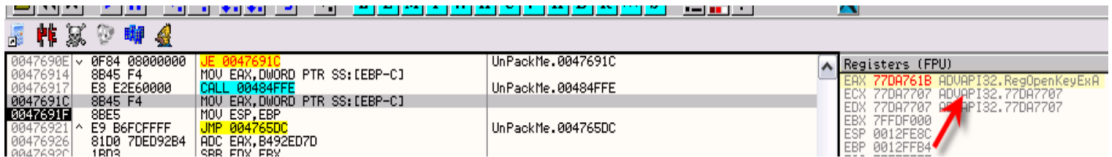
Memory breakpoint when writing to [00460818]

以上是触发 460818 处的内存写入断点之前执行的部分指令,我们需要在上面这些指令中找一条所有待修复的 IAT 项都会执行的指令。

0047691C Main MOV EAX,DWORD PTR SS:[EBP-C] ; EAX=77DA6BF0

就选着 47691C 这条指令吧,当将要执行这个指令的时候,此时 EAX 中正好保存正确的 API 函数地址,我们对这条指令设置一个硬件执行断点,看看遍历到第二个 IAT 项时,断在这里是什么情况。

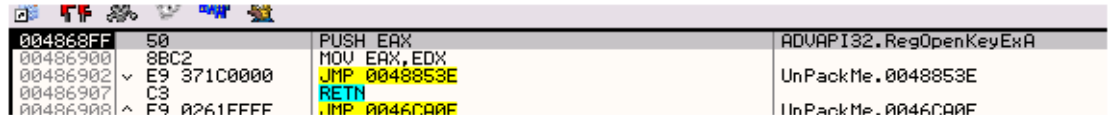
这里我们可以看到遍历到第二个 IAT 项时,EAX 中保存的确是正确的 API 函数地址,我们继续跟踪。



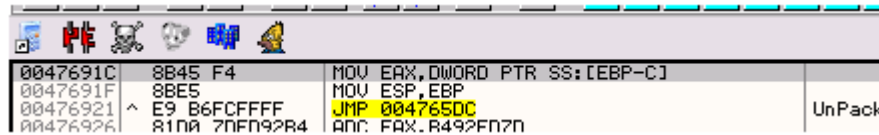
我们继续往下跟踪,会发现从 46E81D 地址处开始各个 IAT 项的执行流程就不同了。

0046E81D C3 RETN

第一个 IAT 项将跳转到修复 IAT 项的指令处,而第二个 IAT 项将跳到这里。



好了,不管怎么说,我们这里还是可以利用所有 IAT 项都会执行的这一条指令-即 47691C 这条指令。



这里正确的 IAT 函数地址将被保存到 EAX 中,所以下面我们来修改脚本,让其断在 47691C 这条指令处,执行完这条指令后,EAX 就保存了正确的 API 函数地址,接着我们将其填充到对应的 IAT 项中。

```
var table
var content
    mov table,460818
start:
    cmp table,460F28
    ja final
    cmp [table],50000000
    ja ToSkip
    mov content,[table]
    cmp content,0
    je ToSkip
    log content
    log table
    mov eip,content
    bphws 47691F,"x"
```

```

cob ToRepair
ToRepair:
    log eax
    mov [table],eax
ToSkip:
    add table,4
    jmp start
final:
    ret

```

这里我们给 47691F 这一行设置一个硬件执行断点,此时 EAX 中已经保存了正确的 API 函数地址了,接着我们将其填充到对应的 IAT 项中,循环往复直到遍历完整个 IAT 为止,我们来执行该脚本看看效果。

