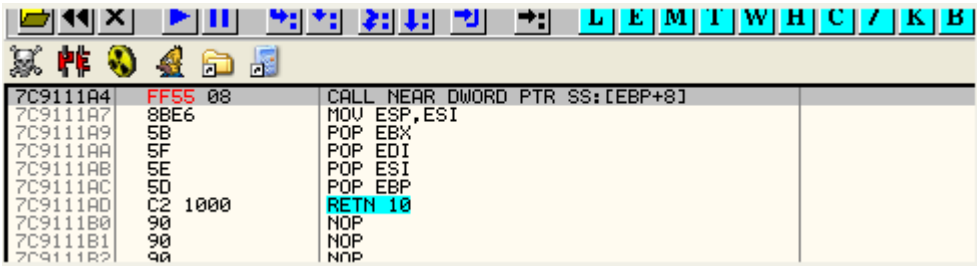
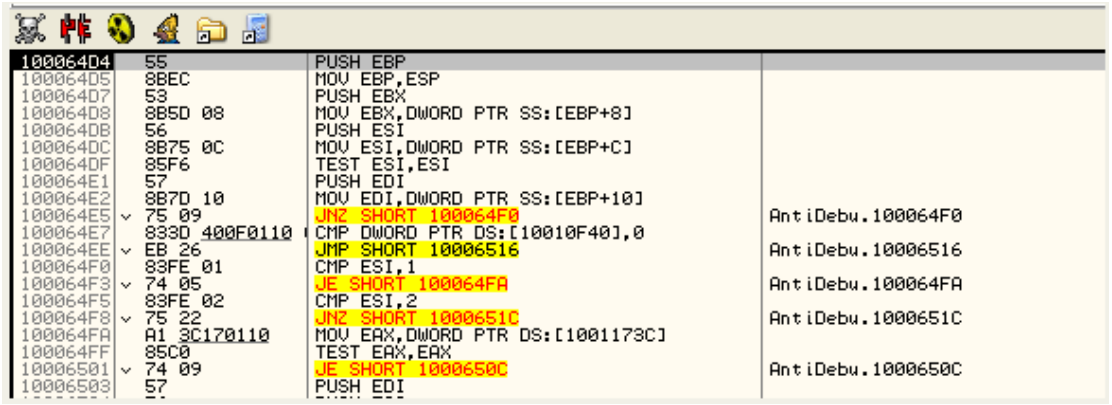
第四十七章-Patrick 的 CrackMe-Part2

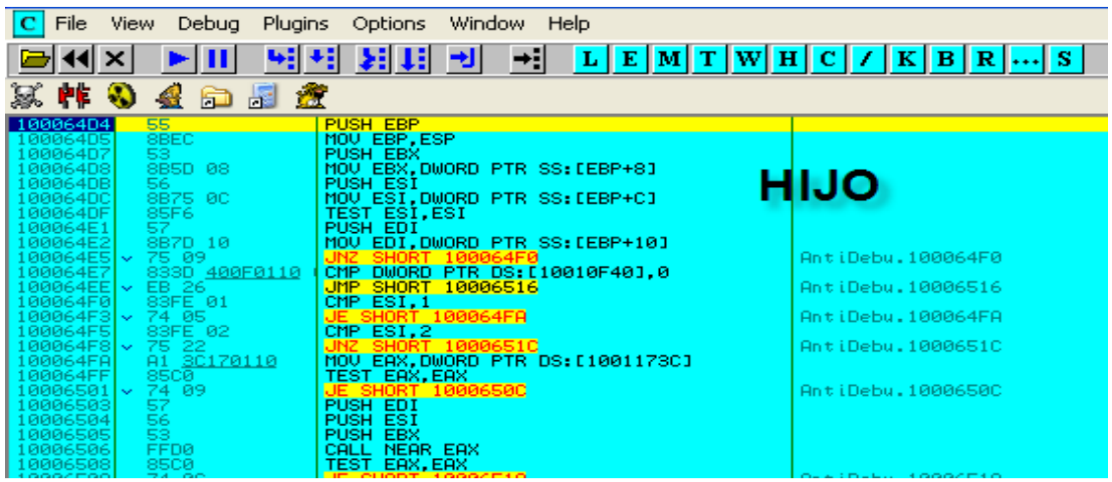
我们接着上一章的继续讲,上一章的结尾处我们是通过如下方式来附加新创建的进程的,首先将 NTDLL.DLL 中调用其他模块的入口点那一条指令设置为一个死循环,然后将 OD 设置为即时调试器(JIT),接着打开任务管理器,选中新创建的进程单击鼠标右键选择调试,这样 OD 就附加了新创建的进程。接着在死循环的指令处设置一个断点,然后将 Patch 过的字节码恢复为原始字节,然后直接按 F9 键运行两次后,模块列表窗口中就会出现 AntiDebugDll.dll 了,接着我们给 AntiDebugDll.dll 的代码段设置内存访问断点,然后删除掉之前设置的 INT 3 断点,运行起来,这样就可以断在 AntiDebugDll.dll 的入口点处了。





现在我们打开了两个 OD,其中一个被调试的为父进程,另一个被调试的为子进程。

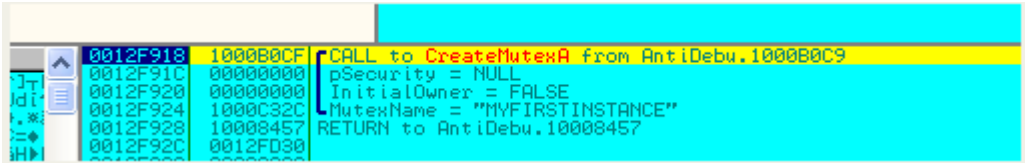
现在父进程处于 CreateProcessA 调用语句的返回地址处。



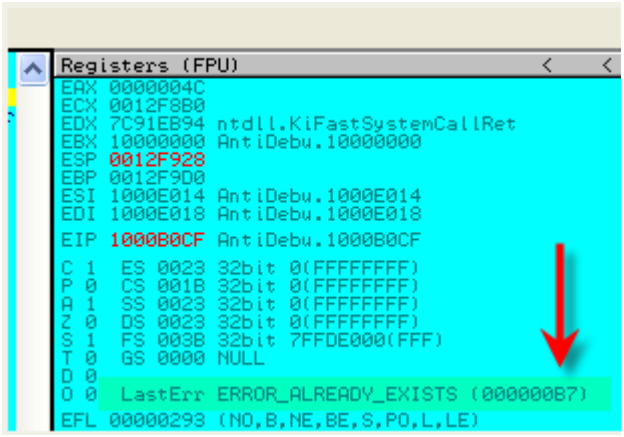
子进程处于 AntiDebugDll.dll 的入口点处。这里我将调试子进程的 OD 换了一种配色方案,这样可以防止大家在阅读的时候将两个 OD 搞混淆了。

理论上来说,现在我们需要同时模拟执行这两个进程,但实际上我们无法做到同时调试。我们只能够分别协同调试两个进程。

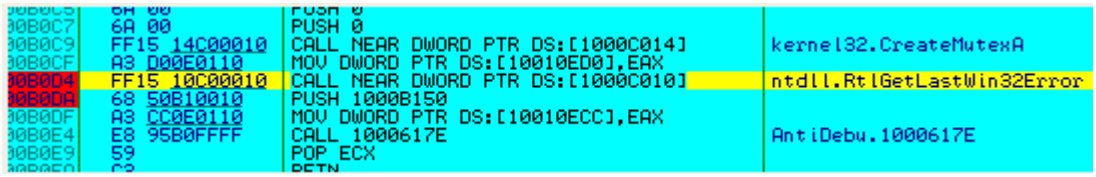
我们知道父进程中调用 CreateMutexA 这个函数,这里我们给子进程的中 CreateMutexA 也设置一个断点。



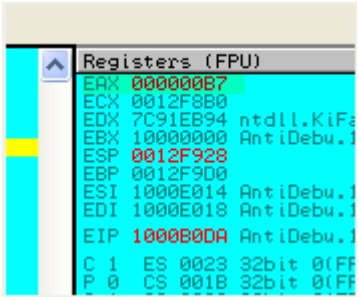
我们运行起来,断在了这里,这里由于父进程已经创建了 MYFIRSTINSTANCE 这个互斥体,所以我们如果执行了该函数,GetLastError 的错误码将返回 0xB7,即 ERROR_ALREADY_EXISTS。这里我们执行到返回验证一下。

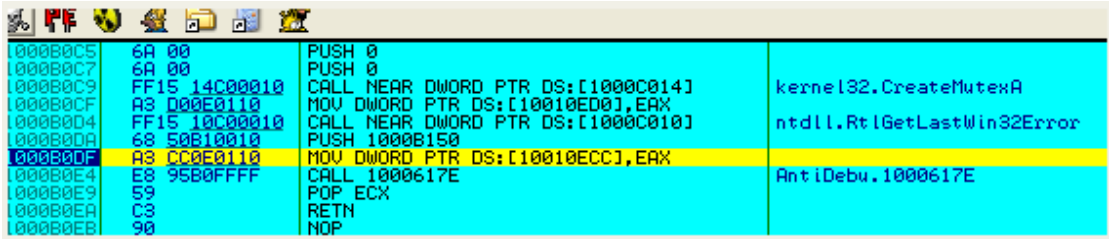


这里我们可以看到 LastErr 为 ERROR_ALREADY_EXISTS,即 0xB7。表示互斥体已经存在。



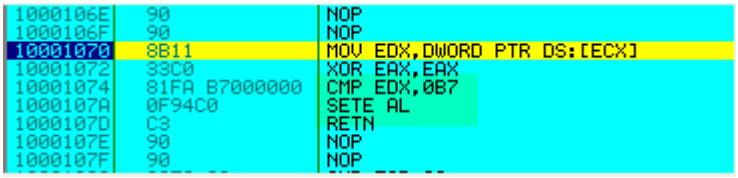
接着这里调用 RtlGetLastWin32Error 获取错误码,这里错误码我们已经知道了是 0xB7 了,即 ERROR_ALREADY_EXISTS。





这里将错误码保存到 10010ECC 指向的内存单元中,我们给该内存单元设置一个内存访问断点,运行起来。

断在了这里,这里是读取该错误码进行比较。



这里将 RtlGetLastWin32Error 返回的错误码与 0xB7 做比较。

接着如果相等的话,通过 SETE 指令将 EAX 置 1,这里明显错误码等于 0xB7,所以 EAX 被置 1,如果是父进程的话,EAX 会被置 0。

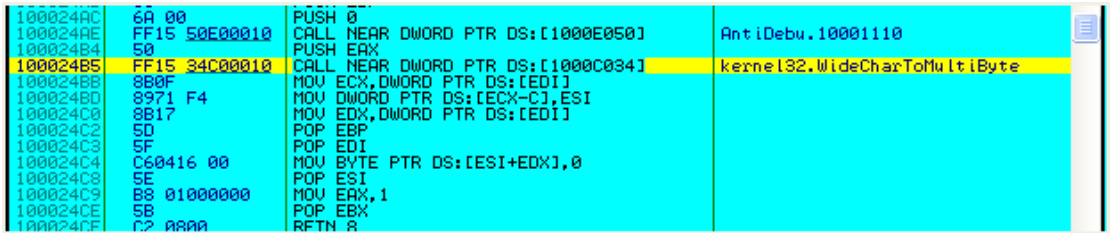
接着我们返回到了这里,子进程中 1000262C 这处条件跳转将成立,而父进程此处跳转将不会成立。所以之后父进程与子进程将执行不同的分支流程,有待我们进一步分析。

这里有没有觉得这样一点点单步跟踪有点麻烦?

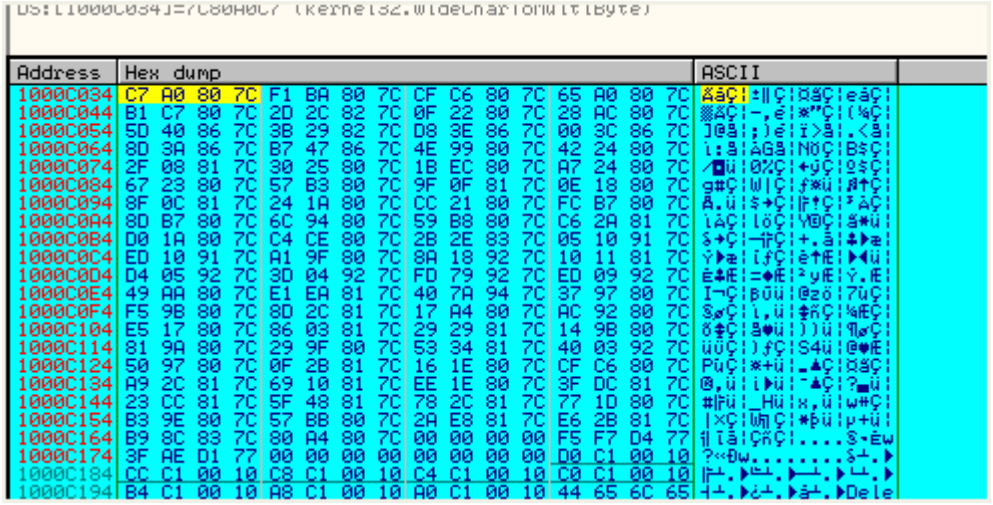
其实有网上很多好用的 API 监视工具,比如说 KAM,APISPY,这些工具可以记录程序中执行了哪些 API 函数,不需要我们一步步的跟踪,可以节省很多时间。

这里其实我们可以利用内存访问断点间接的完成 API 函数的监视工作,不需要这样一步步单步跟踪。具体操作如下:

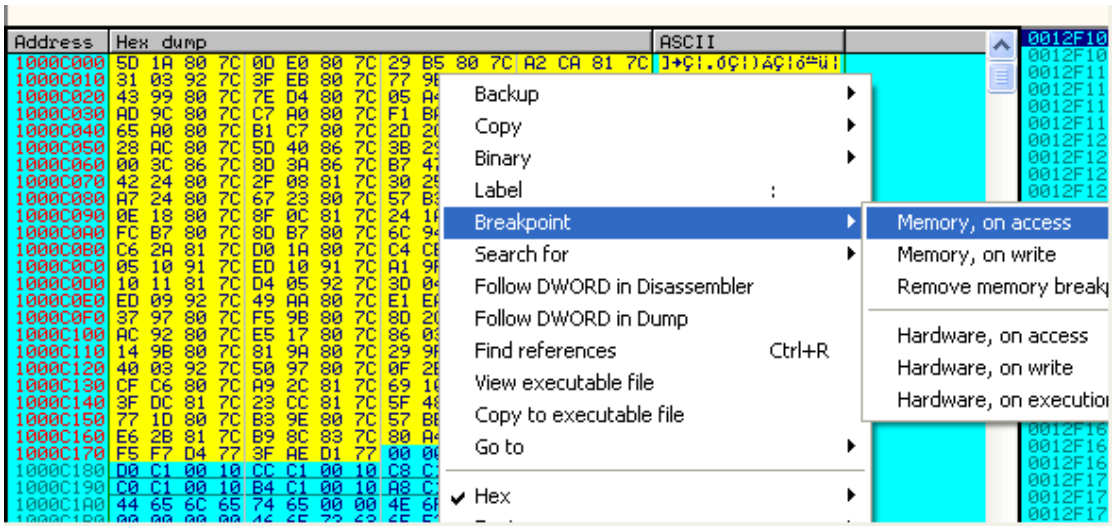
首先我们需要定位子进程中 AntiDebugDll.dll 这个模块的 IAT,这里我们将反汇编窗口往上拉,随便找一个 API 函数调用处。



