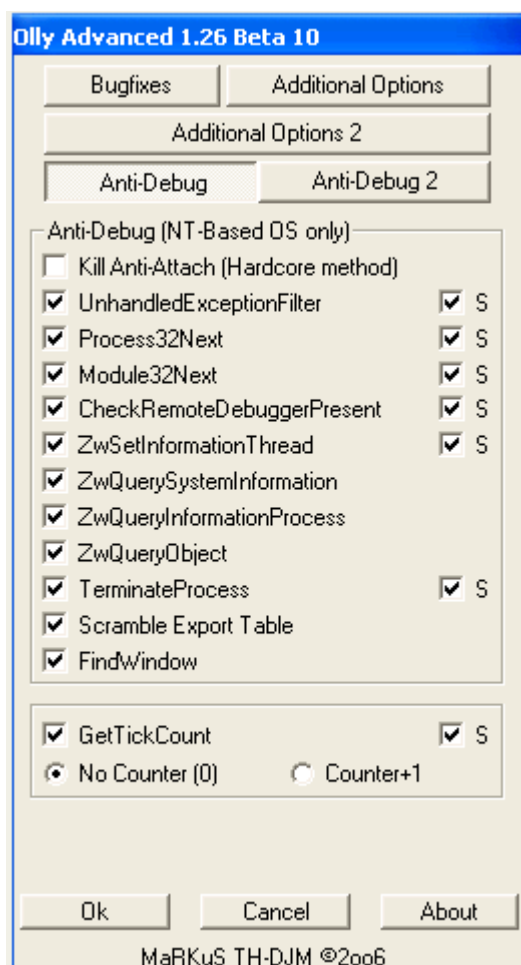


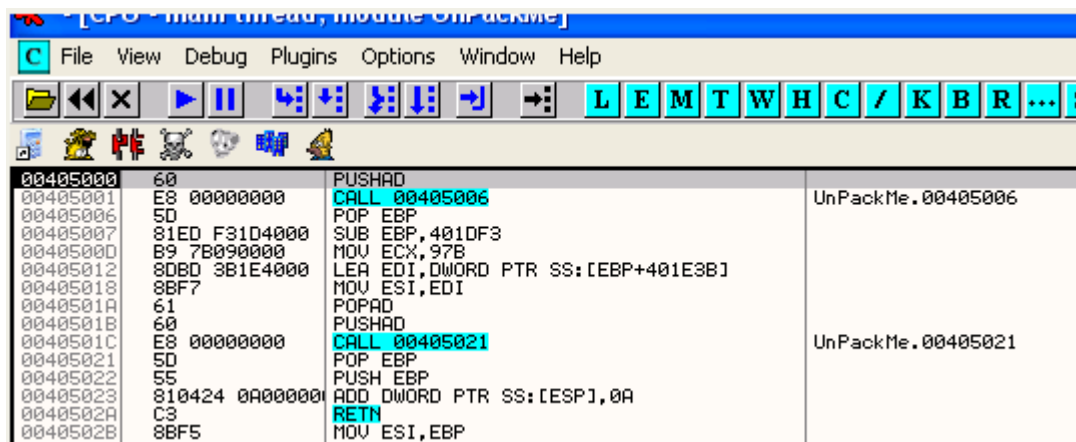
## 第五十章-再谈 Recrypt v0.80 脱壳(OutputDebugString 的妙用)

本章我们回头再来看看 Recrypt v0.80 这个壳,给大家介绍了 Patrick 和 PeSpin 这两款壳以后,再回头来看 Recrypt 这款壳,可以说是小菜一碟了。本章我会给大家介绍 OllyAdvanced 这款反反调试插件的使用,首先我们用专门定位 OEP 的那款来 OD 加载 UnPackMe\_ReCrypt0.80.exe。



我们来配置一下这个插件,这个插件提供了一个 Bugfixes(bug 修复选项卡),其中就有修复 NumOfRva Bug 这一项。

我们直接运行起来,看看会发生什么。



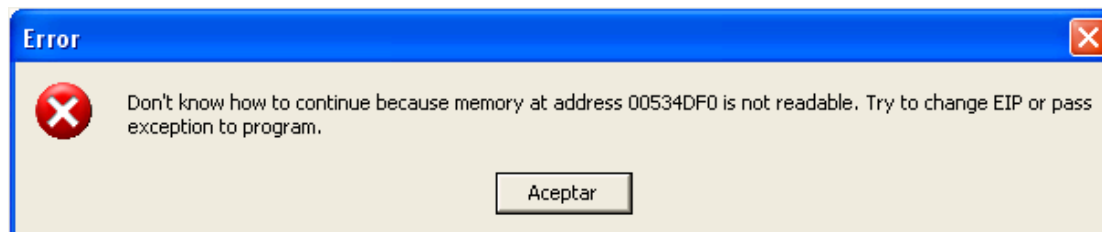
00360000	00041000				Map	R	R	
003B0000	00001000				Priv	RWE	RWE	
003C0000	00001000				Priv	RWE	RWE	
00400000	00001000	UnPackMe		PE header	Imag	R	RWE	
00401000	00001000	UnPackMe	.text	code	Imag	R	RWE	
00402000	00001000	UnPackMe	.rdata	code	Imag	R	RWE	
00403000	00001000	UnPackMe	.data	code,data	Imag	R	RWE	
00404000	00001000	UnPackMe	.rsrc	code,resour	Imag	R	RWE	
00405000	00001000	UnPackMe	RE-Crypt	code,import	Imag	R	RWE	
7C800000	00001000	kernel32		PE header	Imag	R	RWE	
7C801000	00082000	kernel32	.text	code,import	Imag	R	RWE	
7C883000	00005000	kernel32	.data	code,data	Imag	R	RWE	
7C888000	00073000	kernel32	.rsrc	code,resour	Imag	R	RWE	
7C8FB000	00006000	kernel32	.reloc	code,relocat	Imag	R	RWE	
7C910000	00001000	ntdll		PE header	Imag	R	RWE	
7C911000	0007B000	ntdll	.text	code,export	Imag	R	RWE	
7C98C000	00005000	ntdll	.data	code,data	Imag	R	RWE	
7C991000	00032000	ntdll	.rsrc	code,resour	Imag	R	RWE	
7C9C3000	00003000	ntdll	.reloc	code,relocat	Imag	R	RWE	
7F6F0000	00007000				Map	R E	R E	
7FFB0000	00024000				Map	R	R	
7FFDE000	00001000			data block	Priv	RW	RW	
7FFDF000	00001000				Priv	RW	RW	
7FFE0000	00001000				Priv	R	R	