Address	Message
0047691F	content: 0047FCA8 table: 00460818 Hardware breakpoint 1 at UnPackMe.0047691F eax: 77DA6C17 ADVAPI32.RegCloseKey content: 00486F41 table: 0046081C ASCII "AoH"
0047691F	Hardware breakpoint 1 at UnPackMe.0047691F eax: 77DA7842 ADVAPI32.RegOpenKeyExA content: 00480098 table: 00460820
0047691F	Hardware breakpoint 1 at UnPackMe.0047691F eax: 77DAE9E4 ADVAPI32.RegCreateKeyExA content: 0047A4F9 table: 00460824
0047691F	Hardware breakpoint 1 at UnPackMe.0047691F eax: 77DAEAD7 ADVAPI32.RegSetValueExA content: 0048D981 table: 00460828
0047691F	Hardware breakpoint 1 at UnPackMe.0047691F eax: 77DA7AAB ADVAPI32.RegQueryValueExA content: 004889B6 table: 00460974
0047691F	Hardware breakpoint 1 at UnPackMe.0047691F eax: 7C80176F kernel32.GetSystemTime content: 004858A9 table: 00460978 ASCII " H"
0047691F	Hardware breakpoint 1 at UnPackMe.0047691F eax: 7C80A864 kernel32.GetLocalTime content: 0048D82F table: 0046097C
0047691F	Hardware breakpoint 1 at UnPackMe.0047691F eax: 7C810EE1 kernel32.GetFileType content: 0046C28C table: 00460980
0047691F	Hardware breakpoint 1 at UnPackMe.0047691F eax: 7C801EF2 kernel32.GetStartupInfoA content: 004789D0 table: 00460984
0047691F	Hardware breakpoint 1 at UnPackMe.0047691F eax: 7C812FAD kernel32.GetCommandLineA content: 0048D400 table: 00460988
004706A7	Access violation when reading [00000000]
004706A7	Access violation when reading [00000000] Debugged program was unable to process exception eax: 00000000

这里我们可以看到修复了一会儿后,就报错了。确切的说是在尝试修复 460988 这一项的时候发生异常了。

Address	Hex dump	ASCII
00460948	B1 95 EF 77 6F B0 EF 77 8A 5A EF 77 E9 49 F2 77	88'w0'w0Z'w0I=w
00460958	26 F1 F0 77 C9 0D F0 77 51 E0 F0 77 33 8C EF 77	&:-wF!-w00-w3I'w
00460968	6C EC EF 77 29 94 EF 77 00 00 00 00 6B 17 80 7C	lg'w)0'w...k0C!
00460978	04 A7 80 7C 51 0E 81 7C EE 1E 80 7C 1D 2F 81 7C	00C!Q00!-A0!#0!
00460988	00 04 48 00 16 01 49 00 B7 14 49 00 29 F6 46 00	.EH..01.A0I.)÷F.
00460998	34 E6 47 00 F3 C5 47 00 08 23 00 00 86 30 48 00	4pG.%+G.0#H.00H.
004609A8	45 00 48 00 69 7A 47 00 7B D1 49 00 69 24 49 00	E.H.izG.(0H.i0I.
004609B8	E9 C1 48 00 C8 0E 48 00 02 41 47 00 C7 4D 48 00	0+H.00H.0AG.0MH.

看来我们的脚本逻辑上还是有问题,我们重启 OD。

再次定位到 OEP 处,接着定位到 460988 这一个 IAT 项,看看它有没有参考引用处。

00435CC0	- FF25 88094600	JMP NEAR DWORD PTR DS:[460988]	UnPackMe.0048D400
00435CC6	- FF25 000C4600	JMP NEAR DWORD PTR DS:[460C00]	UnPackMe.00485B69

我们跟到这个函数里面看看。

0048D400	E8 0B000000	CALL 0048D410	UnPackMe.0048D410
0048D405	^ FF25 88094600	JMP NEAR DWORD PTR DS:[460988]	UnPackMe.0048D400
0048D408	^ E9 8C61FFFF	JMP 0048359C	UnPackMe.0048359C
0048D410	5A	POP EDX	
0048D411	^ 0F89 E3030000	JNS 0048D7FA	UnPackMe.0048D7FA
0048D417	^ E9 8566FFFF	JMP 00483AA1	UnPackMe.00483AA1
0048D41C	81F5 4436DDE0	XOR EBP,E0D03644	
0048D422	F7C5 0891CD72	TEST EBP,72CD9108	
0048D428	^ E9 F5130000	JMP 0048E822	UnPackMe.0048E822
0048D42D	870424	XCHG DWORD PTR SS:[ESP],EAX	
0048D430	8BEC	MOV EBP,ESP	
0048D432	52	PUSH EDX	
0048D433	03D7	ADD EDX,EDI	
0048D435	^ E9 262AFEFF	JMP 0046FE60	UnPackMe.0046FE60
0048D43A	C1C8 0A	ROR EAX,0A	
0048D43D	52	PUSH EDX	
0048D43E	68 1F11C75B	PUSH SBC7111F	
0048D443	5A	POP EDX	
0048D444	C1C2 0D	ROL EDX,0D	