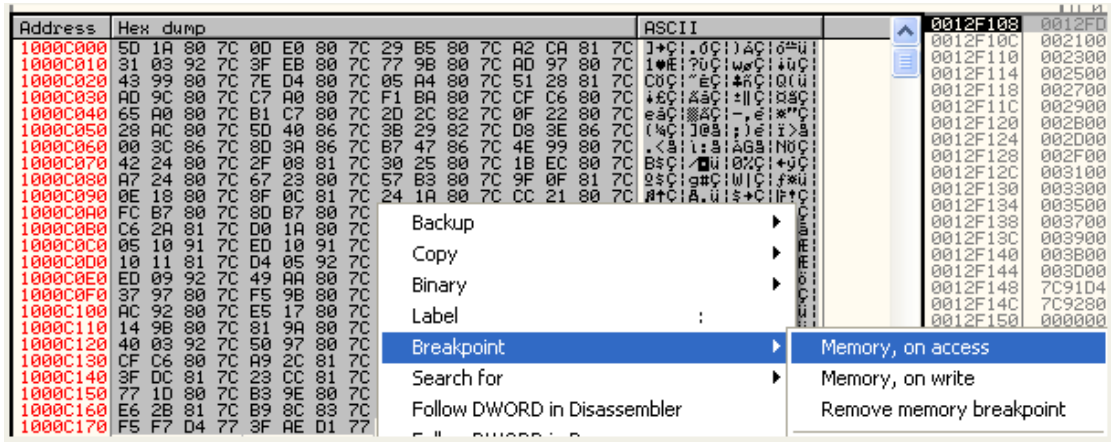
我们可以看到 100024B5 处调用了 WideCharToMultiByte 这个 API 函数,该 API 函数的指针位于 1000C034 地址处,它是 AntiDebugDll.dll 的 IAT 中的一项。我们在数据窗口中定位到该 IAT 项。



这里就是 IAT 了,我们定位到该 IAT 的起始位置和结束位置,然后选中整个 IAT,单击鼠标右键选择 Breakpoint-Memory,on access,给整个 IAT 设置内存访问断点。

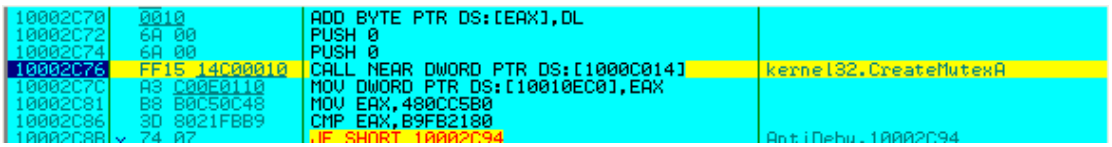


接下来我们同样给父进程的 AntiDebugDll.dll 的整个 IAT 表设置内存访问断点。

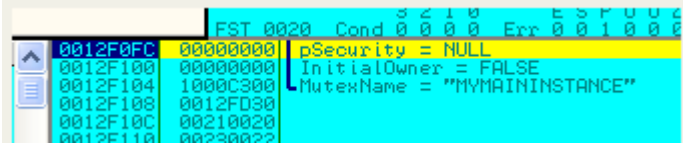


这里我们就给父子进程的 AntiDebugDll.dll 的 IAT 都设置了内存访问断点,也就说当程序中调用 AntiDebugDll.dll 模块 IAT 中的 API 函数时就会断下来,这里还需要注意一点,有可能该 Dll 会通过调用 GetProcAddress 来获取其他的 API 函数指针,所以这里我们给 GetProcAddress 也设置一个断点,以防万一。

好了,现在我们继续调试子进程,我们按 F9 键运行起来,看看调用哪些 API 函数。



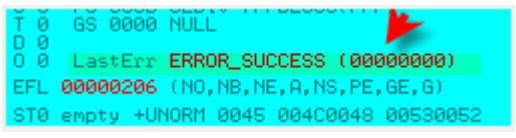
断在了这里,这里要创建 MYMAININSTANCE 这个互斥体,这个互斥体之前未被创建过。



大家应该还记得之前父进程创建的那个互斥体吧,叫做 MYFIRSTINSTANCE,顾名思义:"第一个实例"。

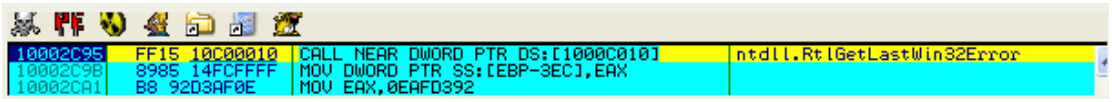
这里要创建的互斥体叫做 MYMAININSTANCE。顾名思义:"主体实例"。

我们按 F8 键执行该 API 函数。

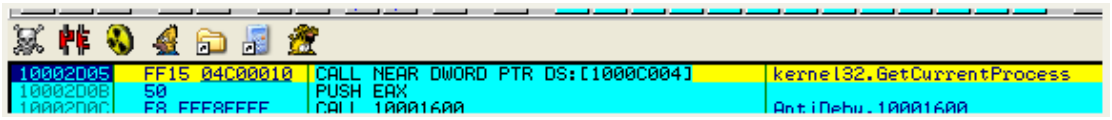


这里我们可以看到错误码为 ERROR_SUCCESS,说明互斥体创建成功。

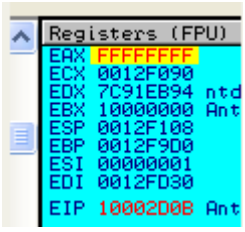
好,我们按 F9 键运行起来,看看下面会调用哪个 API 函数。



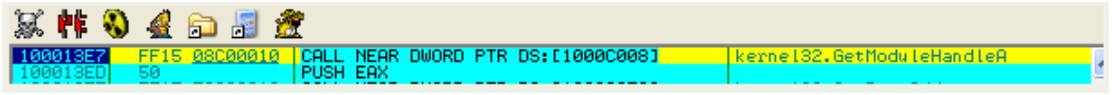
这里又是调用 RtlGetLastWin32Error 获取上次调用 CreateMutexA 这个 API 函数的错误码,我们继续按 F9 键运行。



这里是获取进程句柄,我们按 F8 键执行调用 API 函数,会返回(-1)FFFFFFFF。代表当前进程。该句柄不在句柄表中,不是真正的句柄,我们叫它伪句柄。

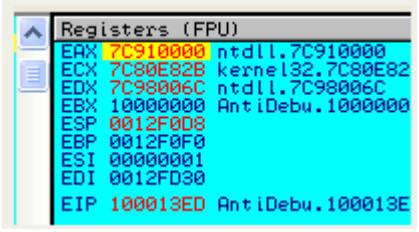


继续按 F9 键运行。

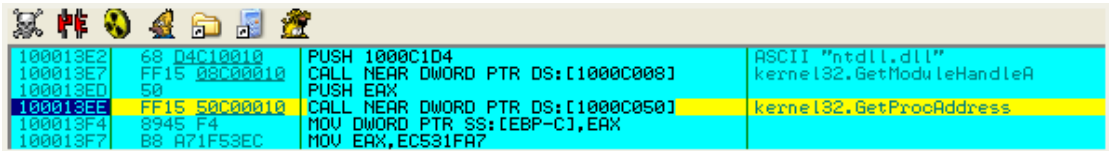


这里调用的是 GetModuleHandleA,这样的跟踪对于我们练习哪些不熟悉的 API 函数是一种极佳的锻炼。我们按 F8 键执行。

这里返回的是 NTDLL.DLL 这个模块的句柄。我猜测该程序会用该句柄作为 GetProcessAddress 的参数来获取 NTDLL.DLL 中包含的 API 函数指针。

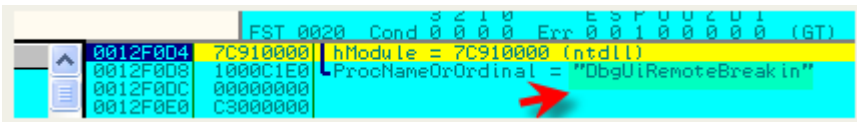


我希望我猜测是正确的,我们按 F9 键运行起来。

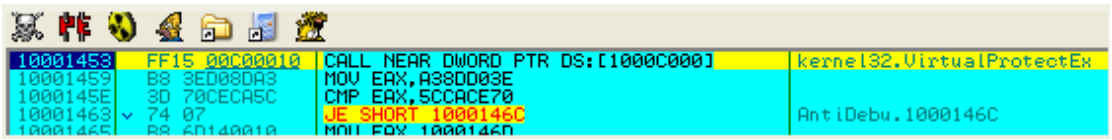


嘿嘿,看来我们的猜测是正确的。

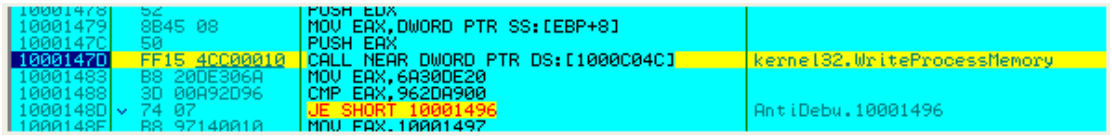
我们来看看其参数。



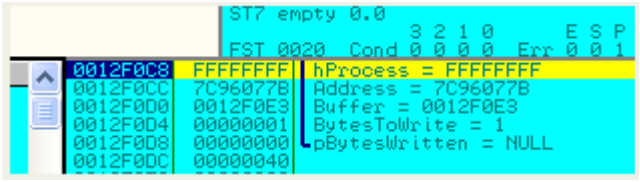
这里可以看到其要获取 DbgUiRemoteBreakin 这个 API 函数的指针。我们继续 F9 运行。



这里我们可以看出一点该程序的意图了,它想通过 Patch DbgUiRemoteBreakin 这个函数的实现代码来达到反调试的目的。这里是修改 DbgUiRemoteBreanin 这个 API 函数首字节的访问属性,将访问属性修改为可读可写可执行。

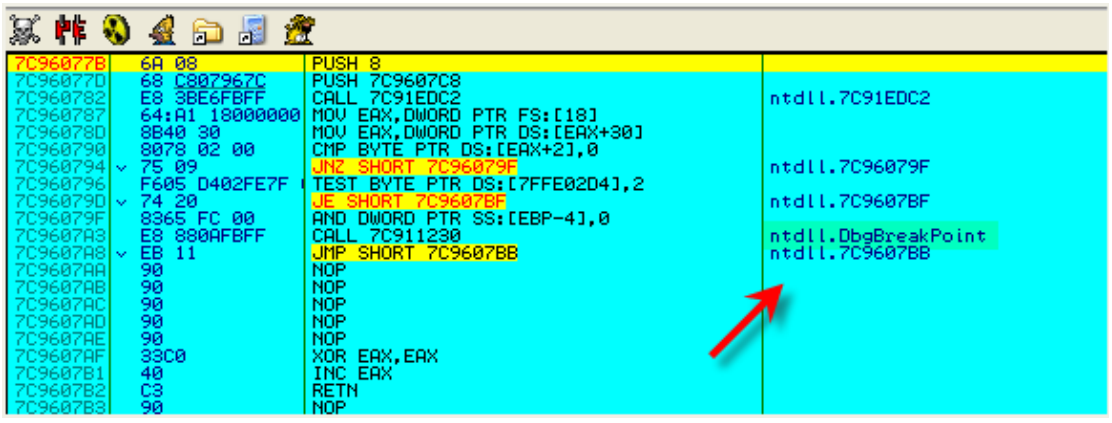


这里要调用 WriteProcessMemory 开始修改 DbgUiRemoteBreakin 的首字节了,我们来看看参数。

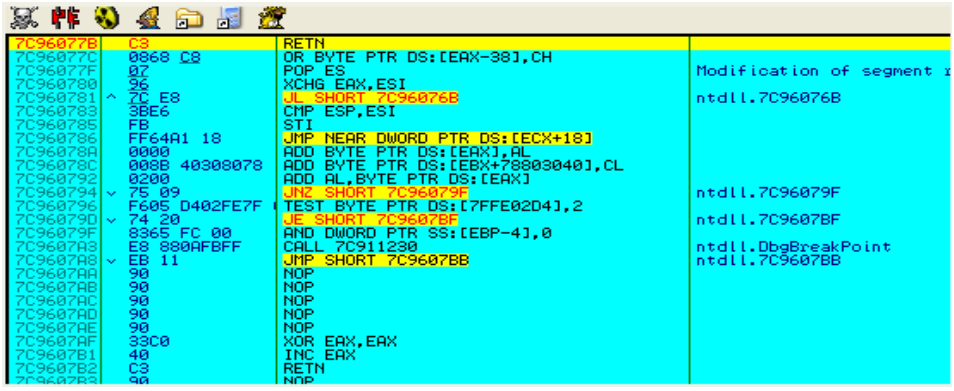


这里我们可以看到进程句柄为 FFFFFFFF,也就是当前进程,通过之前那个 GetCurrentProcess 获取到的。

这里我们来看看 DbgUiRemoteBreakin 的实现代码。

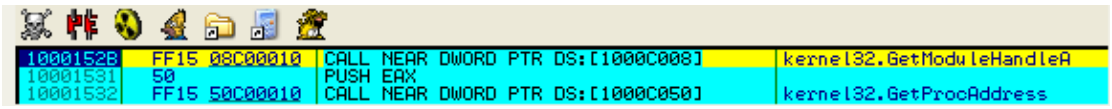


这里我们可以看到 DbgUiRemoteBreakin 中会调用 DbgBreakPoint,我们执行该 API 函数看看其将 DbgUiRemoteBreakin 的首字节修改为了什么。

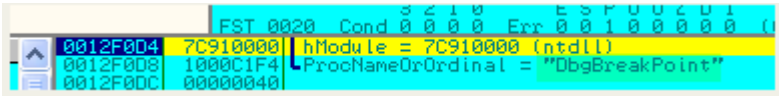


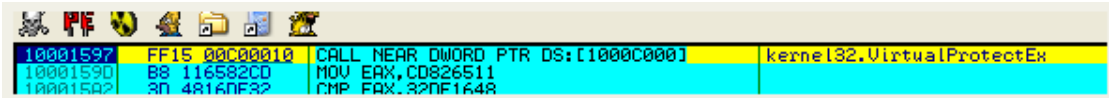
这里我们可以看到 DbgUiRemoteBreakin 的首字节被修改为了 RET 指令。

我们继续运行。

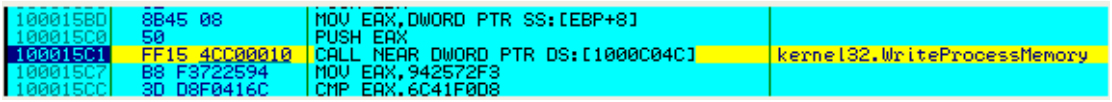


这里又是获取 NTDLL.DLL 的模块句柄,接着调用 GetProcAddress 获取 DbgBreakPoint 的函数指针,接着 Patch 之。

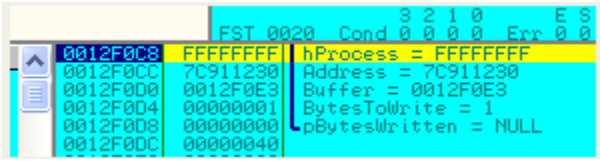




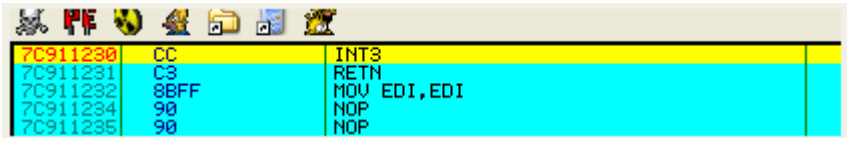
修改 DbgBreakPoint 首字节的访问权限。



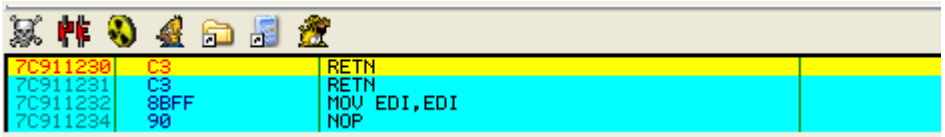
这里要 Patch DbgBreakPoint 首字节了,我们来看看参数。



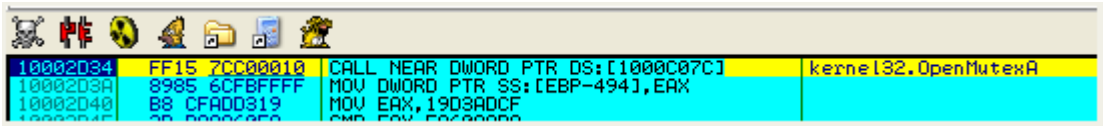
这里我们来看看 DbgBreakPoint 的实现代码。



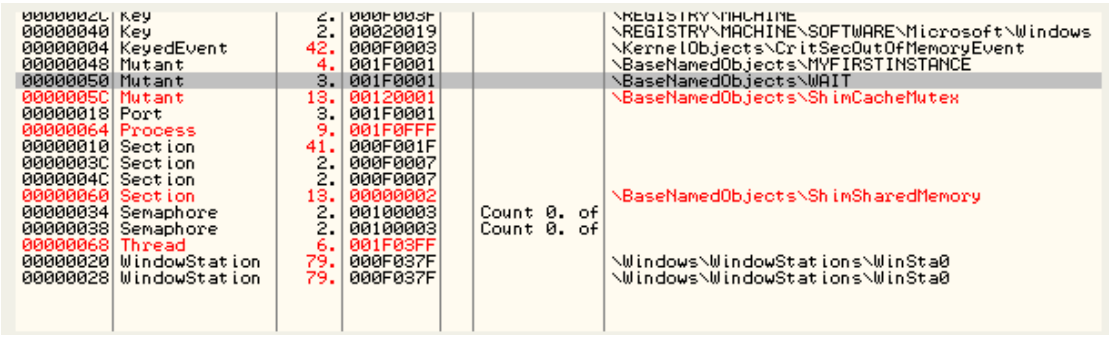
我们按 F8 键执行,看看 DbgBreakPoint 的首字节被修改为了什么。