这里我们可以看到 UnPackMe\_ReCrypt0.80 区段显示都是正常的,我们剔除掉 OllyAdvanced 这个插件看看又会是怎么样的呢。

00270000	00010000				Map	R	R	\Device\HarddiskVolume1\Windows\System32\Unicode.NLS
00290000	0003D000				Map	R	R	\Device\HarddiskVolume1\Windows\System32\LOCAL.NLS
002D0000	00041000				Map	R	R	\Device\HarddiskVolume1\Windows\System32\SORTKEY.NLS
00320000	00006000				Map	R	R	\Device\HarddiskVolume1\Windows\System32\SORTKEY.NLS
00350000	00041000				Map	R	R	
00400000	00001000	UnPackMe		PE header	Imag	R	RWE	
7C800000	00001000	kernel32		PE header	Imag	R	RWE	
7C801000	00082000	kernel32	.text	code,import	Imag	R	RWE	
7C883000	00005000	kernel32	.data	data	Imag	R	RWE	
7C888000	00073000	kernel32	.rsrc	resources	Imag	R	RWE	
7C8FB000	00006000	kernel32	.reloc	relocations	Imag	R	RWE	
7C910000	00001000	ntdll		PE header	Imag	R	RWE	
7C911000	0007B000	ntdll	.text	code,export	Imag	R	RWE	
7C98C000	00005000	ntdll	.data	data	Imag	R	RWE	
7C991000	00032000	ntdll	.rsrc	resources	Imag	R	RWE	
7C9C3000	00003000	ntdll	.reloc	relocations	Imag	R	RWE	
7F6F0000	00007000				Map	R E	R E	
7FFB0000	00024000				Map	R	R	
7FFD9000	00001000			data block	Priv	RW	RW	
7FFDF000	00001000				Priv	RW	RW	
7FFE0000	00001000				Priv	R	R	

这里我们可以看到没有 OllyAdvanced 插件的话,还没有到达 OEP 就报错了。我们来看看区段,会发现主模块 UnPackMe\_ReCrypt0.80 只显示了一个区段。这样的话我们只能通过 PEEditor 来查看代码区的起始位置以及大小,然后手动给代码段所在区域设置内存访问断点来定位 OEP 了,但是有了 OllyAdvanced 这个插件,这一切都可以省略了,嘿嘿。

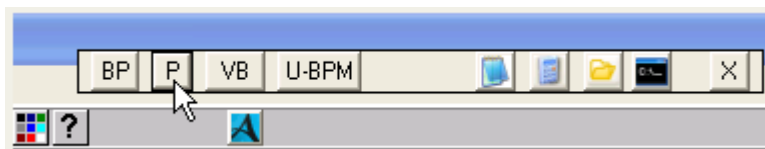
下面我们直接运行起来。



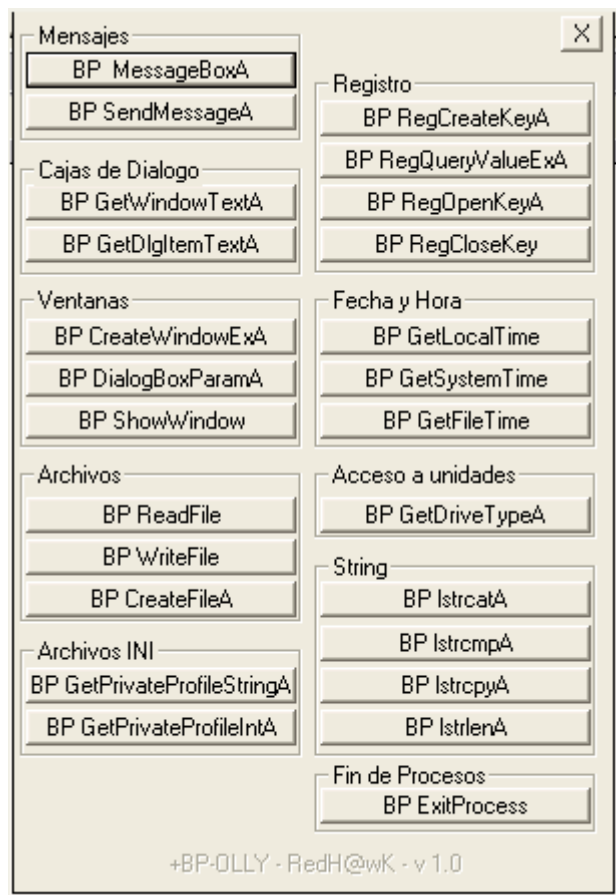
嘿嘿,可以看到弹出了一个错误框,根据提示信息可以看出 OD 遇到了无法处理的异常。我们打开日志窗口看看出错的日志信息。

00400000	Unload C:\Documents and Settings\Ricardo\Escritorio\ReCrypt 0.80\UnPackMe_ReCrypt0.80.exe
7C800000	Unload C:\WINDOWS\system32\kernel32.dll
7C910000	Unload C:\WINDOWS\system32\ntdll.dll
	Process terminated
	New process with ID 00000558 created
	Main thread with ID 00000298 created
00400000	Module C:\Documents and Settings\Ricardo\Escritorio\ReCrypt 0.80\UnPackMe_ReCrypt0.80.exe
7C800000	Module C:\WINDOWS\system32\kernel32.dll
7C910000	Module C:\WINDOWS\system32\ntdll.dll
00400000	Program entry point
7C81EB33	Debug string:
00534DF0	Access violation when executing [00534DF0]