我们在 47691F 处设置一个硬件执行断点看看,运行起来,看看会发生什么。

00476917	E8 E2E00000	CALL 00484FFE	UnPackMe.00484FFE
0047691C	8B45 F4	MOV EAX,DWORD PTR SS:[EBP-C]	
0047691E	8B45 F4	MOV ESP,EBP	
00476921	^ E9 B6FCFFFF	JMP 004765DC	UnPackMe.004765DC
00476926	81D0 70ED92B4	ADC EAX,B492ED7D	

Registers (FPU)
EAX 7C947A40 ntdll.RtlUnwind
ECX 7C947B62 ntdll.7C947B62
EDX 7C947B62 ntdll.7C947B62
EBX 77FDF000

这里我们可以看到此时 EAX 中保存的依然是正确的 API 函数地址,那么刚刚我们脚本修复 IAT 的过程中报错有可能是因为该壳会检测是不是在同一个时间段内 IAT 中的多项同时被修复,如果是的话就报错。

好,那么现在我们将 table 指针初始化为 460988,然后执行脚本,看看还会不会报错。

```
var table
var content
mov table,460988
start:
cmp table,460F28
ja final
cmp [table],50000000
ja ToSkip
mov content,[table]
cmp content,0
je ToSkip
log content
log table
mov eip,content
```



```

bphws 47691F,"x"
cob ToRepair
ToRepair:
    log eax
    mov [table],eax
ToSkip:
    add table,4
    jmp start
final:
    ret

```

我们可以清楚的看到发生了什么。

	content: 0048D400 table: 00460988
0047691F	Hardware breakpoint 1 at UnPackMe.0047691F eax: 7C94ABA5 ntdll.RtlUnwind content: 00490116 table: 0046098C
0047691F	Hardware breakpoint 1 at UnPackMe.0047691F eax: 7C812A99 kernel32.RaiseException content: 004914B7 table: 00460990
0047691F	Hardware breakpoint 1 at UnPackMe.0047691F eax: 7C81CAFA kernel32.ExitProcess content: 0046F629 table: 00460994
0047691F	Hardware breakpoint 1 at UnPackMe.0047691F eax: 7C801E1A kernel32.TerminateProcess content: 0047E634 table: 00460998
0047691F	Hardware breakpoint 1 at UnPackMe.0047691F eax: 7C8099A5 kernel32.GetACP content: 0047C5F3 table: 0046099C
0047691F	Hardware breakpoint 1 at UnPackMe.0047691F eax: 7C810F88 kernel32.HeapDestroy content: 00482308 table: 004609A0
0047691F	Hardware breakpoint 1 at UnPackMe.0047691F eax: 7C812C46 kernel32.HeapCreate content: 00483086 table: 004609A4
00481732	Access violation when reading [000086D3]
00481732	Access violation when reading [000086D3] Debugged program was unable to process exception eax: 000086D3 content: 00480D45 table: 004609A8
74C90000	Module C:\WINDOWS\system32\oledlg.dll Process terminated, exit code C0000005 (-1073741819.)