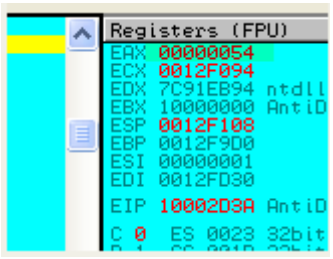
这里我们可以看到 DbgBreakPoint 的首字节被修改为了 RET。这样也可以达到反调试的目的。我们继续 F9 运行。



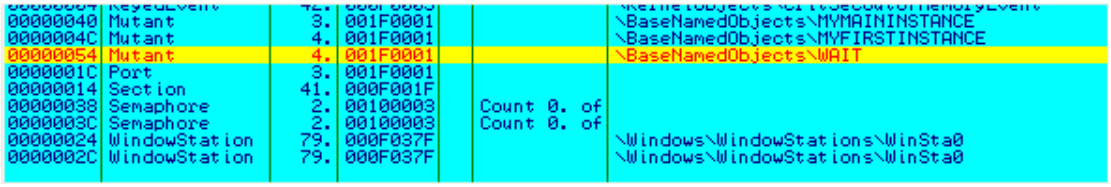
这里调用 OpenMutexA 这个 API 函数,打开名为 WAIT 的互斥体,大家应该还记得父进程已经创建了一个名为 WAIT 的互斥体吧,我们打开父进程所在 OD 的句柄列表窗口看看。



我们可以看到父进程中的确存在一个 WAIT 的互斥体,这里子进程中并没有调用 CreateMutexA,而是调用 OpenMutexA 来打开这个互斥体,这里可以顺利获取到之前父进程中创建的 WAIT 互斥体的句柄。



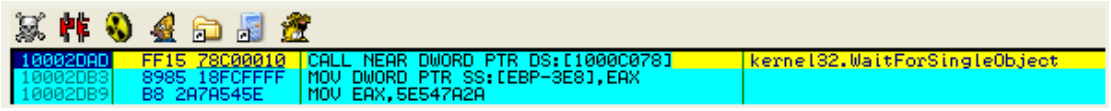
我们按 F8 键执行该函数,可以看到成功获取到了 WAIT 互斥体的句柄,为 0x54。



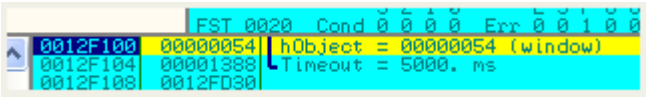
现在我们来看看子进程所在 OD 的句柄列表窗口。

恩,成功打开了 WAIT 这个互斥体。

我们继续 F9 键运行。

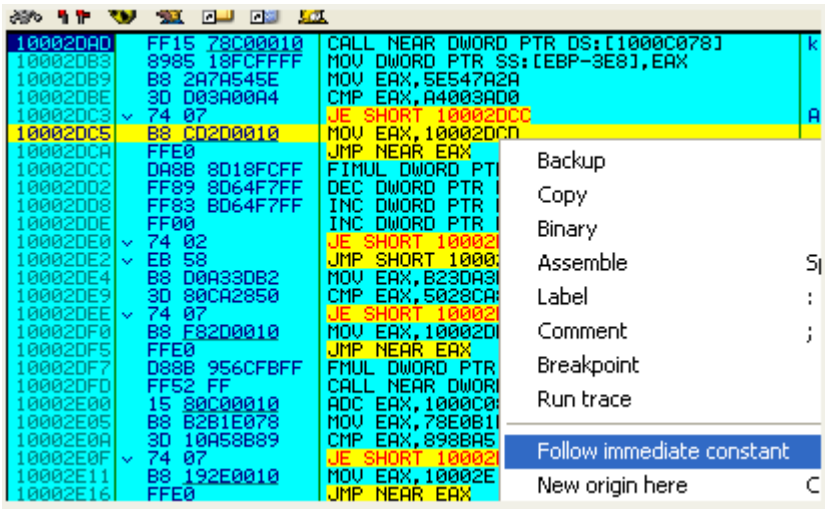


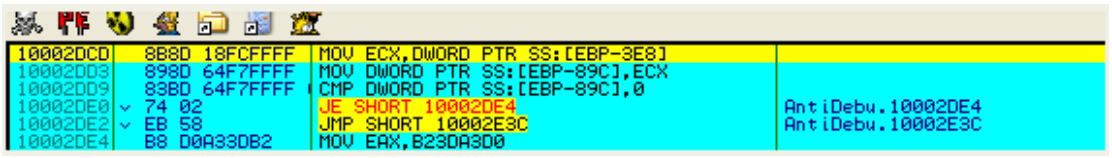
这里我们可以看到调用 WaitForSingleObject 这个函数进行父进程与子进程的同步处理。我们来看看其参数。



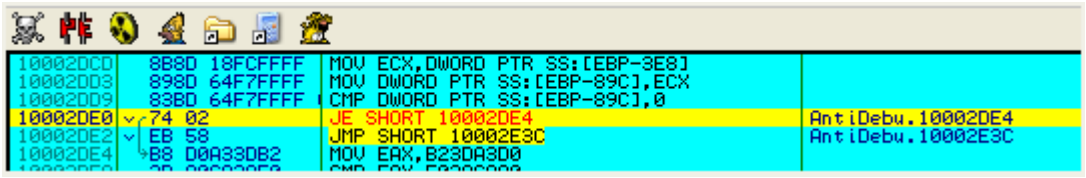
这里第一个参数为 WAIT 这个互斥体的句柄,第二个参数为 5000ms,即超时时间为 5 秒钟。也就是说子进程会等待父进程释放 WAIT 互斥体的信号量,如果超过 5 秒父进程还没有释放该函数就直接返回。

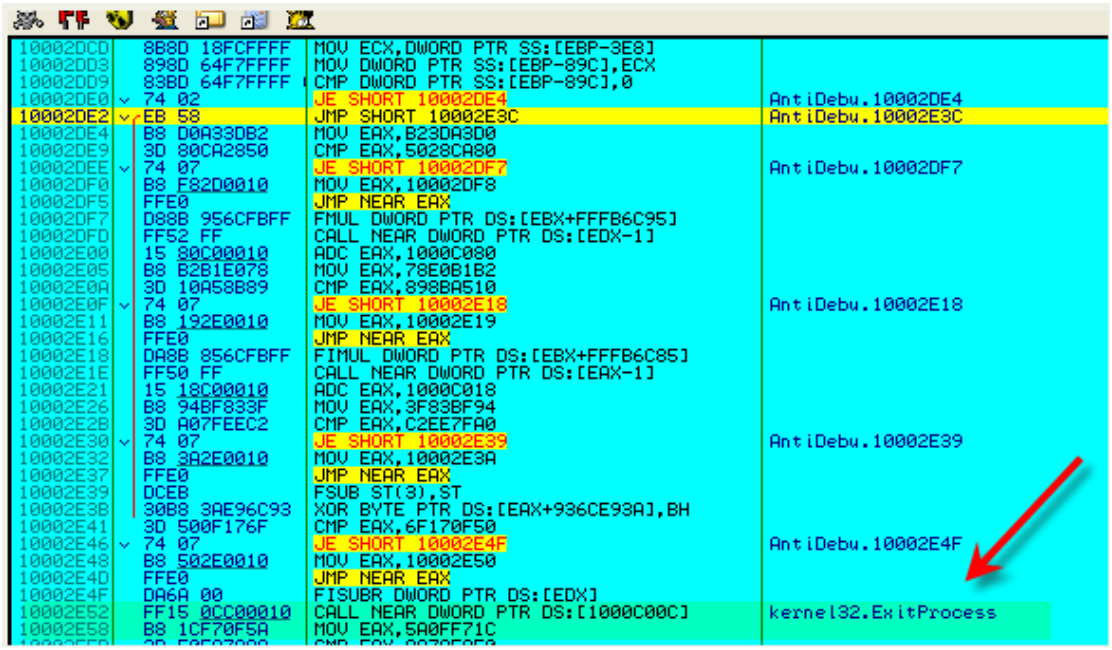
下面一条语句将 WaitForSingleObject 的返回值保存到变量中。



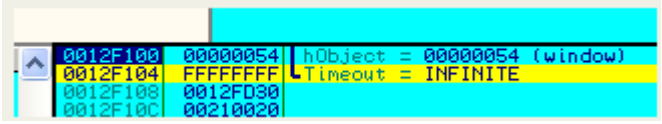


这里我们可以看到将 WaitForSingleObject 的返回值与零进行比较,如果等于零(即 WAIT_OBJECT_0:成功等到信号量释放)就可以绕过下面的 ExitProcess 的调用处。

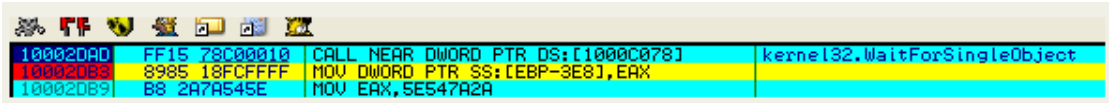




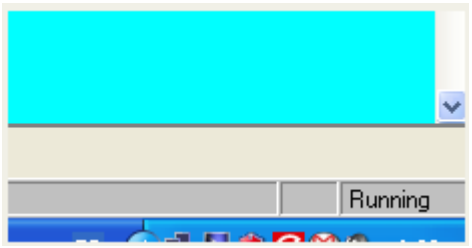
这里为了让子进程不因为 WaitForSingleObject 等待超时而退出,我们将 WaitForSingleObject 的超时时间修改为 INFINITE(无穷大)。



这里我们将 WaitForSingleObject 的第二个参数由 0x1388 修改为了 FFFFFFFF(-1),即 INFINITE。这里为了让父进程释放 WAIT 互斥体信号量的时候我们能够子进程能够顺利断下,我们给下一条指令处设置一个断点。



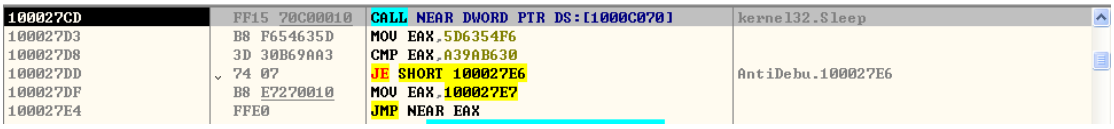
按 F9 键运行起来。



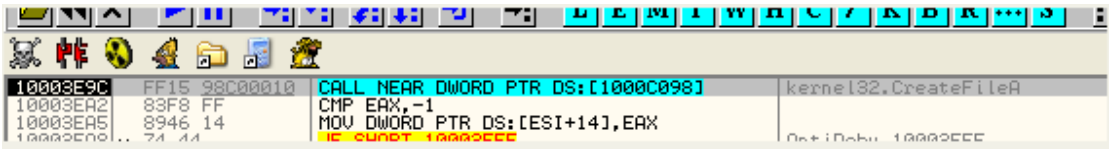
现在我们可以看到子进程已经运行起来了。也就是说子进程正在等待父进程释放 WAIT 这个互斥体的信号量。由于子进程所在 OD 的右下方现在显示的是 Running 状态,所以现在我们不能调试子进程了。转而我们现在来调试父进程。

大家应该还记得子进程中的 WaitForSingleObject 的超时时间为 5 秒吧?如果是 5 秒的话,那么子进程过了 5 秒,父进程还没有释放 WAIT 信号量,子进程就会调用 ExitProcess 退出了。所以我们将超时时间修改为了 FFFFFFFF(即 INFINITE),那么子进程只能乖乖的一直等待,直到父进程释放 WAIT 信号量为止,嘿嘿。

现在我们来调试父进程,我们已经对父进程中 AntiDebugDll.dll 模块的整个 IAT 设置内存访问断点了,所以我们直接运行起来,看看会调用哪些 API 函数。



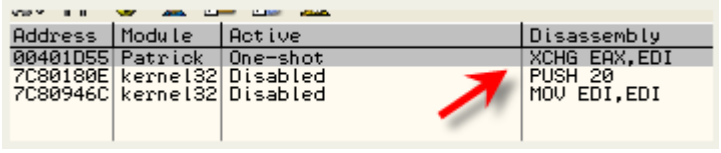
我们可以看到这里调用 Sleep 休眠片刻,我们继续 F9 运行起来,看看还会调用其他的什么重要的 API 函数。



如果断在不是很重要的 API 函数处的话,我们继续运行。

我们可以看到这里调用 CreateFileA 打开文件,有点可疑。我们来看看其参数。

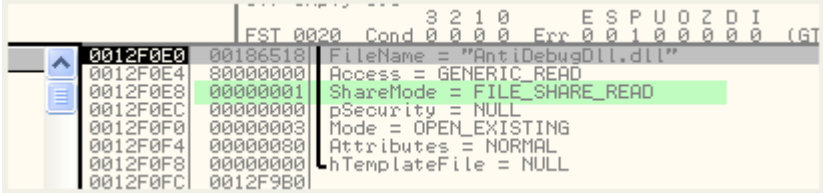
这里调用 CreateFileA 要打开 AntiDebugDll.dll,很可能要检测 AntiDebugDll.dll 中的代码是否被修改,防止被下 INT 3 断点。所以我们这里我们将之前设置的 INT 3 都删除掉。



这里我们将断点列表窗口中的断点都删除掉。

下面我们就不设置 INT 3 断点了,如果实在需要设置断点,我们就用内存断点替代。

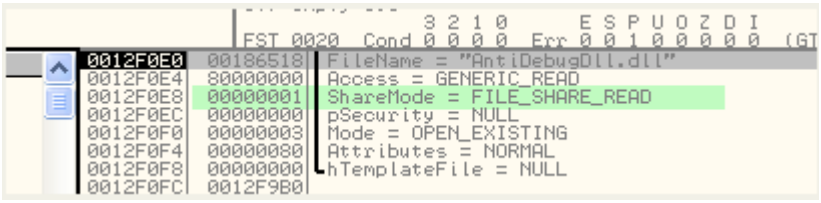
我们来看 CreateFileA 的参数情况:



这里明显这个程序想通过 CreateFileA 打开 AntiDebugDll.dll,然后通过 ReadFile 读取相应字节码进行比较,看看是否被修改。这里

我们并没有修改 Dll 文件中的内容,我们修改的都是内存中的内容。

继续看其他参数。



这里大部分参数我们都不是很关系,我们只需要注意一下 dwShareMode 这个参数,共享模式为 FILE_SHARE_READ,我们来看一下 MSDN 中的解释。

dwShareMode

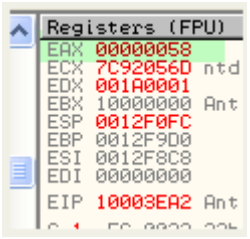
Set of bit flags that specifies how the object can be shared. If *dwShareMode* is 0, the object cannot be shared. Subsequent open operations on the object will fail, until the handle is closed.