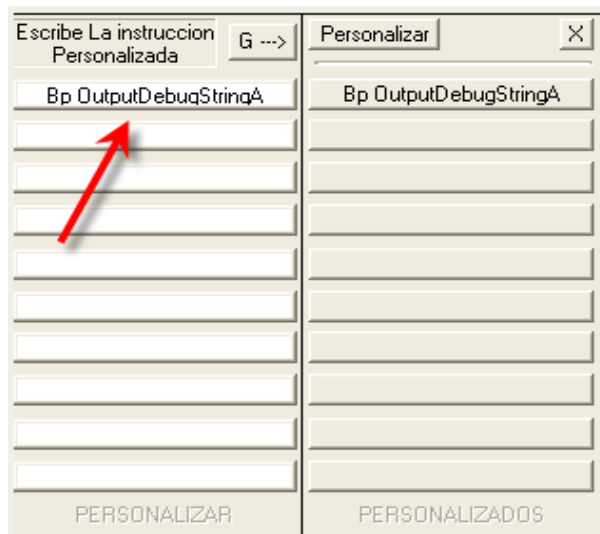
大家注意到这个 Debug string:字符串没有,说明该程序调用了 OutputDebugStringA 这个 API 函数来输出调用信息。OutputDebugString 函数用于向调试器发送一个格式化的字符串,Ollydbg 会在底端显示相应的信息。OllyDbg 存在格式化字符串溢出漏洞,非常严重,轻则崩溃,重则执行任意代码。这个漏洞是由于 Ollydbg 对传递给 kernel32!OutputDebugString()的字符串参数过滤不严导致的,它只对参数进行那个长度检查,只接受 255 个字节,但没对参数进行检查,所以导致缓冲区溢出。虽然 OllyAdvanced 以及其他一些 OD 插件提供了对 OutputDebugString 函数的修复功能,但是我们这里还是无法正常运行。我们给 OutputDebugStringA 这个 API 函数下个断点一探究竟。



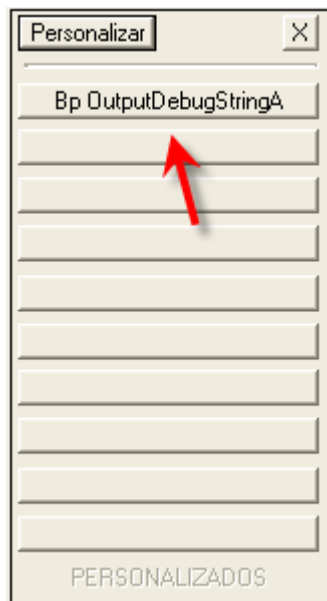
这里由于我不想每次都手动在命令栏中一个字母一个字母的输入 API 函数的名称,所以这里我使用网上的朋友写的一款插件来完成这个任务,这个插件的名字叫做+BP-Olly。有了这个插件,我们只需要通过一次简单的按钮单击就可以完成对常见 API 函数的断点设置。



我们打开 BP 这个选项卡发现并没有 OutputDebugStringA 这个 API 函数,怎么办呢?不要紧,该插件有自定义功能,我们可以通过单击 P(Personal:个人的)按钮将 OutputDebugStringA 这个 API 函数添加到列表当中。

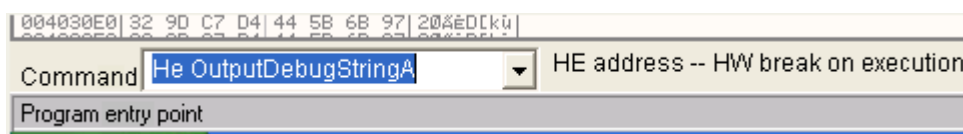


我们单击 Personalizar 按钮,左边就多显示处一个对话框,我们输入 BP OutputDebugStringA,接着单击 Guardar(添加)按钮。  
请注意,API 函数的名称大小写要正确。下面我们再次单击 P 按钮。

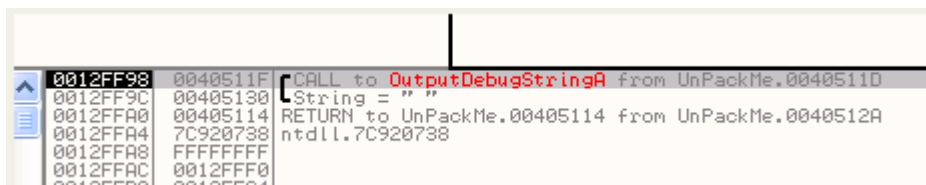


这里我们可以看到 BP OutputDebugStringA 这个按钮就出现了,我们只需要单击一下 BP OutputDebugStringA 这个按钮,就可以给 OutputDebugStringA 这个 API 函数设置上断点。是不是很方便。嘿嘿。(PS:其实国内也有很多 OD DIY 的版本也提供了这个功能,譬如说吾爱破解的 OD)

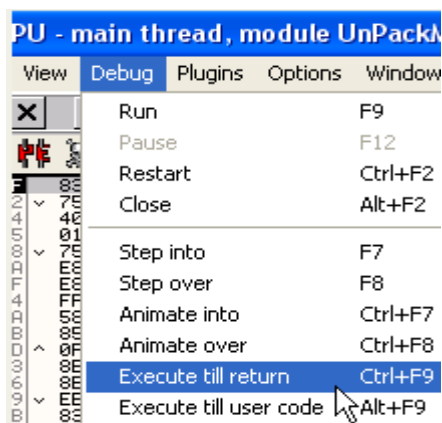
接着我们运行起来,会发现还是会报错。就是说简单的给 OutputDebugStringA 这个 API 函数设置 INT3 断点并不起作用。所以这里我们重启 OD,接着在命令栏中输入 HE OutputDebugStringA 给该 API 函数设置一个硬件执行断点试试看。