我们可以看到大约修复了 5 到 6 个 IAT 项后又报错了,嘿嘿,说明该壳会检测修复 5 到 6 个左右 IAT 项所用的时间,如果修复 IAT 项的时间过于连续的话,就会抛出异常。

导致我们无法继续修复,下面大家来看看我的解决方案,嘿嘿。

我的做法是监视 `ZwTerminateProcess`(调用了该函数程序就直接退出了)这个函数,我们要做的就是当将要执行该函数的时候,我们让其继续执行脚本而不是直接退出程序。

ToRepair:

```
cmp eip,7C91E88E
je ToSkip
log eax
mov [table],eax
run
```

我的机器上 `ZwTerminateProcess` 这个 API 函数的地址为 `7C91E88E`(不同的机器上这个地址可能会不同,大家根据自己机器的实际情况来决定)。当脚本执行的过程中触发了异常的话,我们判断该程序是不是在尝试调用 `ZwTerminateProcess` 这个函数来结束进程,如果是的话,我们就直接跳到 `ToSkip` 标签处去遍历下一个 IAT 项。

还有一种情况我们需要考虑就是:在修复了一个 IAT 项之后,继续执行后面的代码过程中发生了异常怎么办?即执行完了 `47691C` 这条指令之后,接着在执行后面的代码的过程中发生了异常,进而转入 `ZwTerminateProcess`。按照现在这个逻辑就是跳转到 `ZwTerminateProcess` 处,好,跳转 `ZwTerminateProcess` 处这个操作被我们的脚本捕获到了,我们脚本的处理是跳到下一个 IAT 项处,但是我们当前这个 IAT 项并没有修复啊,所以说这个逻辑还是有问题的。我们可以这样做,就是我们人为的在 `47691F` 地址处造一个异常,让其去调用 `ZwTerminateProcess` 这个函数。

经修改后,我们的脚本就变成了这个样子:

start:

```
cmp table,460F28
ja final
cmp [table],50000000
ja ToSkip
mov content,[table]
cmp content,0
je ToSkip
log content
log table
mov eip,content
bphws 47691F,"x"
mov [47691F],0
mov [476920],0
cob ToRepair
run
```