To share the object, use a combination of one or more of the following values:

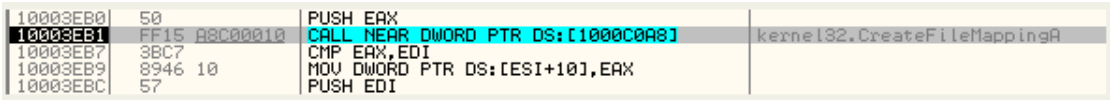
Value	Meaning
FILE_SHARE_DELETE	**Windows NT only:** Subsequent open operations on the object will succeed only if delete access is requested.
FILE_SHARE_READ	Subsequent open operations on the object will succeed only if read access is requested.
FILE_SHARE_WRITE	Subsequent open operations on the object will succeed only if write access is requested.

如果是读取操作的话,会返回成功。

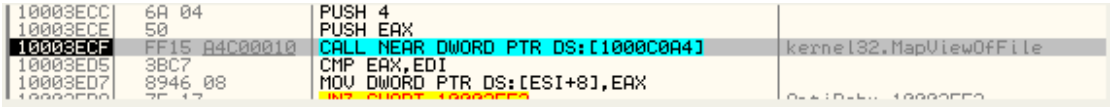
我们按 F8 键执行,可以看到返回的句柄值。



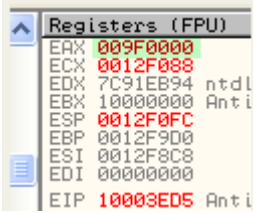
这里打开的 AntiDebugDll.dll 的文件句柄为 0x58。我们继续执行,看看还会调用什么 API 函数。



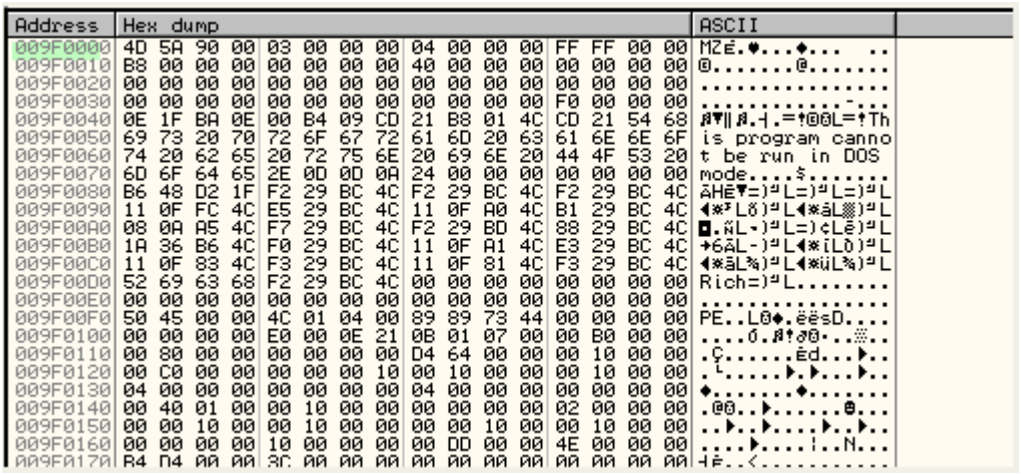
这里调用的是 CreateFileMappingA 这个 API 函数,创建一个文件映射。



接着这里调用 MapViewOfFile 将文件的内容映射到内存中,我们按 F8 键执行该函数,看看被映射的地址是多少。

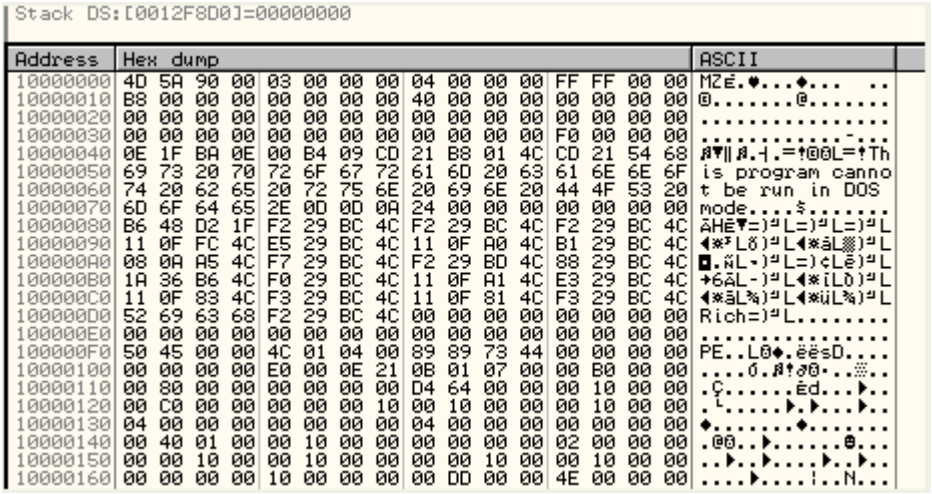


这里 AntiDebugDll.dll 就被映射到了内存中,起始地址为9F0000,接着它将会怎么做呢?将所有的字节都逐一比较吗?不,它没有,它仅仅只比较了起始的几个字节。我们在数据窗口中定位到 9F0000 地址处。



Address	Hex dump	ASCII
009F0000	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZé.♥...♦... ..
009F0010	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00	©.......@.......
009F0020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
009F0030	00 00 00 00 00 00 00 00 00 00 00 00 F0 00 00 00	............-...
009F0040	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	♫▼‖♫.┤.=↑@@L=↑Th
009F0050	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
009F0060	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
009F0070	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00	mode....$.......
009F0080	B6 48 D2 1F F2 29 BC 4C F2 29 BC 4C F2 29 BC 4C	╢HÒ▼=)╝L=)╝L=)╝L
009F0090	11 0F FC 4C E5 29 BC 4C 11 0F A0 4C B1 29 BC 4C	◄☼ⁿLσ)╝L◄☼áL▒)╝L
009F00A0	08 0A A5 4C F7 29 BC 4C F2 29 BD 4C 88 29 BC 4C	◘.ñL-)╝L=)¢Lë)╝L
009F00B0	1A 36 B6 4C F0 29 BC 4C 11 0F A1 4C E3 29 BC 4C	→6╢L-)╝L◄☼íLδ)╝L
009F00C0	11 0F 83 4C F3 29 BC 4C 11 0F 81 4C F3 29 BC 4C	◄☼âL%)╝L◄☼üL%)╝L
009F00D0	52 69 63 68 F2 29 BC 4C 00 00 00 00 00 00 00 00	Rich=)╝L........
009F00E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
009F00F0	50 45 00 00 4C 01 04 00 89 89 73 44 00 00 00 00	PE..L☺♦.ëësD....
009F0100	00 00 00 00 E0 00 0E 21 0B 01 07 00 00 B0 00 00	....ó.♫!♪☺0...▒..
009F0110	00 80 00 00 00 00 00 00 D4 64 00 00 00 10 00 00	.Ç.......Ôd...►..
009F0120	00 C0 00 00 00 00 00 10 00 10 00 00 00 10 00 00	.└......►.►...►..
009F0130	04 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00	♦.......♦.......
009F0140	00 40 01 00 00 10 00 00 00 00 00 00 02 00 00 00	.@☺.►.......☻...
009F0150	00 00 10 00 00 10 00 00 00 00 10 00 00 10 00 00	..►..►....►..►..
009F0160	00 00 00 00 10 00 00 00 00 DD 00 00 4E 00 00 00	....►....¦..N...
009F0170	B4 D4 A0 A0 3C A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0	┤╘..<.....

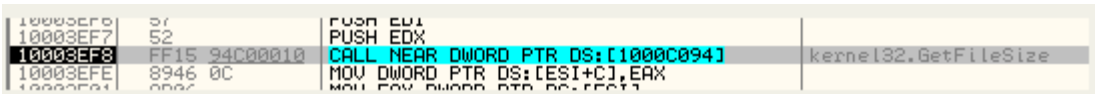
而我们当前正在运行的 AntiDebugDll.dll 的基地址为 10000000,我们再到数据窗口中定位到该地址,我们可以看到两者的内容是一致的。



Stack DS:[0012F8D0]=00000000

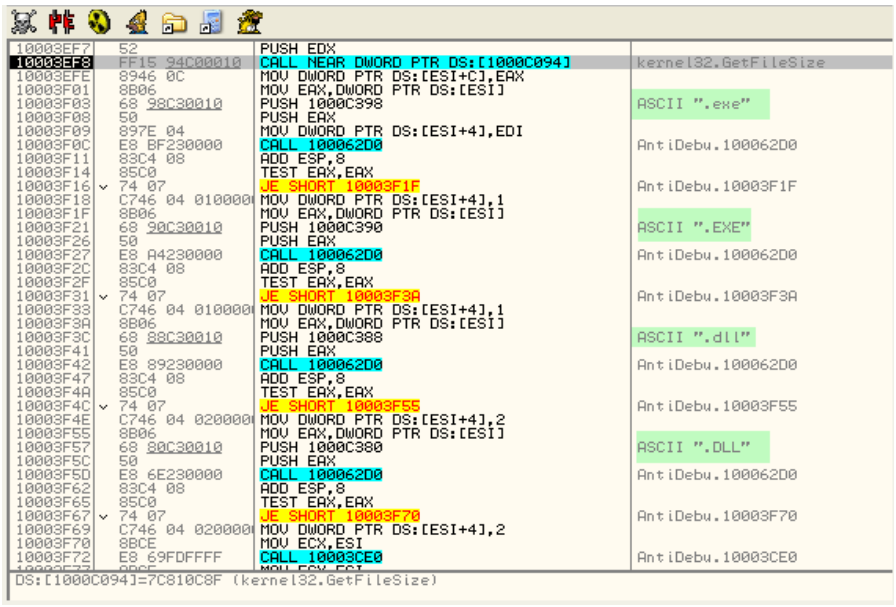
Address	Hex dump	ASCII
10000000	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZé.♥...♦... ..
10000010	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00	©.......@.......
10000020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
10000030	00 00 00 00 00 00 00 00 00 00 00 00 F0 00 00 00	............-...
10000040	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	♫▼‖♫.┤.=↑@@L=↑Th
10000050	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
10000060	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
10000070	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00	mode....$.......
10000080	B6 48 D2 1F F2 29 BC 4C F2 29 BC 4C F2 29 BC 4C	╢HÒ▼=)╝L=)╝L=)╝L
10000090	11 0F FC 4C E5 29 BC 4C 11 0F A0 4C B1 29 BC 4C	◄☼ⁿLσ)╝L◄☼áL▒)╝L
100000A0	08 0A A5 4C F7 29 BC 4C F2 29 BD 4C 88 29 BC 4C	◘.ñL-)╝L=)¢Lë)╝L
100000B0	1A 36 B6 4C F0 29 BC 4C 11 0F A1 4C E3 29 BC 4C	→6╢L-)╝L◄☼íLδ)╝L
100000C0	11 0F 83 4C F3 29 BC 4C 11 0F 81 4C F3 29 BC 4C	◄☼âL%)╝L◄☼üL%)╝L
100000D0	52 69 63 68 F2 29 BC 4C 00 00 00 00 00 00 00 00	Rich=)╝L........
100000E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
100000F0	50 45 00 00 4C 01 04 00 89 89 73 44 00 00 00 00	PE..L☺♦.ëësD....
10000100	00 00 00 00 E0 00 0E 21 0B 01 07 00 00 B0 00 00	....ó.♫!♪☺0...▒..
10000110	00 80 00 00 00 00 00 00 D4 64 00 00 00 10 00 00	.Ç.......Ôd...►..
10000120	00 C0 00 00 00 00 00 10 00 10 00 00 00 10 00 00	.└......►.►...►..
10000130	04 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00	♦.......♦.......
10000140	00 40 01 00 00 10 00 00 00 00 00 00 02 00 00 00	.@☺.►.......☻...
10000150	00 00 10 00 00 10 00 00 00 00 10 00 00 10 00 00	..►..►....►..►..
10000160	00 00 00 00 10 00 00 00 00 DD 00 00 4E 00 00 00	....►....¦..N...

我们继续 F9 键运行。

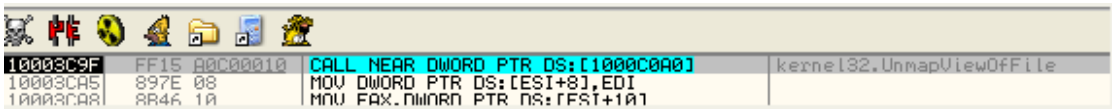


10003EF6	57	PUSH EDI	
10003EF7	52	PUSH EDX	
10003EF8	FF15 94C00010	CALL NEAR DWORD PTR DS:[1000C094]	kernel32.GetFileSize
10003EFE	8946 0C	MOV DWORD PTR DS:[ESI+C],EAX	
10003F01	8B86	MOV EAX,DWORD PTR DS:[ESI]	

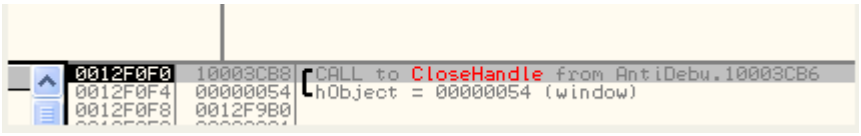
这里是读取 DLL 文件的大小然后校验,下面接着判断文件扩展名是否为,DLL,dll,EXE,exe。



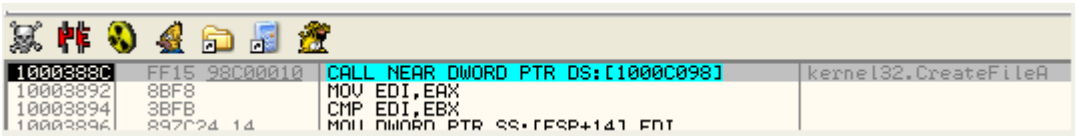
继续:



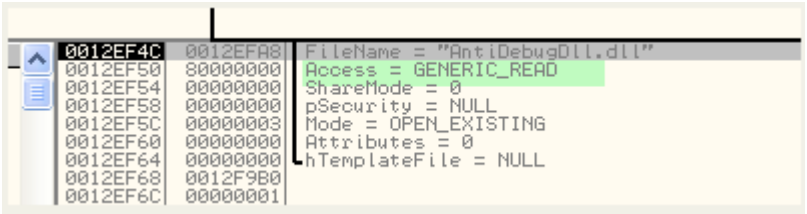
删除 9F0000 地址处的内存映射,说明不再需要这些字节了。



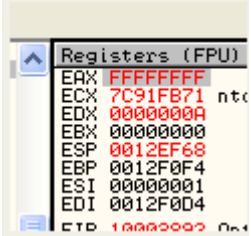
关闭掉文件句柄。



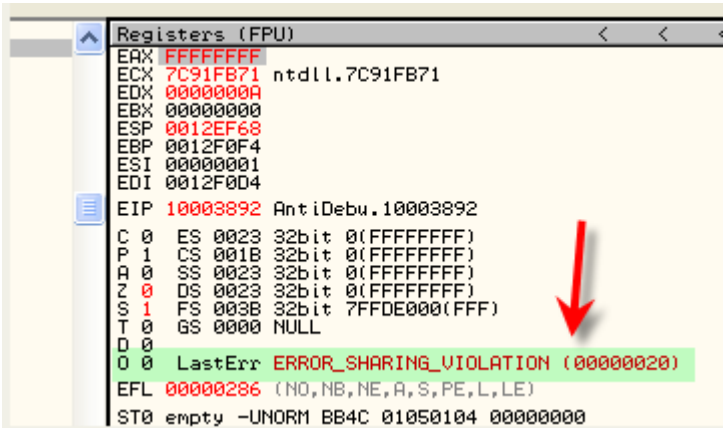
这里又调用了 CreateFileA,参数如下:



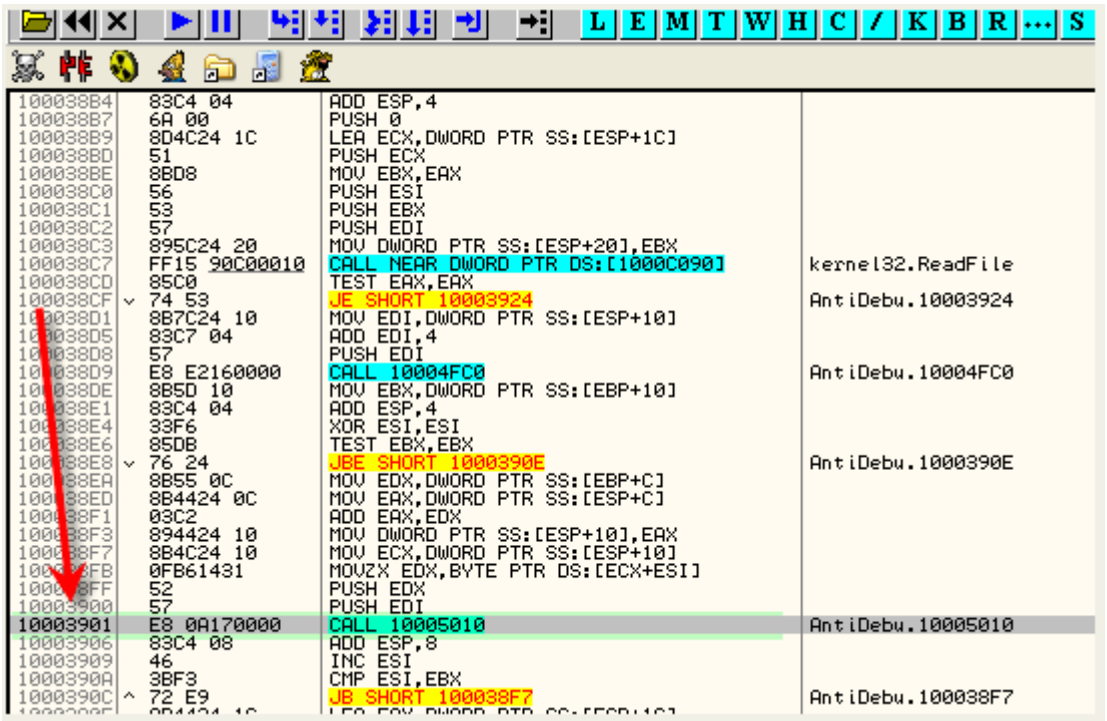
dwDesiredAccess 的值为 GENERIC_READ,但是 dwShareMode 的值被设置为了 0。MSDN 中的解释是如果 dwShareMode 被设置为零,该句柄不共享,再关闭之前不能被再次打开。这里如果我们按 F8 键执行的话,将打开文件失败。EAX 返回-1(即 FFFFFFFF)。



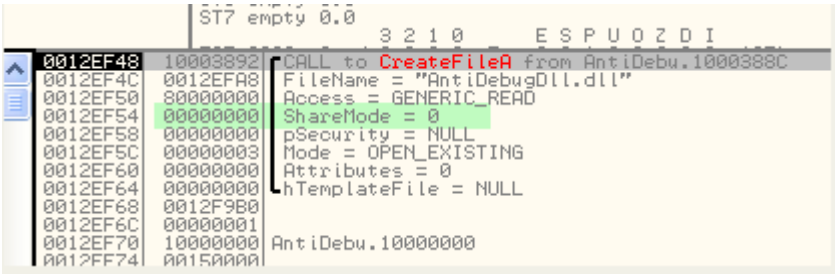
这里我们可以看到返回的句柄为 FFFFFFFF,即打开文件失败,LastErr 为 ERROR_SHARING_VIOLATION。



继续。我们可以看到下面这部分代码,如果刚刚读取文件打开文件成功的话,那么接着将读取文件成功。接着这里会对读取到的文件内容进行处理,有可能是解密代码哟!下面 10003901 地址处调用的 10005010 是解密的核心部分。



这里刚刚由于打开文件失败,所以并不会执行刚刚解密操作,这里我们需要重来一遍前面的步骤,回到刚刚 CreateFileA 处。



我们将 dwShareMode 修改为 FILE_SHARE_READ,即 1,让打开文件成功。接下来会读取文件,然后对读取到的文件内容进行相应的处理,我们来看看对文件内容处理以后后续流程会有什么影响。

```
                                                3 2 1 0      E S P U O Z D I
0012EF48  10003892  CALL to CreateFileA from AntiDebu.1000388C
0012EF4C  0012EFA8  FileName = "AntiDebugDll.dll"
0012EF50  80000000  Access = GENERIC_READ
0012EF54  00000001  ShareMode = FILE_SHARE_READ
0012EF58  00000000  pSecurity = NULL
0012EF5C  00000003  Mode = OPEN_EXISTING
0012EF60  00000000  Attributes = 0
0012EF64  00000000  hTemplateFile = NULL
0012EF68  0012F9B0
```

按 F8 键执行。

```
Registers (FPU)
EAX 00000058
ECX 7C92056D ntdll.
EDX 00150608
EBX 00000000
ESP 0012EF68
EBP 0012F0F4
ESI 00000001
EDI 0012F0D4
EIP 10003892 AntiDe
```

这里获取到了句柄,我们继续。

```
100038C7   FF15 90C00010   CALL NEAR DWORD PTR DS:[1000C090]   kernel32.ReadFile
100038CD   85C0            TEST EAX,EAX
100038CF   74 53           JE SHORT 10003924                   AntiDebu.10003924
100038D1   8B7C24 10       MOV EDI,DWORD PTR SS:[ESP+10]
```

又是 ReadFile,按 F8 键。

```
Registers (FPU)
EAX 00000001
ECX 7C801898 kerne
EDX 7C91EB94 ntdll
EBX 009F0048
ESP 0012EF68
EBP 0012F0F4
ESI 00013000
EDI 00000058
EIP 100038CD AntiD
```

这里 EAX 返回 1,接下来会对读取到的文件内容进行相应的处理,很可能是解密处理。

```
100038B7   6A 00           PUSH 0
100038B9   8D4C24 1C       LEA ECX,DWORD PTR SS:[ESP+1C]
100038BD   51              PUSH ECX
100038BE   8BD8            MOV EBX,EAX
100038C0   56              PUSH ESI
100038C1   53              PUSH EBX
100038C2   57              PUSH EDI
100038C3   895C24 20       MOV DWORD PTR SS:[ESP+20],EBX
100038C7   FF15 90C00010   CALL NEAR DWORD PTR DS:[1000C090]   kernel32.ReadFile
100038CD   85C0            TEST EAX,EAX
100038CF   74 53           JE SHORT 10003924                   AntiDebu.10003924
100038D1   8B7C24 10       MOV EDI,DWORD PTR SS:[ESP+10]
100038D5   83C7 04         ADD EDI,4
100038D8   57              PUSH EDI
100038D9   E8 E21F0000      CALL 10004FC0                      AntiDebu.10004FC0
100038DE   8B5D 10         MOV EBX,DWORD PTR SS:[EBP+10]
100038E1   83C4 04         ADD ESP,4
100038E4   33F6            XOR ESI,ESI
100038E6   85DB            TEST EBX,EBX
100038E8   76 24           JBE SHORT 1000390E                  AntiDebu.1000390E
100038EA   8B55 0C         MOV EDX,DWORD PTR SS:[EBP+C]
100038ED   8B4424 0C       MOV EAX,DWORD PTR SS:[ESP+C]
100038F1   03C2            ADD EAX,EDX
100038F3   894424 10       MOV DWORD PTR SS:[ESP+10],EAX
100038F7   8B4C24 10       MOV ECX,DWORD PTR SS:[ESP+10]
100038FB   0FB61431        MOVZX EDX,BYTE PTR DS:[ECX+ESI]
100038FF   52              PUSH EDX
10003900   57              PUSH EDI
10003901   E8 0A170000      CALL 10005010                      AntiDebu.10005010
10003906   83C4 08         ADD ESP,8
10003909   46              INC ESI
1000390A   3BF3            CMP ESI,EBX
1000390C   72 F9           JB SHORT 100038E7                   AntiDebu.100038E7
```

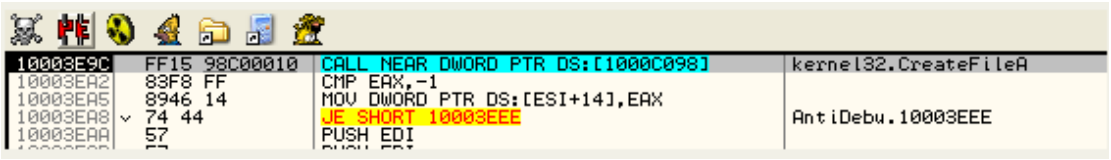
NoSALTA

我们继续按 F9 键运行。

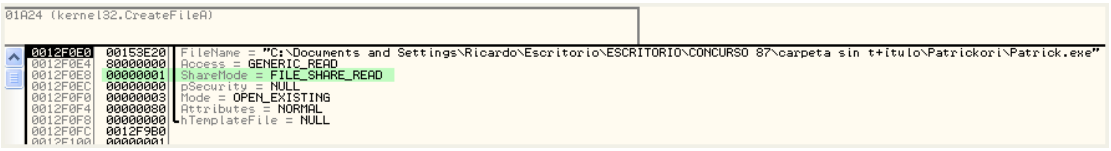
```
10003921   83C4 08         ADD ESP,8
10003924   57              PUSH EDI
10003925   FF15 18C00010   CALL NEAR DWORD PTR DS:[1000C018]   kernel32.CloseHandle
1000392B   53              PUSH EBX
1000392C   E8 5F280000      CALL 10006190                      AntiDebu.10006190
10003931   83C4 04         ADD ESP,4
10003934   33DB            XOR EBX,EBX
```

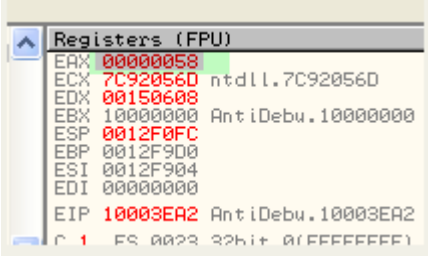
关闭句柄。

这里又到了 CreateFileA 处。

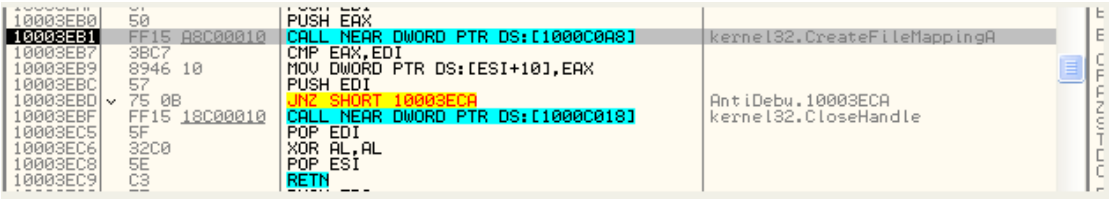


我们会发现该程序多次调用 CreateFileA,但是要注意了,这里要打开的 Patrick.exe 这个文件。

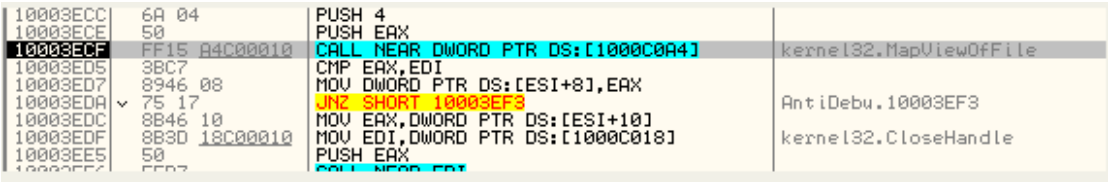




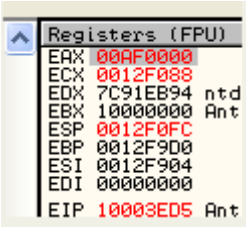
这里由于是第一次打开,所以成功获取到了文件句柄。



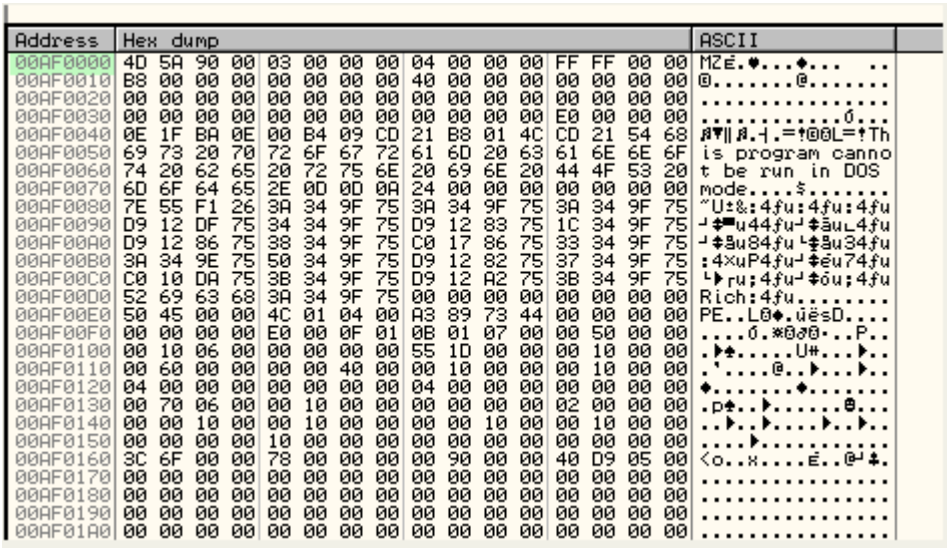
现在又要对 Patrick.exe 创建文件映射了,跟之前 AntiDebugDll.dll 的类似。