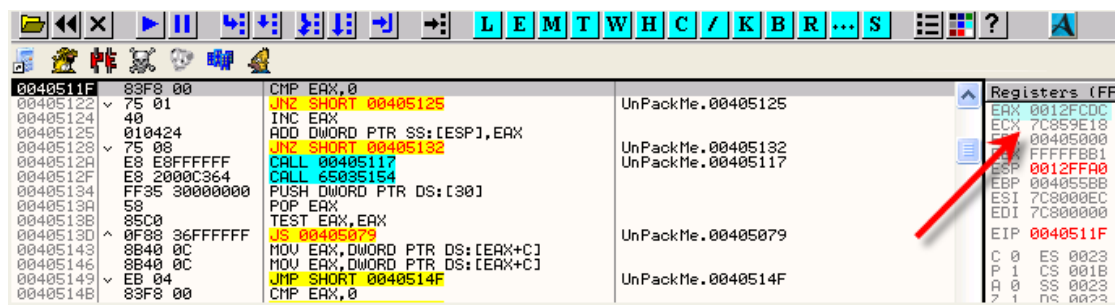
删除掉刚刚设置的 BP 断点,接着运行起来。(PS:这里下 INT3 断点会报错因为反调试插件之间存在冲突)



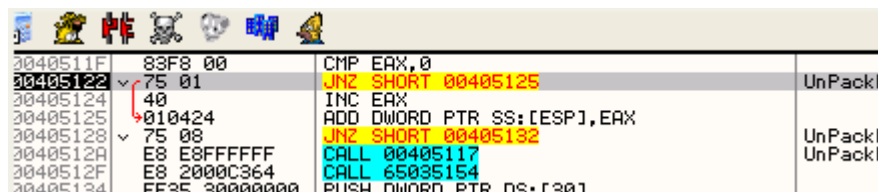
断了下来,我们执行到返回。



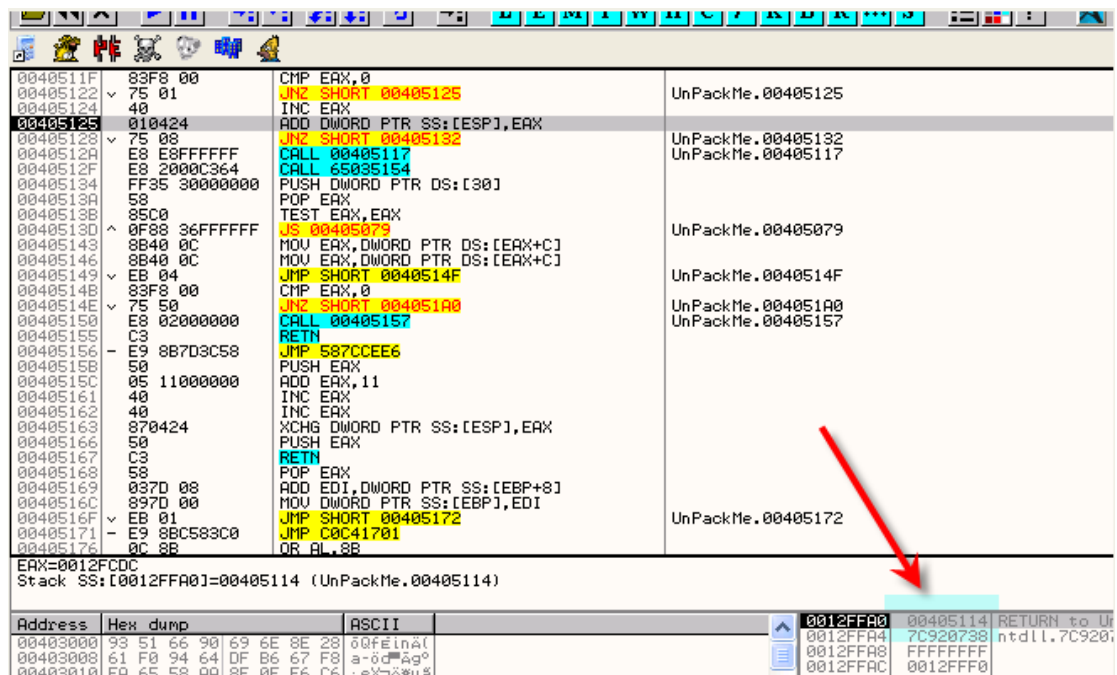
返回到了这里。



这里我们可以看到将 EAX 的值与零做比较,看看会发生什么。



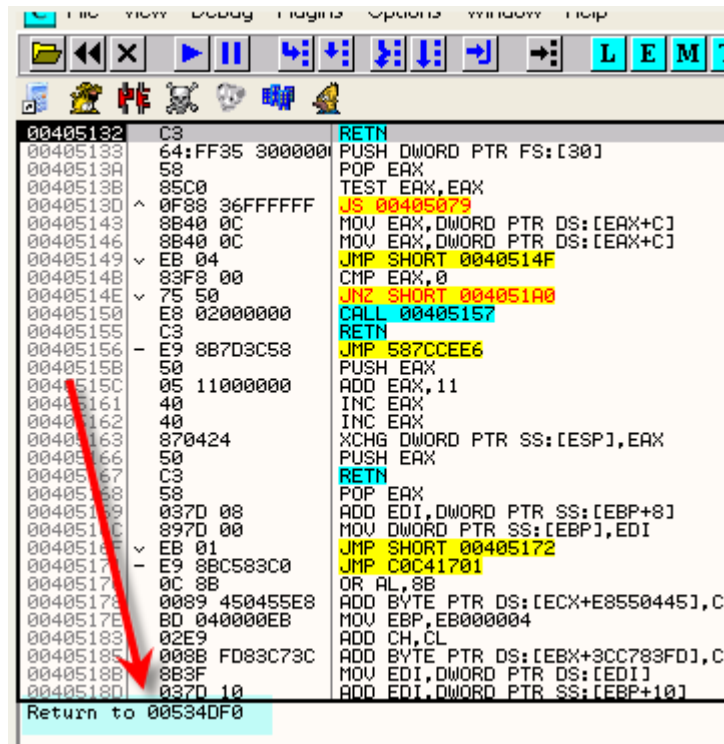
这里我们可以看到 EAX 的值不等于零,接着跳转到下面将 EAX 的值与栈顶指针指向的内容相加。