ToRepair:

```
cmp eip,7C91E88E
je ToSkip
log eax
mov [table],eax
run
```

ToSkip:

```
add table,4
jmp start
```

final:

```
ret
```

这里我们把忽略内存访问异常这个选项的对勾去掉,执行脚本。

Address	Hex dump	ASCII
004607EC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004607FC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00460800	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00-k rw
0046081C	1B 76 DA 77 F4 EA DA 77 E7 EB DA 77 83 78 DA 77	+v rwU rwB rwA rw
0046082C	00 00 00 00 CF 65 C3 58 D8 03 C4 58 00 00 00 00	...DeXtX...
0046083C	04 6A EF 77 66 95 EF 77 89 6A EF 77 F3 AD EF 77	Ej' wf0' wEj' wA' w
0046084C	ED D9 EF 77 99 88 EF 77 C0 B5 EF 77 2A 7D EF 77	Yj' w0 l' wA' w* j' w
0046085C	B2 7C EF 77 77 53 F2 77 1E C9 F1 77 0C BC EF 77	Wj' wW S= wA f= w. j' w
0046086C	52 D4 EF 77 FA 80 EF 77 F1 D0 EF 77 51 B2 EF 77	Re' w. i' wA' w0 w
0046087C	26 D5 EF 77 2A E3 EF 77 5F 39 F2 77 71 B4 EF 77	& j' w* 0' w_ 9= wq l' w
0046088C	2E AD EF 77 E1 61 EF 77 B8 85 EF 77 CC D2 EF 77	. j' wB a' w0 a' w fE' w
0046089C	43 70 EF 77 FB EA F0 77 12 83 EF 77 01 72 F0 77	Cp' w' 0- w+ a' w0 r- w
004608AC	A9 34 F0 77 D5 93 EF 77 68 EF EF 77 AA D2 EF 77	04- w' 0' wh' wE' w
004608BC	B2 6F EF 77 3F 38 F2 77 D6 E8 EF 77 68 E0 EF 77	W0' w? 8= wif' wh0' w
004608CC	00 60 EF 77 90 58 EF 77 6D AC EF 77 94 6C F0 77	. j' wE l' wM q' w0 l- w
004608DC	22 8D EF 77 3D C8 F1 77 3D 6D F0 77 6F C0 EF 77	* l' w= t= w= m- w0 l' w
004608EC	85 78 EF 77 26 D9 EF 77 FB 5E EF 77 36 8A EF 77	a l' w& j' w' ^' w6 e' w
004608FC	FC 8A EF 77 0F 62 EF 77 49 5E EF 77 97 5D EF 77	z e' w* B' wI' w0 j' w
0046090C	1A 9A EF 77 68 FA EF 77 7B C9 F0 77 DA 98 F2 77	+ 0' wk' w l' f- w r0 = w
0046091C	1A 40 F2 77 55 EA EF 77 C5 61 EF 77 70 E6 EF 77	+ 0= wU 0' w t+ a' w p0' w
0046092C	F0 81 EF 77 2D 6C EF 77 98 6E EF 77 4F 83 EF 77	- 0' w- l' w0 n' w0 a' w
0046093C	09 ED EF 77 EB AA EF 77 26 69 F0 77 B1 95 EF 77	. j' w0- w& i- w0 0' w
0046094C	6F B0 EF 77 8A 5A EF 77 E9 49 F2 77 26 F1 F0 77	c0' wE z' wU' w0 a' w
0046095C	C9 D0 F0 77 51 E0 F0 77 33 8C EF 77 6C EC EF 77	f l' w0 0- w3 0' w l0' w
0046096C	29 94 EF 77 00 00 00 00 68 17 80 7C D4 A7 80 7C	0 0' w... k 0' E0 0'!
0046097C	51 0E 81 7C EE 1E 80 7C 10 2F 81 7C 40 7A 94 7C	Q0 0'! - 0' k/ 0' 0z 0'!
0046098C	09 2A 81 7C DA CD 81 7C 16 1E 80 7C 15 99 80 7C	* 0' j- 0'! - 0' S0 0'!
0046099C	F8 0E 81 7C B6 2B 81 7C E4 9A 80 7C 51 9A 80 7C	0 0' j' A+ 0' S0 0' Q0 0'!
004609AC	E3 14 82 7C BF 50 83 7C E8 8D 83 7C AA CC 80 7C	0 0' j' 0' 0' 0' 0' 0'!
004609BC	62 2E 86 7C 77 DF 81 7C E7 4A 81 7C 58 CF 81 7C	b. 0' j' w0 j' 0' 0' 0'!
004609CC	08 2F 81 7C 9D 47 84 7C 0C 8A 83 7C 90 A4 80 7C	0' 0' 0' 0' 0' 0' 0'!
004609DC	CF BC 80 7C 62 D2 80 7C 62 15 81 7C 77 D0 80 7C	0' 0' 0' 0' 0' 0' 0'!
004609EC	5E A3 80 7C 78 34 83 7C ED 09 92 7C FD 79 92 7C	0' 0' 0' 0' 0' 0' 0'!
004609FC	D4 05 92 7C 3D 04 92 7C A7 27 81 7C 76 2E 81 7C	0' 0' 0' 0' 0' 0' 0'!
00460A0C	8B 82 85 7C A9 60 83 7C 45 1C 83 7C 77 0A 81 7C	0' 0' 0' 0' 0' 0' 0'!
00460A1C	3C 15 81 7C FE 4F 83 7C 54 5D 83 7C 23 2C 81 7C	0' 0' 0' 0' 0' 0' 0'!
00460A2C	72 67 83 7C 40 97 80 7C 27 09 83 7C 05 98 80 7C	0' 0' 0' 0' 0' 0' 0'!
00460A3C	05 10 91 7C B9 23 81 7C ED 10 91 7C B9 4C 83 7C	0' 0' 0' 0' 0' 0' 0'!
00460A4C	8A 18 92 7C 9F 2D 81 7C F1 9E 80 7C FC 38 81 7C	0' 0' 0' 0' 0' 0' 0'!
00460A5C	A5 18 82 7C 44 20 83 7C BC 22 83 7C 61 23 83 7C	0' 0' 0' 0' 0' 0' 0'!
00460A6C	47 98 80 7C 41 26 81 7C 8E 08 81 7C 87 0D 81 7C	0' 0' 0' 0' 0' 0' 0'!
00460A7C	0E 18 80 7C 24 1A 80 7C F5 D0 80 7C FE D0 80 7C	0' 0' 0' 0' 0' 0' 0'!
00460A8C	01 BF 80 7C 0F AC 80 7C 00 E7 82 7C B8 0B 83 7C	0' 0' 0' 0' 0' 0' 0'!