创建内存映射,映射到了 AF0000 这个地址处。

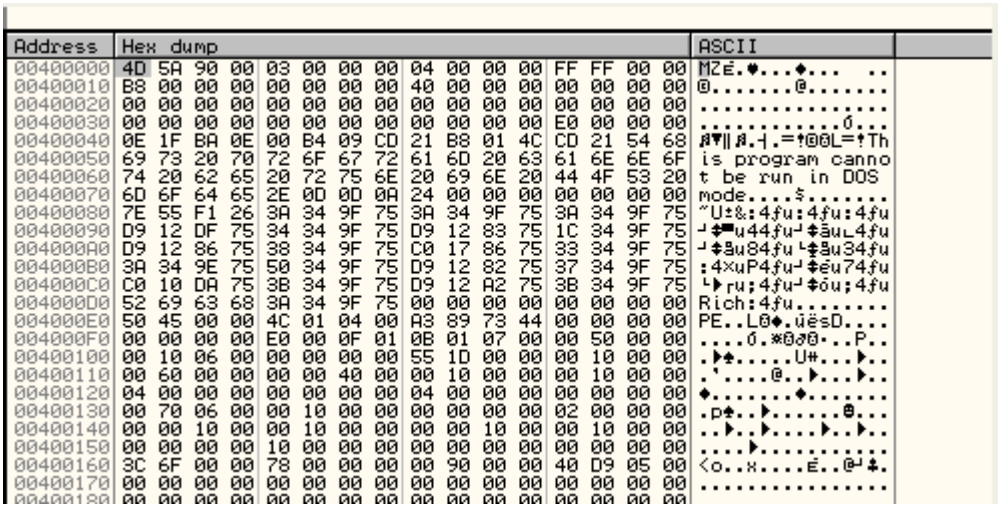


这里我们在数据窗口中定位 AF0000 地址处,可以看到 Patrick.exe 的 PE 头部情况。



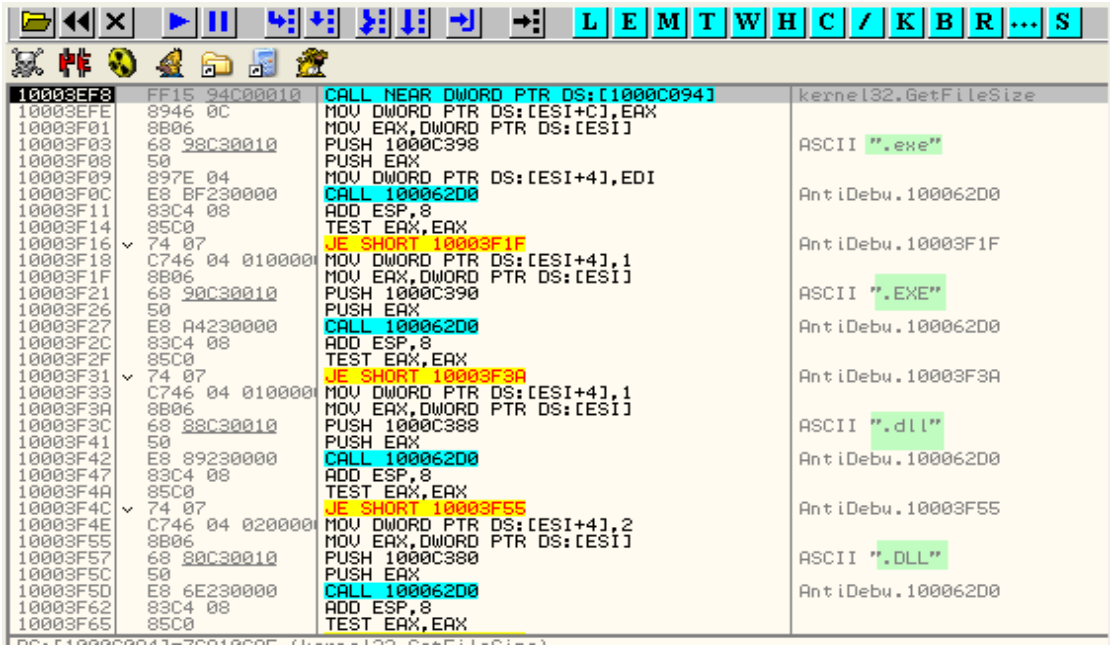
Address	Hex dump	ASCII
00AF0000	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZÉ.♥...♦... ..
00AF0010	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00	®.......@.......
00AF0020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00AF0030	00 00 00 00 00 00 00 00 00 00 00 00 E0 00 00 00	............ó...
00AF0040	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	♫▼‖♫.┤.=↑@@L=↑Th
00AF0050	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
00AF0060	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
00AF0070	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00	mode....$.......
00AF0080	7E 55 F1 26 3A 34 9F 75 3A 34 9F 75 3A 34 9F 75	~Uñ&:4ƒu:4ƒu:4ƒu
00AF0090	D9 12 DF 75 34 34 9F 75 D9 12 83 75 1C 34 9F 75	┘↕▀u44ƒu┘↕âu∟4ƒu
00AF00A0	D9 12 86 75 38 34 9F 75 C0 17 86 75 33 34 9F 75	┘↕åu84ƒu└↨åu34ƒu
00AF00B0	3A 34 9E 75 50 34 9F 75 D9 12 82 75 37 34 9F 75	:4×uP4ƒu┘↕éu74ƒu
00AF00C0	C0 10 DA 75 3B 34 9F 75 D9 12 A2 75 3B 34 9F 75	└►ru;4ƒu┘↕óu;4ƒu
00AF00D0	52 69 63 68 3A 34 9F 75 00 00 00 00 00 00 00 00	Rich:4ƒu........
00AF00E0	50 45 00 00 4C 01 04 00 A3 89 73 44 00 00 00 00	PE..L☺♦.ûësD....
00AF00F0	00 00 00 00 E0 00 0F 01 0B 01 07 00 00 50 00 00	....ó.♫☺•☺...P..
00AF0100	00 10 06 00 00 00 00 00 55 1D 00 00 00 10 00 00	.►♠.....U#...►..
00AF0110	00 60 00 00 00 00 40 00 00 10 00 00 00 10 00 00	.`.....@..►...►..
00AF0120	04 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00	♦.......♦.......
00AF0130	00 70 06 00 00 10 00 00 00 00 00 00 02 00 00 00	.p♠..►.....☻...
00AF0140	00 00 10 00 00 10 00 00 00 00 10 00 00 10 00 00	..►..►....►..►..
00AF0150	00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00	....►...........
00AF0160	3C 6F 00 00 78 00 00 00 00 90 00 00 40 D9 05 00	<o..x....É..@┘♣.
00AF0170	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00AF0180	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00AF0190	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00AF01A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....

我们再在数据窗口中定位当前程序的基地址 400000。



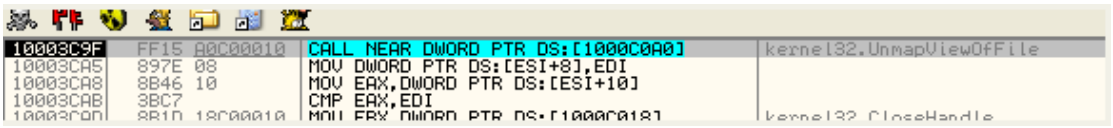
Address	Hex dump	ASCII
00400000	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZÉ.♥...♦... ..
00400010	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00	®.......@.......
00400020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00400030	00 00 00 00 00 00 00 00 00 00 00 00 E0 00 00 00	............ó...
00400040	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	♫▼‖♫.┤.=↑@@L=↑Th
00400050	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
00400060	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
00400070	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00	mode....$.......
00400080	7E 55 F1 26 3A 34 9F 75 3A 34 9F 75 3A 34 9F 75	~Uñ&:4ƒu:4ƒu:4ƒu
00400090	D9 12 DF 75 34 34 9F 75 D9 12 83 75 1C 34 9F 75	┘↕▀u44ƒu┘↕âu∟4ƒu
004000A0	D9 12 86 75 38 34 9F 75 C0 17 86 75 33 34 9F 75	┘↕åu84ƒu└↨åu34ƒu
004000B0	3A 34 9E 75 50 34 9F 75 D9 12 82 75 37 34 9F 75	:4×uP4ƒu┘↕éu74ƒu
004000C0	C0 10 DA 75 3B 34 9F 75 D9 12 A2 75 3B 34 9F 75	└►ru;4ƒu┘↕óu;4ƒu
004000D0	52 69 63 68 3A 34 9F 75 00 00 00 00 00 00 00 00	Rich:4ƒu........
004000E0	50 45 00 00 4C 01 04 00 A3 89 73 44 00 00 00 00	PE..L☺♦.ûësD....
004000F0	00 00 00 00 E0 00 0F 01 0B 01 07 00 00 50 00 00	....ó.♫☺•☺...P..
00400100	00 10 06 00 00 00 00 00 55 1D 00 00 00 10 00 00	.►♠.....U#...►..
00400110	00 60 00 00 00 00 40 00 00 10 00 00 00 10 00 00	.`.....@..►...►..
00400120	04 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00	♦.......♦.......
00400130	00 70 06 00 00 10 00 00 00 00 00 00 02 00 00 00	.p♠..►.....☻...
00400140	00 00 10 00 00 10 00 00 00 00 10 00 00 10 00 00	..►..►....►..►..
00400150	00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00	....►...........
00400160	3C 6F 00 00 78 00 00 00 00 90 00 00 40 D9 05 00	<o..x....É..@┘♣.
00400170	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00400180	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....

我们可以看到内容是一致的。我们按 F9 键继续运行。

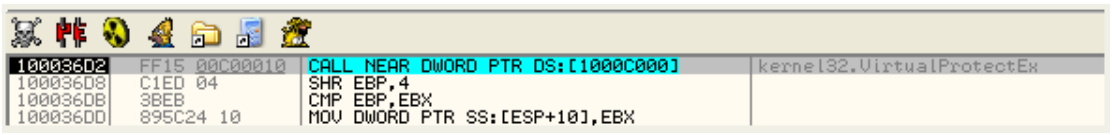


```
10003EF8   FF15 94C00010   CALL NEAR DWORD PTR DS:[1000C094]   kernel32.GetFileSize
10003EFE   8946 0C         MOV DWORD PTR DS:[ESI+C],EAX
10003F01   8B06            MOV EAX,DWORD PTR DS:[ESI]
10003F03   68 98C30010     PUSH 1000C398                       ASCII ".exe"
10003F08   50              PUSH EAX
10003F09   897E 04         MOV DWORD PTR DS:[ESI+4],EDI
10003F0C   E8 BF230000     CALL 100062D0                       AntiDebu.100062D0
10003F11   83C4 08         ADD ESP,8
10003F14   85C0            TEST EAX,EAX
10003F16   74 07           JE SHORT 10003F1F                   AntiDebu.10003F1F
10003F18   C746 04 010000  MOV DWORD PTR DS:[ESI+4],1
10003F1F   8B06            MOV EAX,DWORD PTR DS:[ESI]
10003F21   68 90C30010     PUSH 1000C390                       ASCII ".EXE"
10003F26   50              PUSH EAX
10003F27   E8 A4230000     CALL 100062D0                       AntiDebu.100062D0
10003F2C   83C4 08         ADD ESP,8
10003F2F   85C0            TEST EAX,EAX
10003F31   74 07           JE SHORT 10003F3A                   AntiDebu.10003F3A
10003F33   C746 04 010000  MOV DWORD PTR DS:[ESI+4],1
10003F3A   8B06            MOV EAX,DWORD PTR DS:[ESI]
10003F3C   68 88C30010     PUSH 1000C388                       ASCII ".dll"
10003F41   50              PUSH EAX
10003F42   E8 89230000     CALL 100062D0                       AntiDebu.100062D0
10003F47   83C4 08         ADD ESP,8
10003F4A   85C0            TEST EAX,EAX
10003F4C   74 07           JE SHORT 10003F55                   AntiDebu.10003F55
10003F4E   C746 04 020000  MOV DWORD PTR DS:[ESI+4],2
10003F55   8B06            MOV EAX,DWORD PTR DS:[ESI]
10003F57   68 80C30010     PUSH 1000C380                       ASCII ".DLL"
10003F5C   50              PUSH EAX
10003F5D   E8 6E230000     CALL 100062D0                       AntiDebu.100062D0
10003F62   83C4 08         ADD ESP,8
10003F65   85C0            TEST EAX,EAX
```

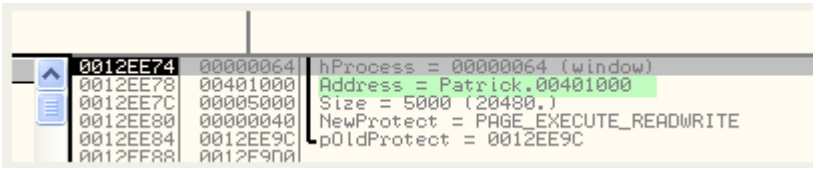
判断文件扩展名。



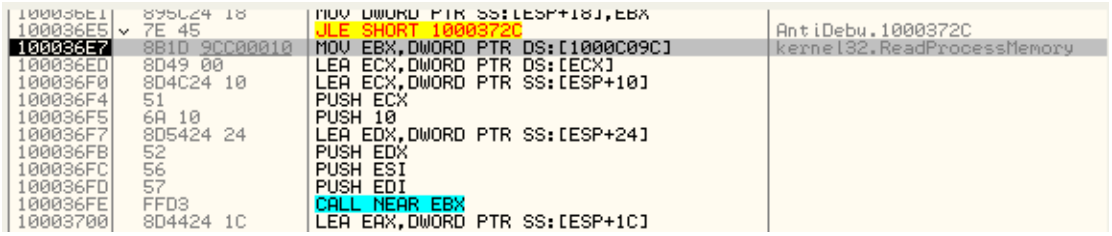
删除内存映射,貌似没有做文件内容的检测呀。



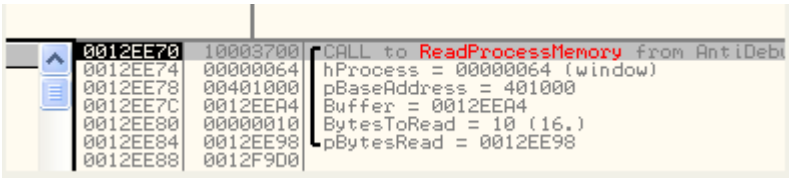
这里经过了几个不重要的 API 函数以后到了这里,这里要修改起始地址 401000 内存单元的访问权限。



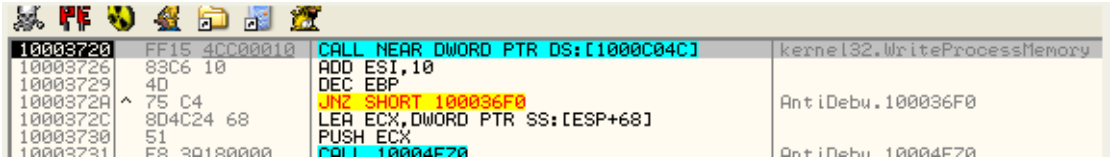
这里我们可以看到是要修改子进程的 401000 地址处的访问权限。

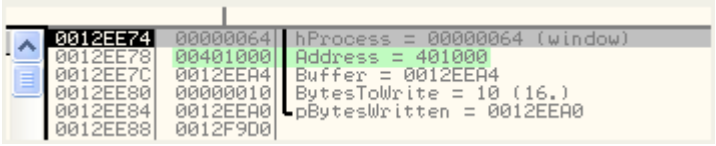


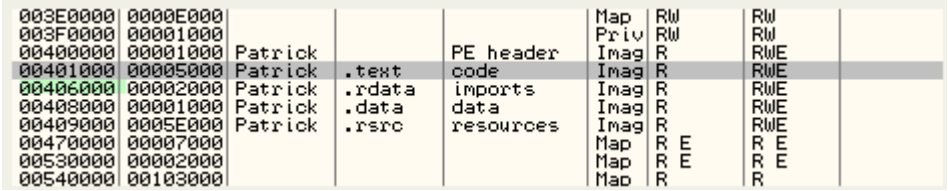
这里调用 ReadProcessMemory 读取进程内容。



这里我们可以看到是读取起始地址为 401000,长度为 16 个字节的内容。







起始地址为 401000 的这个区段是主程序的代码段。

我们先在来看看 401000 地址处的代码。

```
00401011   6F              OUTS DX,DWORD PTR ES:[EDI]               I/O command
00401012   DCAB 5C10A061    FSUBR QWORD PTR DS:[EBX+61A0105C]
00401018   24 85           AND AL,85
0040101A   FF29            JMP FAR FWORD PTR DS:[ECX]               Far jump
0040101C   56              PUSH ESI
0040101D   A6              CMPS BYTE PTR DS:[ESI],BYTE PTR ES:[EDI]
0040101E   2195 ED95659B   AND DWORD PTR SS:[EBP+9B6595ED],EDX
00401024   01B1 28FBA33A   ADD DWORD PTR DS:[ECX+3AA3FB28],ESI
0040102A   E1 76           LOOPDE SHORT 004010A2                    Patrick.004010A2
0040102C   E6 7B           OUT 7B,AL                                I/O command
0040102E   D300            ROL DWORD PTR DS:[EAX],CL
00401030   EC              IN AL,DX                                 I/O command
00401031   4F              DEC EDI
00401032   A7              CMPS DWORD PTR DS:[ESI],DWORD PTR ES:[EDI]
00401033   8BCD            MOV ECX,EBP
00401035   90              NOP
00401036   3C 8D           CMP AL,8D
00401038   90              NOP
00401039   A8 C7           TEST AL,0C7
0040103B   E5 52           IN EAX,52                                I/O command
0040103D   B2 38           MOV DL,38
0040103F   43              INC EBX
00401040   EE              OUT DX,AL                                I/O command
00401041   A3 B6FF3233     MOV DWORD PTR DS:[3332FFB6],EAX
00401046   3890 5AFDC4BF   CMP BYTE PTR DS:[EAX+BFC4FD5A],DL
```