相加以后栈顶的内容变成了:



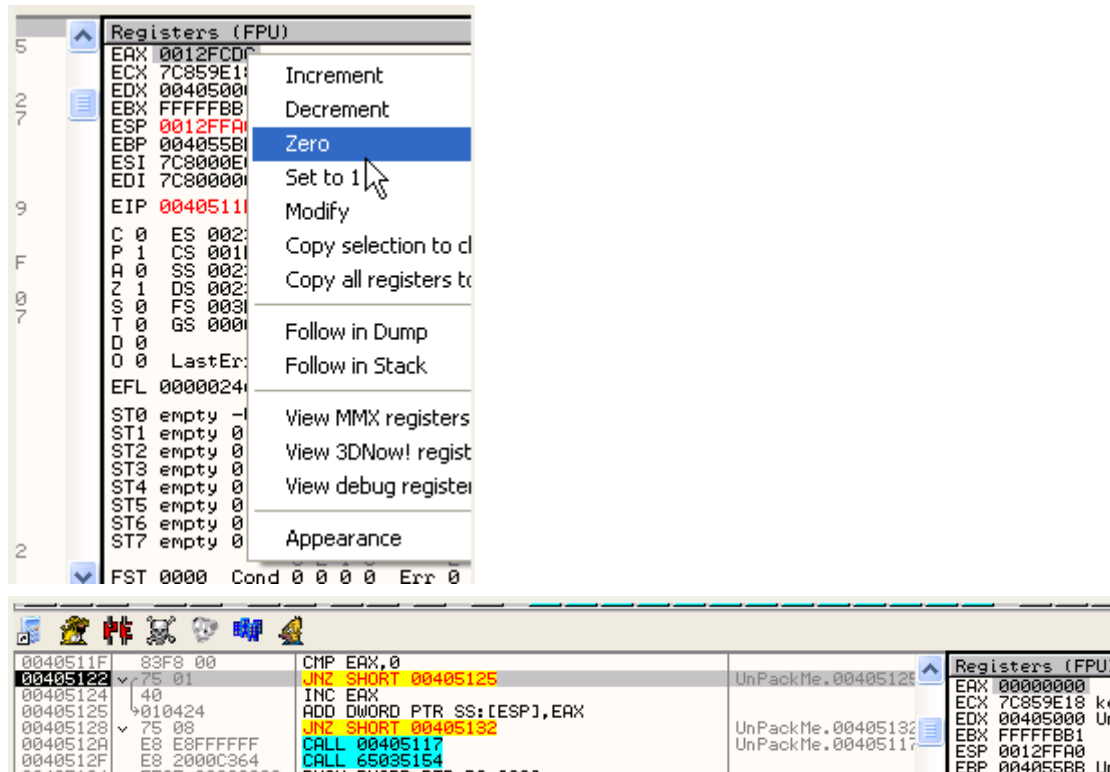
我们继续往下跟踪,会发现下一条指令是 RET,那么相加到栈顶的内容就是返回地址了。



534DF0 这个地址并不存在,所以我们直接运行起来会抛出异常,而这个异常是调试器无法处理的,所以就报错了。

好了,报错的具体原因我们已经弄明白了,下面我们来修复它。

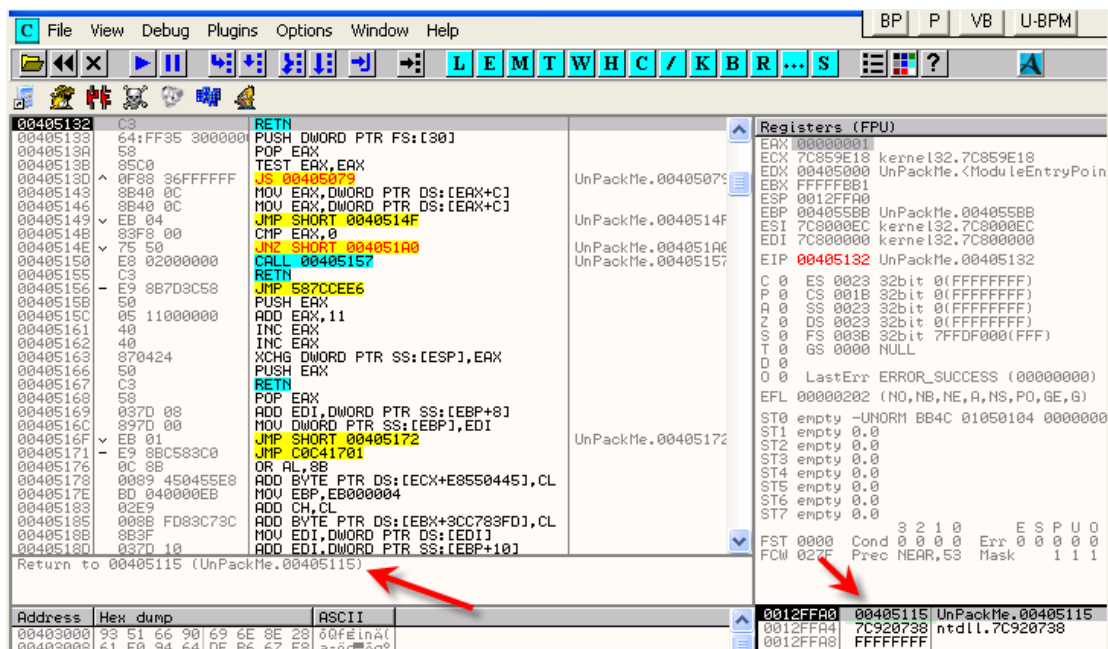
我们重启 OD,运行起来,断在了 OutputDebugStringA 处,执行到返回,接着直接将 EAX 的值修改为 0。