我们可以看到 IAT 项都被修复了,下面我来进行 dump。

004271B0 SS PUSH EBP
004271B1 8BEC MOV EBP,ESP
004271B3 6A FF PUSH -1
004271B5 68 600E4500 PUSH 450E60
004271B8 68 C8924200 PUSH 4292C8
004271BF 64:A1 00000000 MOV EAX,DWORD PTR FS:[0]
004271C5 50 PUSH EAX
004271C6 64:8925 00000000 MOV DWORD PTR FS:[0],ESP
004271CD 83C4 A8 ADD ESP,-58
004271D0 53 PUSH EB3
004271D1 56 PUSH ESI
004271D2 57 PUSH EDI
004271D3 8965 E8 MOV DWORD PTR SS:[EBP]
004271D6 FF15 DC0A4600 CALL NEAR DWORD PTR D
004271DC 33D2 XOR EDX,EDX
004271DE 90D4 MOV DL,AH
004271E0 8915 34E64500 MOV DWORD PTR DS:[45E
004271E6 8BC8 MOV ECX,EAX
004271E8 81E1 FF000000 AND ECX,0FF
004271EE 890D 30E64500 MOV DWORD PTR DS:[45E
004271F4 C1E1 08 SHL ECX,8
004271F7 03CA ADD ECX,EDX
004271F9 890D 2CE64500 MOV DWORD PTR DS:[45E
004271FF C1E8 10 SHR EAX,10
00427202 A3 28E64500 MOV DWORD PTR DS:[45E
00427207 E8 94210000 CALL 00429380
0042720C 5EC8 TEST EAX,EAX
0042720E 75 0A JNZ SHORT 0042721A
00427210 6A 1C PUSH 1C
00427212 E8 49010000 CALL 00427360
00427217 83C4 04 ADD ESP,4
0042721A E8 D12F0000 CALL 0042A1F0
0042721F 5EC8 TEST EAX,EAX
00427221 75 0A JNZ SHORT 0042722D
00427223 6A 10 PUSH 10
EBP=0004C5F0