明显经过加密处理了。

这里在循环调用 WriteProcessMemory,写入主程序的整个代码段,我们得一直按 F9 键运行,直到循环结束为止。

写入整个代码段,貌似像是在解密区段的样子。继续。

```
0012EE74   00000064   hProcess = 00000064 (window)
0012EE78   00405E00   Address = 405E00
0012EE7C   0012EEA4   Buffer = 0012EEA4
0012EE80   00000010   BytesToWrite = 10 (16.)
0012EE84   0012EEA0   pBytesWritten = 0012EEA0
0012EE88   0012F9D0
0012EE8C   0012F9B0
0012EE90   00000001
0012EE94   10000000   AntiDebu.10000000
```

这里马上循环写入快结束了。

```
10002B7C   FF15 80C00010   CALL NEAR DWORD PTR DS:[1000C080]   kernel32.ReleaseMutex
10002B82   B8 816E2D41     MOV EAX,412D6E81
10002B87   3D C8E57EC0     CMP EAX,C07EE5C8
10002B8C   74 07           JE SHORT 10002B95                   AntiDebu.10002B95
```

这里是释放 WAIT 互斥体信号量,也就是子进程调用 WaitForSingleObject 正在等待的信号量。

```
0012F104   00000050   hMutex = 00000050 (window)
0012F108   0012FD30
0012F10C   00150640
```

```
0000002C  Key              2.  000F003F  \REGISTRY\MACHINE
00000040  Key              2.  00020019  \REGISTRY\MACHINE\SOFTWARE\Microsoft\Winc
00000044  KeyedEvent      37.  000F0003  \KernelObjects\CritSecOutOfMemoryEvent
00000048  Mutant           4.  001F0001  \BaseNamedObjects\MYFIRSTINSTANCE
00000050  Mutant           5.  001F0001  \BaseNamedObjects\WAIT
0000005C  Mutant          16.  00120001  \BaseNamedObjects\ShimCacheMutex
00000018  Port             3.  001F0001
00000064  Process          9.  001F0FFF
```

现在我们按 F8 键,子进程就会运行起来。我们直接按 F9 键运行。

```
10002BB4   52              PUSH EDX
10002BB5   FF15 70C10010   CALL NEAR DWORD PTR DS:[1000C170]   USER32.WaitForInputIdle
10002BBB   B8 3611C531     MOV EAX,31C51136
10002BC0   3D 3008F6CF     CMP EAX,CFF60830
10002BC5   74 07           JE SHORT 10002BCE                   AntiDebu.10002BCE
```

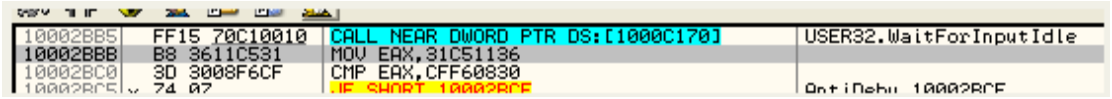
这里调用的是 WaitForInputIdle 这个函数,我们来看看 MSDN 中的说明。

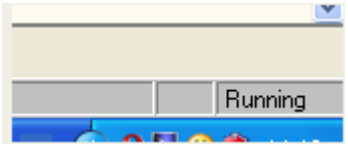
Remarks

The **WaitForInputIdle** function enables a thread to suspend its execution until a specified process has finished its initialization and is waiting for user input with no input pending. This can be useful for synchronizing a parent process and a newly created child process. When a parent process creates a child process, the **CreateProcess** function returns without waiting for the child process to finish its initialization. Before trying to communicate with the child process, the parent process can use **WaitForInputIdle** to determine when the child's initialization has been completed. For example, the parent process should use **WaitForInputIdle** before trying to find a window associated with the child process.

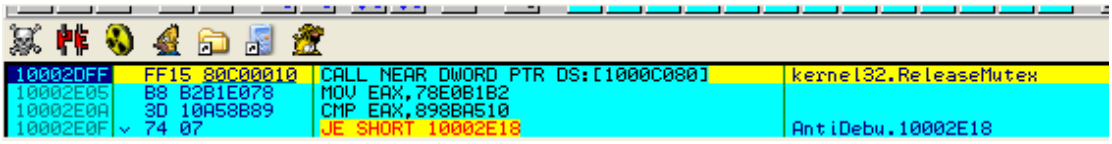
我们可以看到该函数可以使一个线程挂起,直到规定线程初始化完成为止。对于父子进程之间的同步极为有用。

我们直接对下一条指令处设置一个断点,然后运行起来。

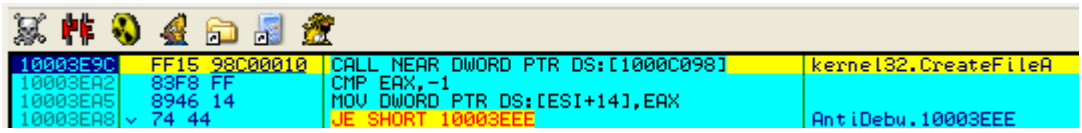




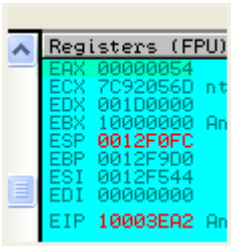
我们可以看到父进程现在已经处于运行状态了,我们现在接着来看子进程。

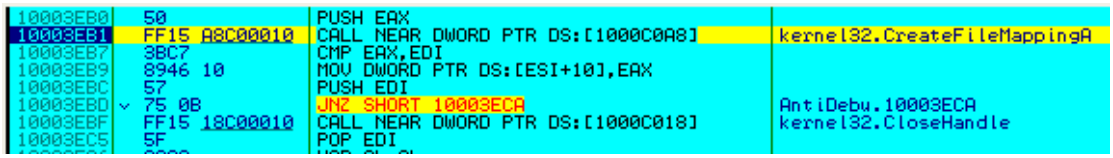


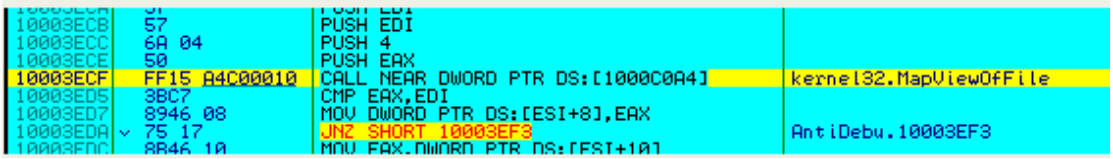
现在子进程调用 ReleaseMutex 释放 WAIT 互斥体的信号量,但是父进程并没有调用 WaitForSingleObject 进行 WAIT 信号量的等待,所以我们继续跟踪子进程,继续按 F9 键运行。

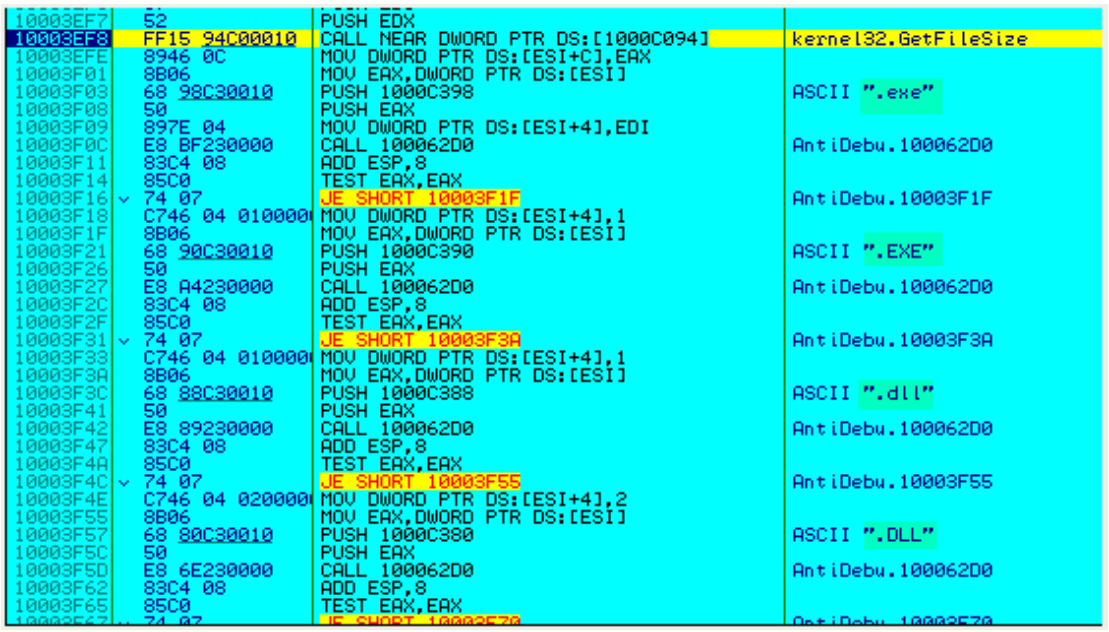


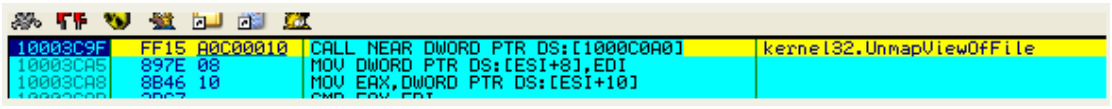
这里又是跟父进程一样检查 AntiDebugDll.dll 文件。不再赘述。



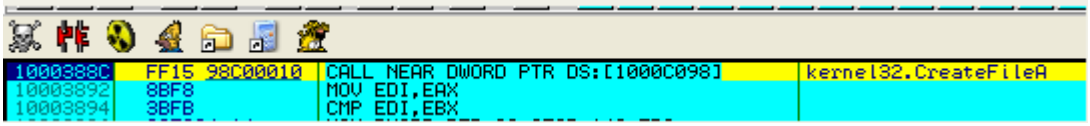


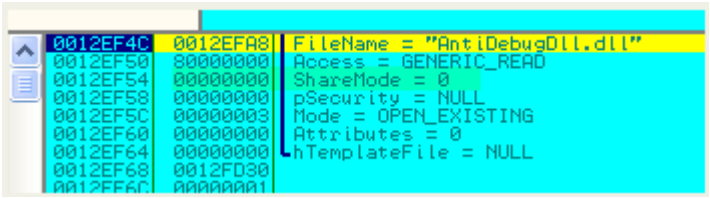




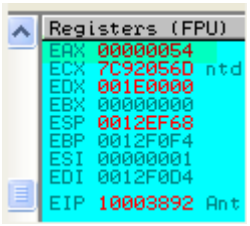


现在到了关键部位了。

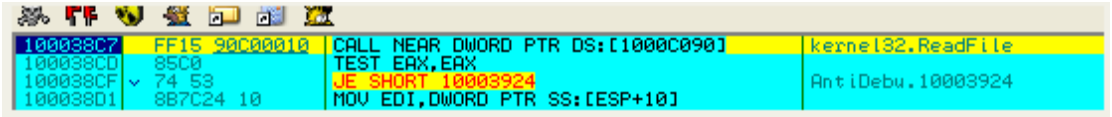




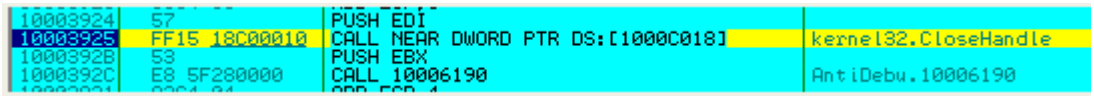
跟之前一样我们将 dwShareMode 修改为 1。



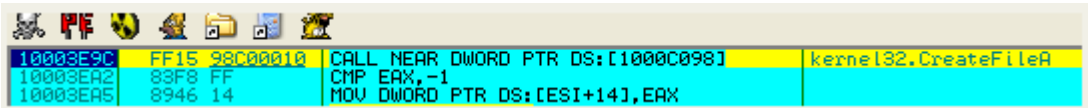
成功返回句柄。



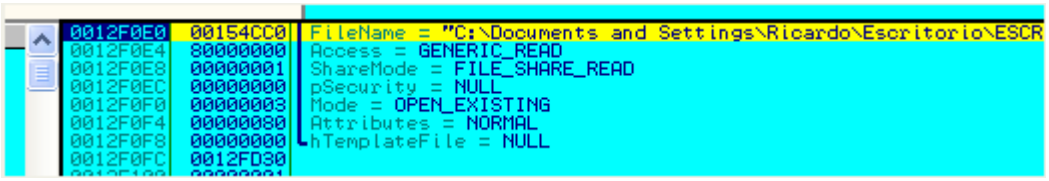
期间又调用了几个不重要的 API,然后就到了 ReadFile 这里,读取起始的几个字节判断有没有被修改。



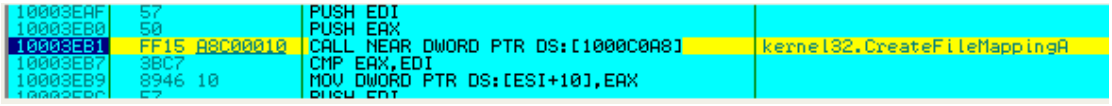
接着这里关闭文件句柄。



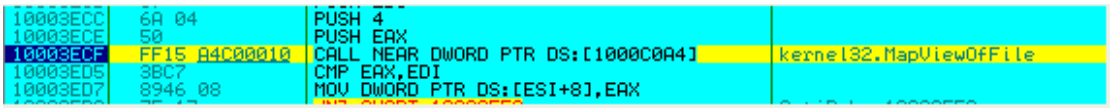
这里再次调用 CreateFileA 打开 Patrick.exe。

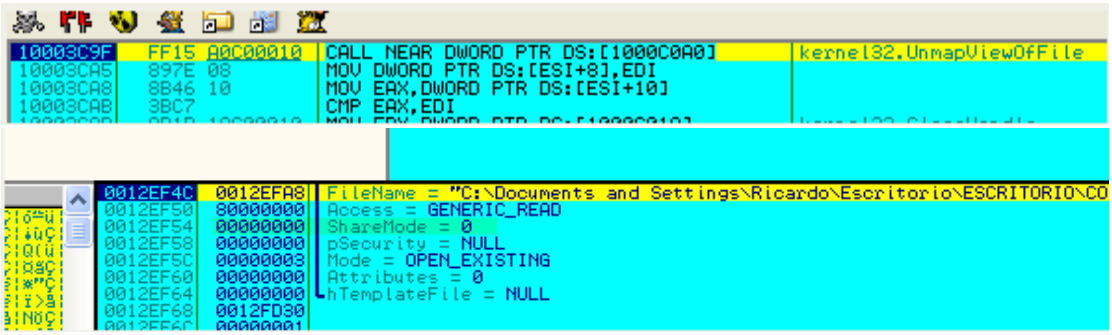


成功返回句柄。

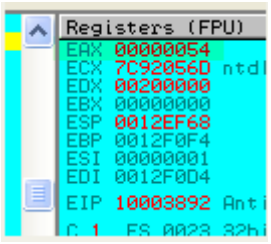


这部分与父进程是一样的,我们不再赘述。

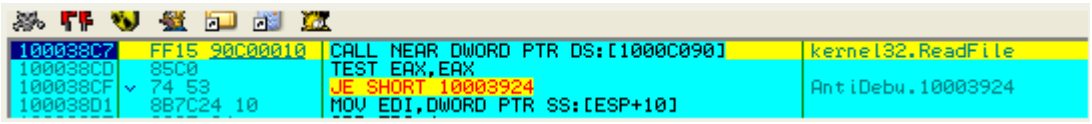


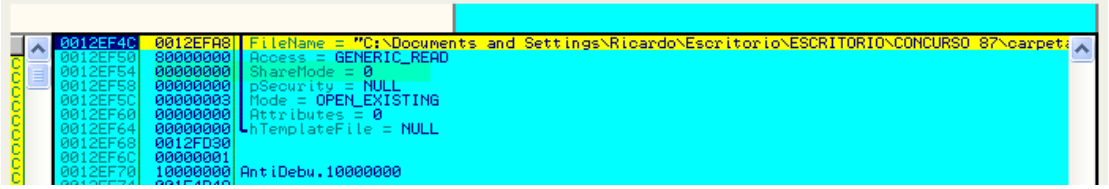


这里我们依然将 dwShareMode 修改为 1。

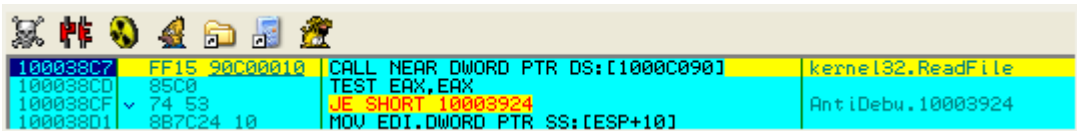


我们继续按 F9 键运行,经过几个 API 函数以后会再次调用 ReadFile,读取并检测起始的几个字节内容。

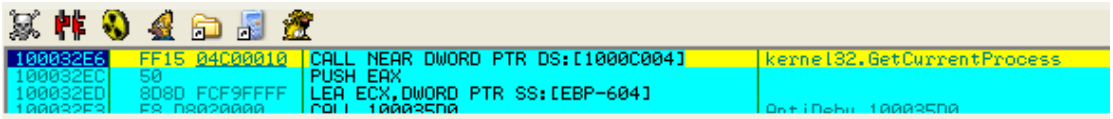




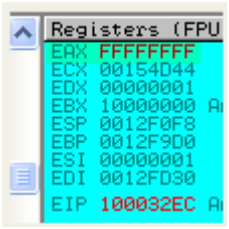
这里这些步骤都是相似的,我们只要遇到 CreateFileA,将 dwShareMode 修改为 1 即可。