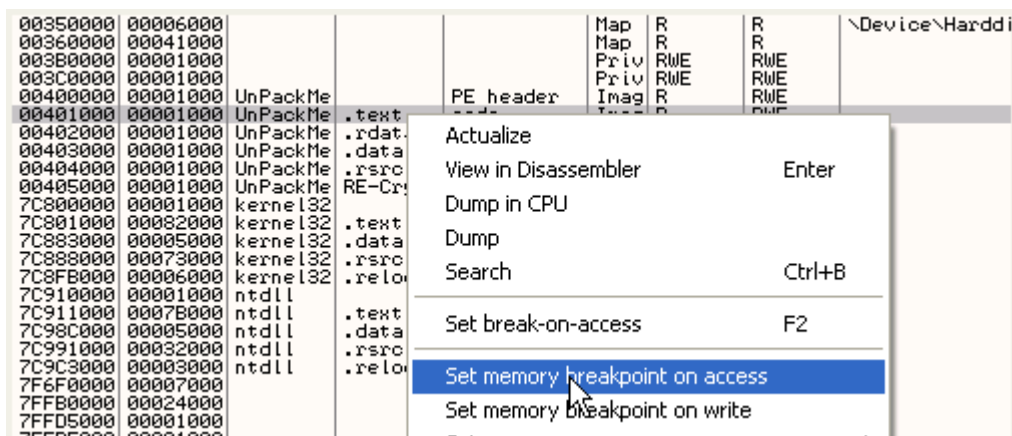
下面这个条件跳转就不会成立了,接下来 EAX 加 1,现在 EAX 的值为 1,下来继续将栈顶指针指向的内容加上 EAX,也就是加 1。

接着就是调用 RET 指令返回了。

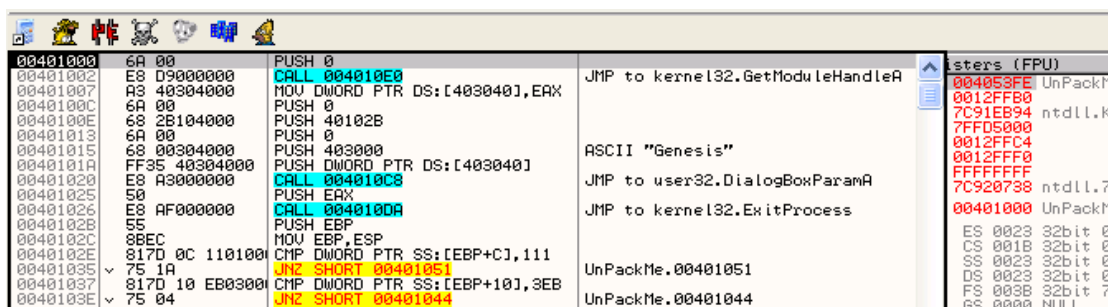




现在这个返回地址是 405115,这个地址是存在的,也就不会报错了。现在我们直接给主模块的代码段设置内存访问断点(这个 OD 是 Patch 过的,内存访问断点只是单单执行的时候才会断点来,读取或者写入并不会断下来,这样就方便我们定位 OEP 了,大家应该还记得吧),运行起来。



不一会儿就断下了 OEP 处。



这里我们就断在了 OEP 处。

00401000	6A 00	PUSH 0			
00401002	E8 D9000000	CALL 004010E0	JMP to kernel32.GetModuleHandleA		
00401007	A3 40304000	MOV DWORD PTR DS:[403040],EAX			
0040100C	6A 00	PUSH 0			
0040100E	68 2B104000	PUSH 40102B			
00401013	6A 00	PUSH 0			
00401015	68 00304000	PUSH 403000	ASCII "Genesis"		
0040101A	FF35 40304000	PUSH DWORD PTR DS:[403040]			
00401020	E8 A3000000	CALL 004010C8	JMP to user32.DialogBoxParamA		
00401025	50	PUSH EAX			
00401026	E8 AF000000	CALL 004010DA	JMP to kernel32.ExitProcess		
0040102B	55	PUSH EBP			
0040102C	8BEC	MOV EBP,ESP			
0040102E	817D 0C 110100	CMPL DWORD PTR SS:[EBP+C],111			
00401035	75 1A	JNZ SHORT 00401051	UnPackMe.00401051		
00401037	817D 10 EB0300	CMPL DWORD PTR SS:[EBP+10],3EB			
0040103E	75 04	JNZ SHORT 00401044	UnPackMe.00401044		

我们单击鼠标右键选择 Search for All intermodular calls 搜索所有的 API 函数调用处。

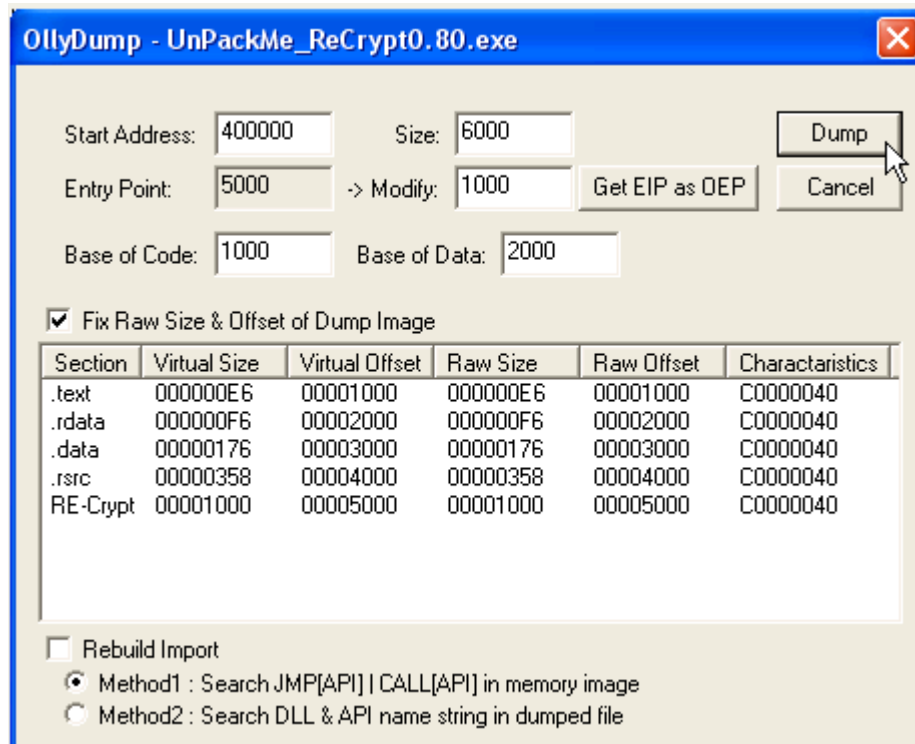
Address	Disassembly	Destination
00401000	PUSH 0	(Initial CPU selection)
00401002	CALL 004010E0	kernel32.GetModuleHandleA
00401020	CALL 004010C8	user32.DialogBoxParamA
00401026	CALL 004010DA	kernel32.ExitProcess
0040106B	CALL 004010CE	user32.GetDlgItemTextA
004010AD	CALL 004010D4	user32.MessageBoxA
004010B7	CALL 004010DA	kernel32.ExitProcess