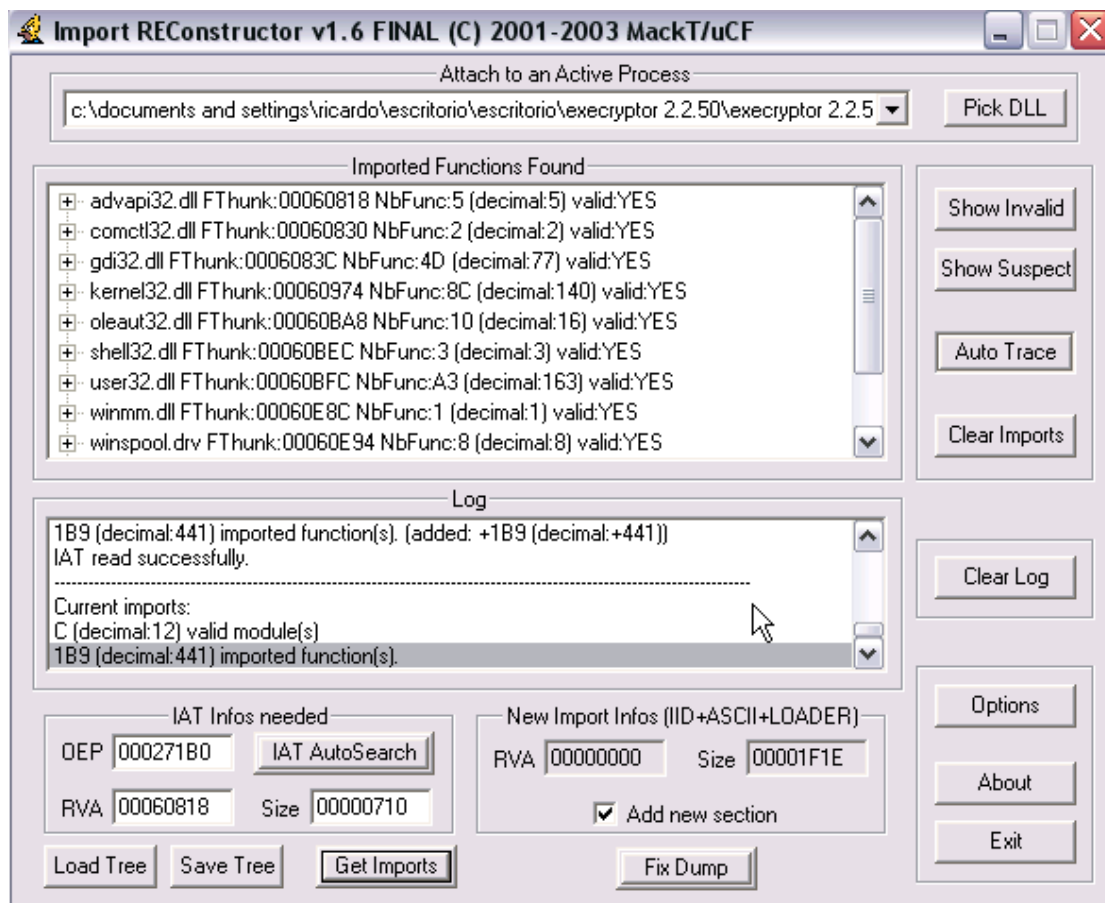
Address Hex dump
004607EC 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004607FC 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00460800 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0046081C 1B 76 DA 77 F4 EA DA 77 E7 EB DA 77 83 78 DA 77
0046082C 00 00 00 00 CF 65 C3 58 D8 03 C4 58 00 00 00 00
0046083C 04 6A EF 77 66 95 EF 77 89 6A EF 77 F3 AD EF 77
0046084C ED D9 EF 77 99 88 EF 77 C0 B5 EF 77 2A 7D EF 77
0046085C B2 7C EF 77 77 53 F2 77 1E C9 F1 77 0C BC EF 77
0046086C 52 D4 EF 77 FA 80 EF 77 F1 D0 EF 77 51 B2 EF 77
0046087C 26 D5 EF 77 2A E3 EF 77 5F 39 F2 77 71 B4 EF 77
0046088C 2E AD EF 77 E1 61 EF 77 B8 85 EF 77 CC D2 EF 77
0046089C 43 70 EF 77 FB EA F0 77 12 83 EF 77 01 72 F0 77
004608AC A9 34 F0 77 D5 93 EF 77 68 EF EF 77 AA D2 EF 77
004608BC B2 6F EF 77 3F 38 F2 77 D6 E8 EF 77 68 E0 EF 77
004608CC 00 60 EF 77 90 58 EF 77 6D AC EF 77 94 6C F0 77
004608DC 22 8D EF 77 3D C8 F1 77 3D 6D F0 77 6F C0 EF 77
004608EC 85 78 EF 77 26 D9 EF 77 FB 5E EF 77 36 8A EF 77
004608FC FC 8A EF 77 0F 62 EF 77 49 5E EF 77 97 5D EF 77
0046090C 1A 9A EF 77 68 FA EF 77 7B C9 F0 77 DA 98 F2 77
0046091C 1A 40 F2 77 55 EA EF 77 C5 61 EF 77 70 E6 EF 77
0046092C F0 81 EF 77 2D 6C EF 77 98 6E EF 77 4F 83 EF 77
0046093C 09 ED EF 77 EB AA EF 77 26 69 F0 77 B1 95 EF 77
0046094C 6F B0 EF 77 8A 5A EF 77 E9 49 F2 77 26 F1 F0 77
0046095C C9 D0 F0 77 51 E0 F0 77 33 8C EF 77 6C EC EF 77
0046096C 29 94 EF 77 00 00 00 00 68 17 80 7C D4 A7 80 7C
0046097C 51 0E 81 7C EE 1E 80 7C 10 2F 81 7C 40 7A 94 7C
0046098C 09 2A 81 7C DA CD 81 7C 16 1E 80 7C 15 99 80 7C
0046099C F8 0E 81 7C B6 2B 81 7C E4 9A 80 7C 51 9A 80 7C
004609AC E3 14 82 7C BF 50 83 7C E8 8D 83 7C AA CC 80 7C
004609BC 62 2E 86 7C 77 DF 81 7C E7 4A 81 7C 58 CF 81 7C
004609CC 08 2F 81 7C 9D 47 84 7C 0C 8A 83 7C 90 A4 80 7C
004609DC CF BC 80 7C 62 D2 80 7C 62 15 81 7C 77 D0 80 7C
004609EC 5E A3 80 7C 78 34 83 7C ED 09 92 7C FD 79 92 7C
004609FC D4 05 92 7C 3D 04 92 7C A7 27 81 7C 76 2E 81 7C
00460A0C 8B 82 85 7C A9 60 83 7C 45 1C 83 7C 77 0A 81 7C
00460A1C 3C 15 81 7C FE 4F 83 7C 54 5D 83 7C 23 2C 81 7C
00460A2C 72 67 83 7C 40 97 80 7C 27 09 83 7C 05 98 80 7C
00460A3C 05 10 91 7C B9 23 81 7C ED 10 91 7C B9 4C 83 7C
00460A4C 8A 18 92 7C 9F 2D 81 7C F1 9E 80 7C FC 38 81 7C
00460A5C A5 18 82 7C 44 20 83 7C BC 22 83 7C 61 23 83 7C
00460A6C 47 98 80 7C 41 26 81 7C 8E 08 81 7C 87 0D 81 7C
00460A7C 0E 18 80 7C 24 1A 80 7C F5 D0 80 7C FE D0 80 7C
00460A8C 01 BF 80 7C 0F AC 80 7C 00 E7 82 7C B8 0B 83 7C