我们可以看到只有少量的 API 函数调用,IAT 如下:

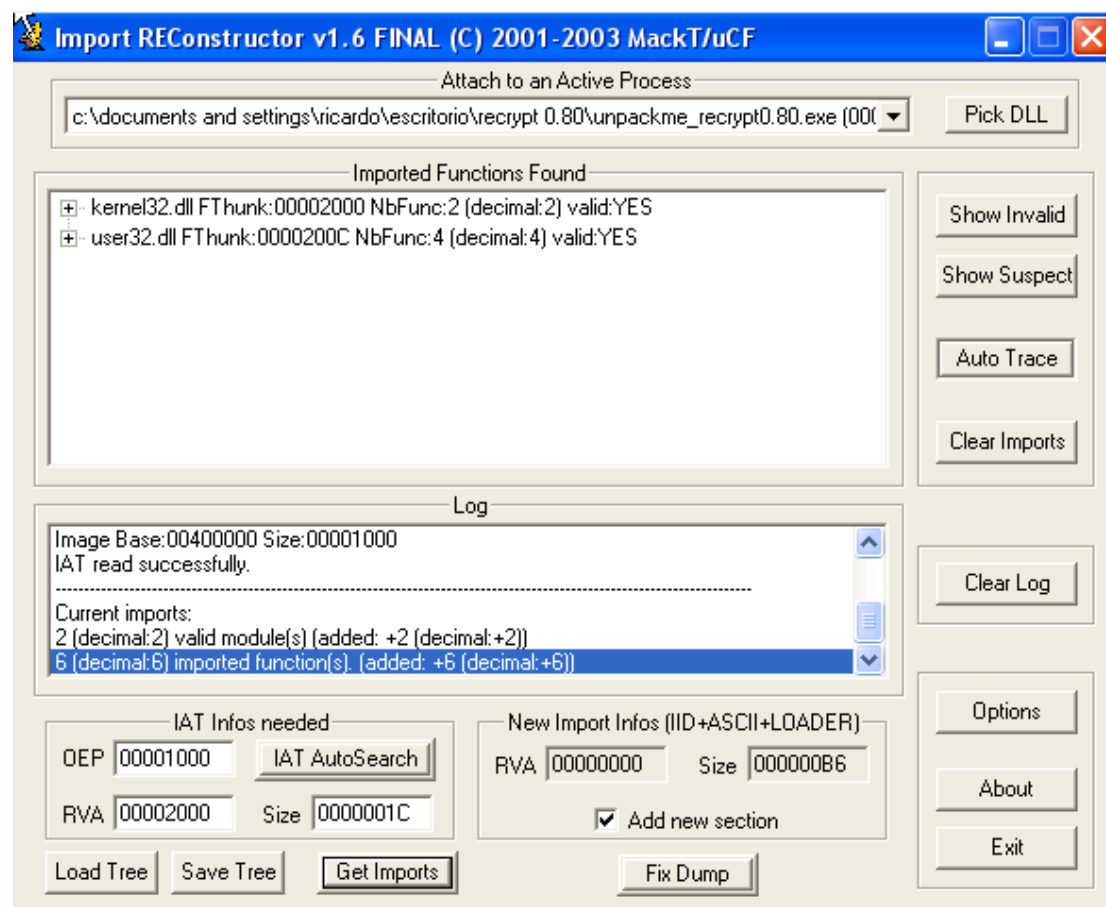
004010BF	C2 1000	RETN 10	
004010C2	- FF25 18204000	JMP NEAR DWORD PTR DS:[402018]	user32.wsprintfA
004010C8	- FF25 14204000	JMP NEAR DWORD PTR DS:[402014]	user32.DialogBoxParamA
004010CE	- FF25 10204000	JMP NEAR DWORD PTR DS:[402010]	user32.GetDlgItemTextA
004010D4	- FF25 0C204000	JMP NEAR DWORD PTR DS:[40200C]	user32.MessageBoxA
004010DA	- FF25 04204000	JMP NEAR DWORD PTR DS:[402004]	kernel32.ExitProcess
004010E0	- FF25 00204000	JMP NEAR DWORD PTR DS:[402000]	kernel32.GetModuleHandleA
004010E6	0000	ADD BYTE PTR DS:[EAX],AL	
004010E8	0000	ADD BYTE PTR DS:[EAX],AL	
004010EA	0000	ADD BYTE PTR DS:[EAX],AL	
004010EC	0000	ADD BYTE PTR DS:[EAX],AL	
004010EE	0000	ADD BYTE PTR DS:[EAX],AL	
004010F0	0000	ADD BYTE PTR DS:[EAX],AL	
004010F2	0000	ADD BYTE PTR DS:[EAX],AL	
004010F4	0000	ADD BYTE PTR DS:[EAX],AL	
004010F6	0000	ADD BYTE PTR DS:[EAX],AL	
004010F8	0000	ADD BYTE PTR DS:[EAX],AL	
004010FA	0000	ADD BYTE PTR DS:[EAX],AL	
004010FC	0000	ADD BYTE PTR DS:[EAX],AL	
004010FE	0000	ADD BYTE PTR DS:[EAX],AL	
00401100	0000	ADD BYTE PTR DS:[EAX],AL	
00401102	0000	ADD BYTE PTR DS:[EAX],AL	
00401104	0000	ADD BYTE PTR DS:[EAX],AL	
00401106	0000	ADD BYTE PTR DS:[EAX],AL	
00401108	0000	ADD BYTE PTR DS:[EAX],AL	
0040110A	0000	ADD BYTE PTR DS:[EAX],AL	
0040110C	0000	ADD BYTE PTR DS:[EAX],AL	
0040110E	0000	ADD BYTE PTR DS:[EAX],AL	
00401110	0000	ADD BYTE PTR DS:[EAX],AL	
00401112	0000	ADD BYTE PTR DS:[EAX],AL	
DS:[00402000]=7C80B529 (kernel32.GetModuleHandleA)			
Address	Hex dump	ASCII	
00402000	29 B5 80 7C A2 CA 81 7C	)AÇ!ô±u!	0012FFC4
00402008	00 00 00 00 EA 04 05 77	...0♦'w	0012FFC8
00402010	1E AC D6 77 1C B1 03 77	▲%iWLw	0012FFCC
00402018	AD A8 D1 77 00 00 00 00	¡ðW....	0012FFD0
00402020	68 20 00 00 00 00 00 00	h .....	0012FFD4
00402028	00 00 00 00 BA 20 00 00	....ll ..	0012FFD8
00402030	00 00 00 00 00 00 00 00	.....	0012FFDC
			0012FFE0

这里我们可以看到 IAT 的起始地址为 402000,结束地址为 40201C。也就是说 IAT 的起始地址的 RVA 为 2000,长度为 1C。OEP 的 RVA 为 1000,下面我们将其 DUMP 出来,接着用 IMP REC 来修复其 IAT。

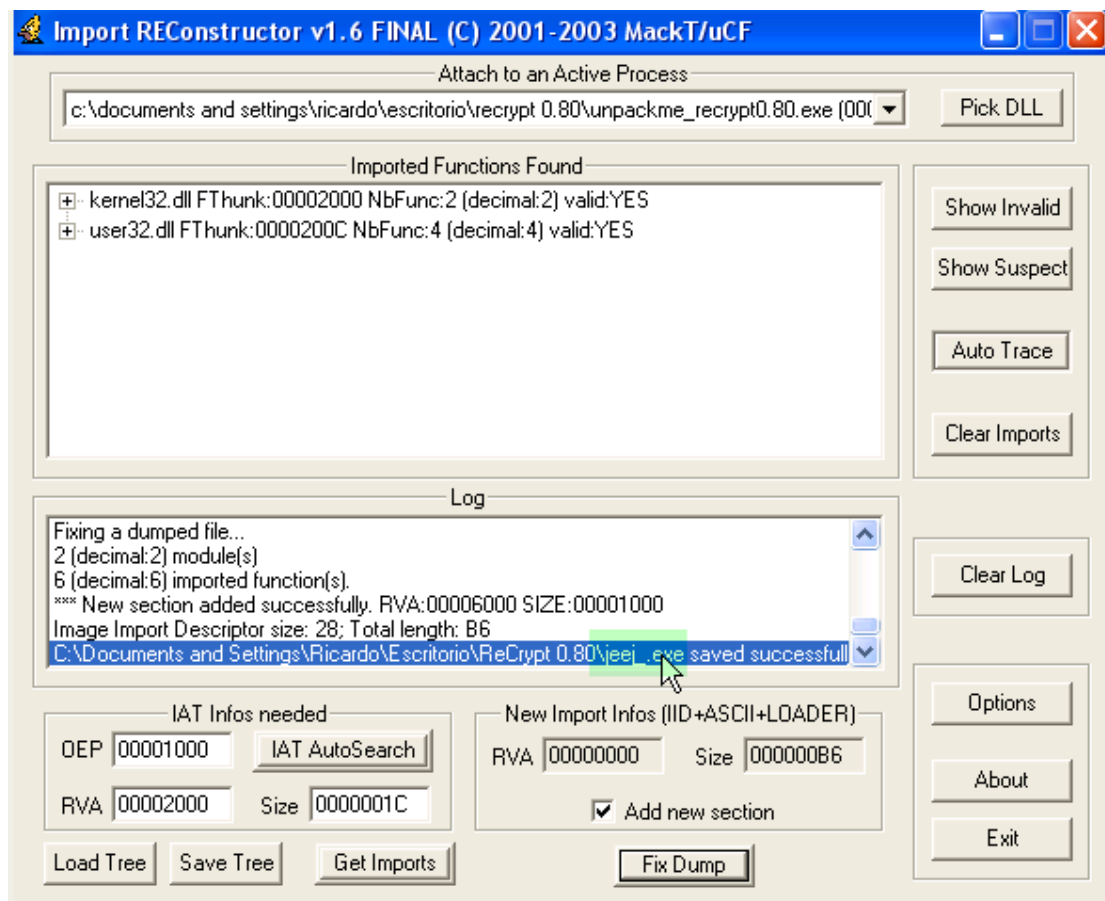




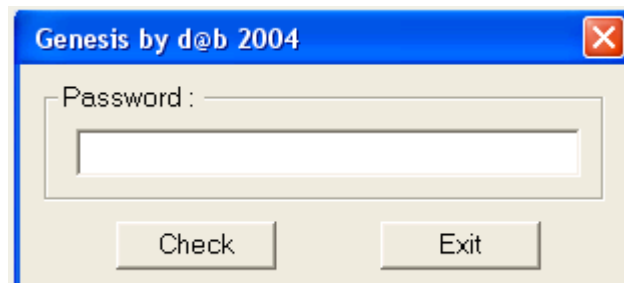
我们打开 IMP REC。



填上 OEP 的 RVA,IAT 的 RVA,大小之后单击 Get Imports 获取导入表,接着单击 Fix Dump 修复刚刚 dump 出来的文件。



我们直接运行修复后的文件。



嘿嘿,搞定。