OllyDump - UnPackMe_ExecCryptor2.2.50.a.exe
Start Address: 400000 Size: DE000 Dump
Entry Point: DDEA6 -> Modify: 271B0 Get EIP as OEP Cancel
Base of Code: 93000 Base of Data: 4B000
☒ Fix Raw Size & Offset of Dump Image

Section	Virtual Size	Virtual Offset	Raw Size	Raw Offset	Characteristics
.text	0004A000	00001000	0004A000	00001000	E0000020
.rdata	0000C000	0004B000	0000C000	0004B000	C0000040
.data	00009000	00057000	00009000	00057000	C0000040
.dyntioj	00003000	00060000	00003000	00060000	E0000060
.rsrc	00008000	00063000	00008000	00063000	40000040
xd4am...	00001000	00068000	00001000	00068000	C0000040
uvxww...	00027000	0006C000	00027000	0006C000	E0000020
mmainjn	0004B000	00093000	0004B000	00093000	E0000060

☐ Rebuild Import
☒ Method1 : Search JMP[API] | CALL[API] in memory image
☐ Method2 : Search DLL & API name string in dumped file

打开 IMP REC。



现在我们来修复 dump 文件,然后将 TLS Table 的指针和大小都设置为零。

00400148	00000000	DD 00000000	Global Ptr address = 0
0040014C	00000000	DD 00000000	Must be 0
00400150	00000000	DD 00000000	TLS Table address = 0
00400154	00000000	DD 00000000	TLS Table size = 0
00400158	00000000	DD 00000000	Load Config Table address = 0
0040015C	00000000	DD 00000000	Load Config Table size = 0

现在入口点就是 4271B0 了。



我们运行修复后的 dump 文件,可以看到完美运行。

好了本章到此结束。

(PS:这里执行了最终的这个脚本,我并没有修复成功,触发了异常以后,我的 OD 并没有进入 `ZwTerminateProcess` 的流程,具体原因有待进一步分析,我个人感觉这个方法不是很通用,如果大家也搞不定的话,可以参考我之前发过的那套国外的脱壳教程全集,里面有 `ExeCryptor` 的正规脱壳方案)