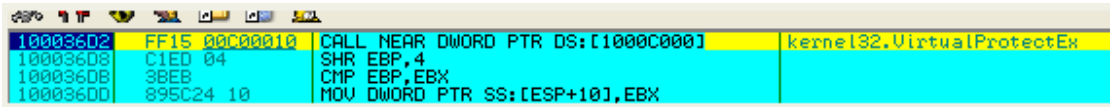


又是调用 ReadFile 进行检查。

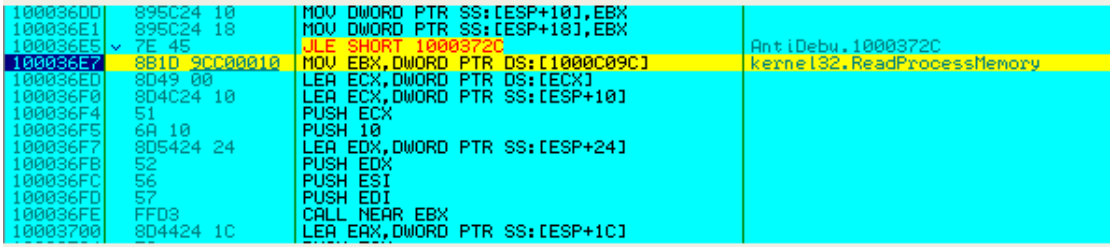


继续往下跟,这里调用 GetCurrentProcess,获取当前进程的句柄,貌似要进行一些新的处理了,我们一起来看看。

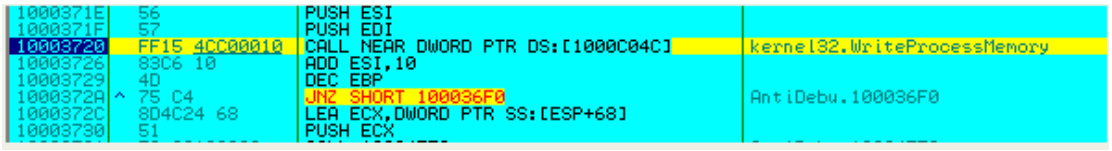




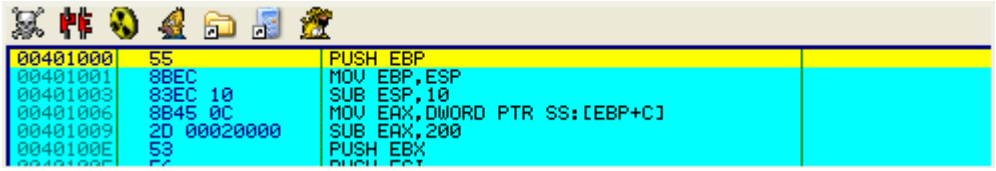
这里又是跟父进程一样调用 VirtualProtectEx 修改内存访问权限,为下一步调用 ReadProcessMemory,WriteProcesMemory 读写内存做铺垫。



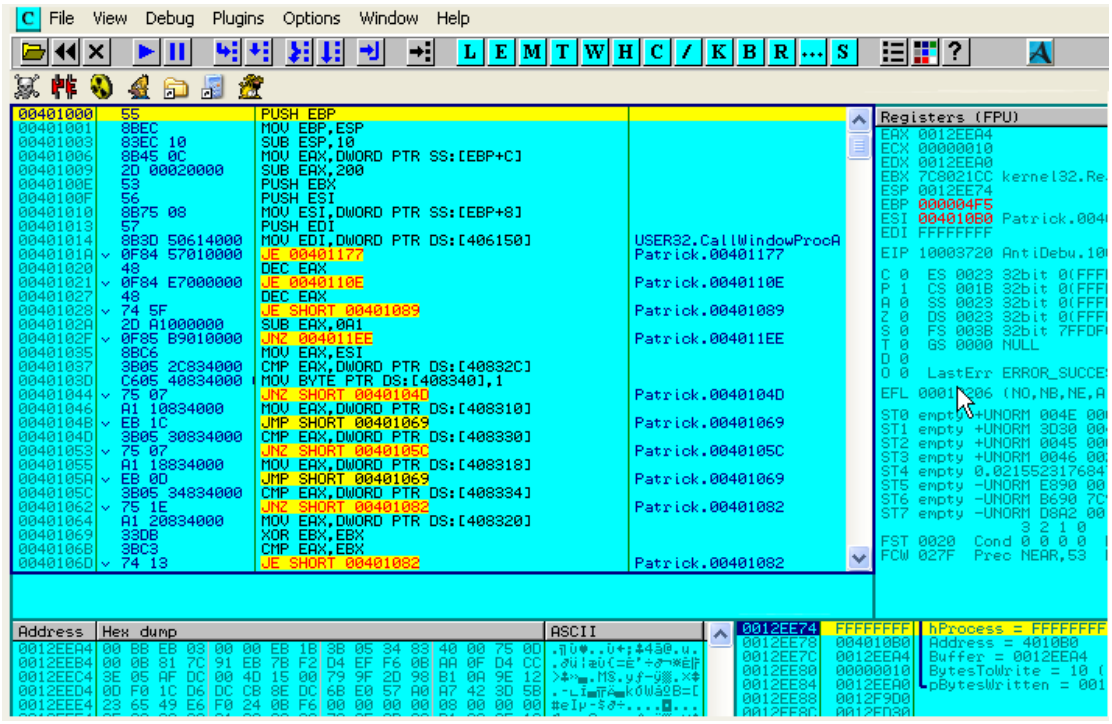
接着又是 WriteProcessMemory。

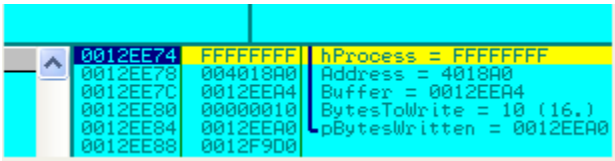


这里又是写入起始地址为 401000,长度为 16 的内容。



我们定位到 401000 看看。

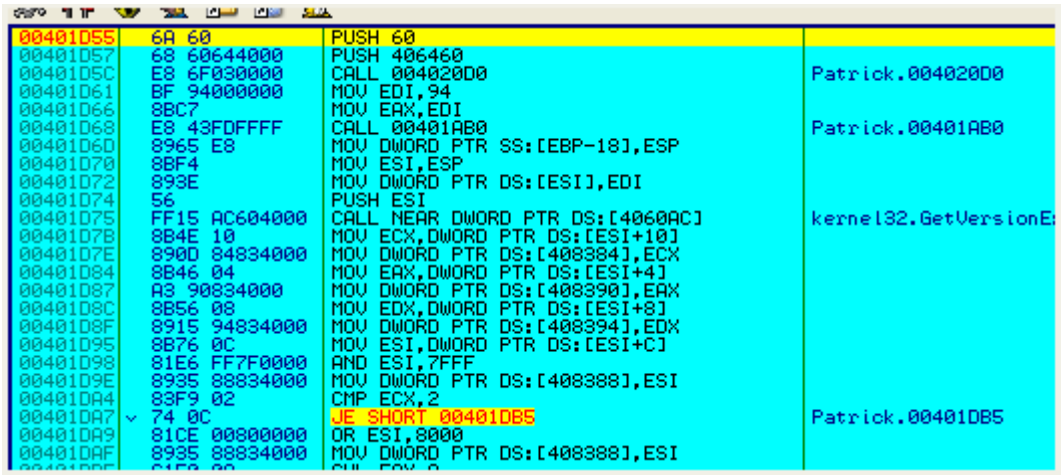


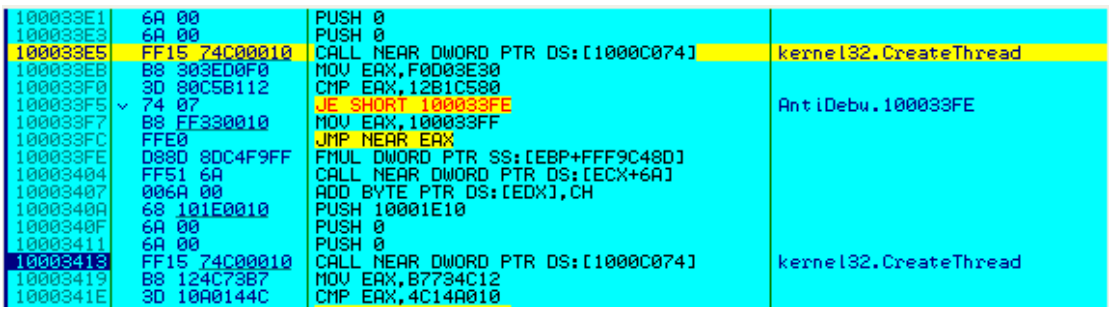


这里又是循环写入,我们一直按 F9 键,直到整个主程序代码段都被写完毕为止。

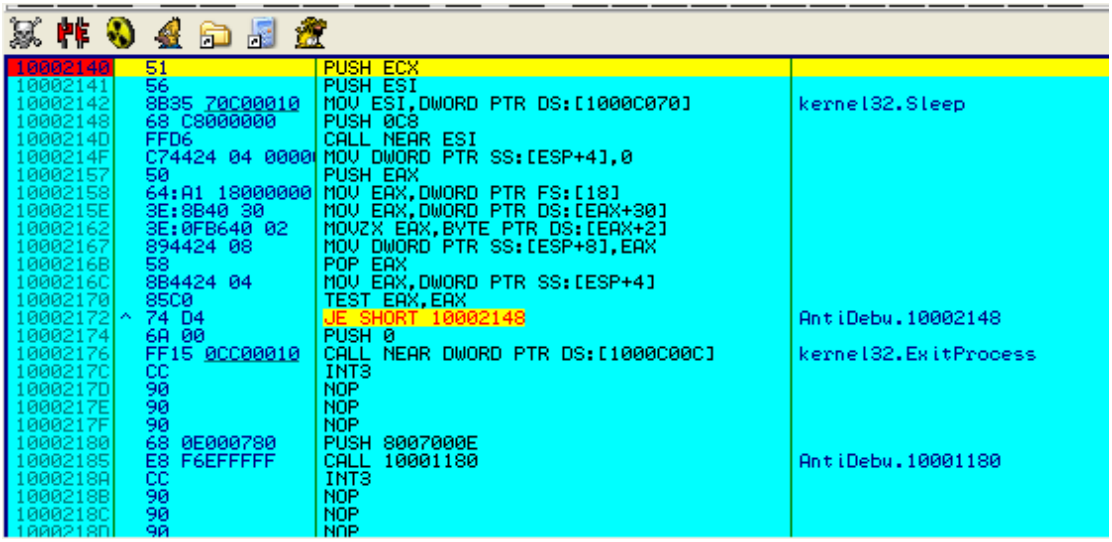
好,现在已经循环写入完毕了,通过 PEEditor 可以得知主程序的 OEP 为 1D55,我们现在来看看 401D55 处的内容是什么。

可能看到的确像是入口点的样子,说明之前循环写入的确是在解密主程序代码段。

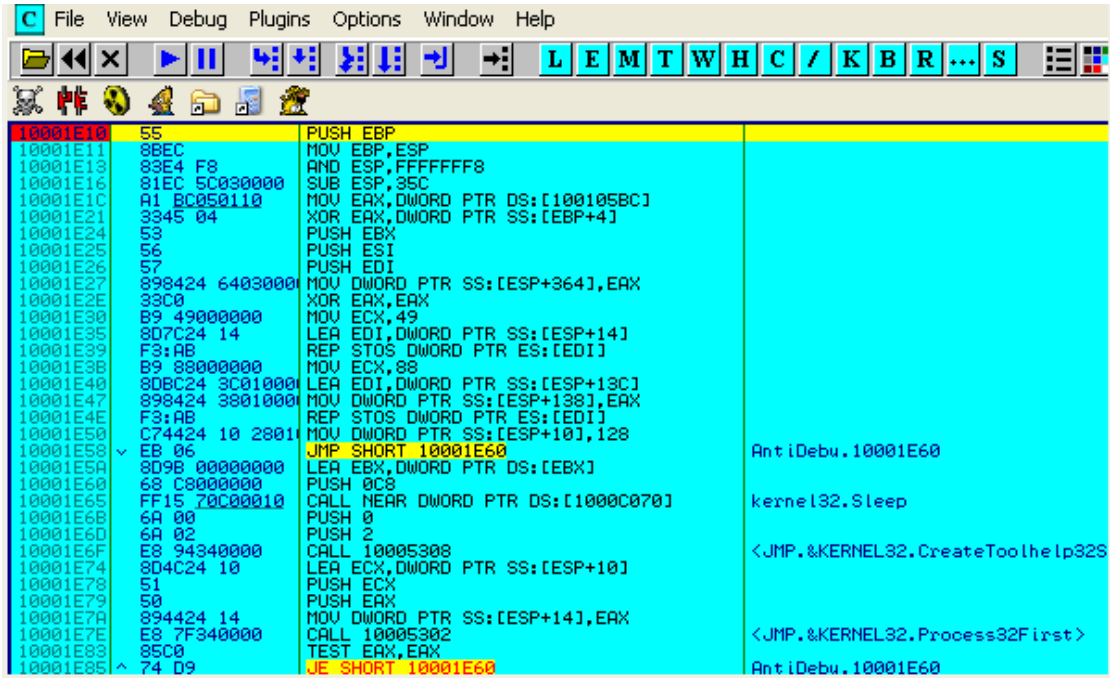




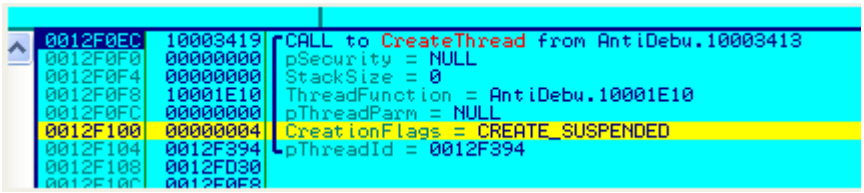
现在区段已经解密完毕了,接着这里要调用 CreateThread 创建线程,我们将 CreationFlags 修改为 4,即挂起状态。



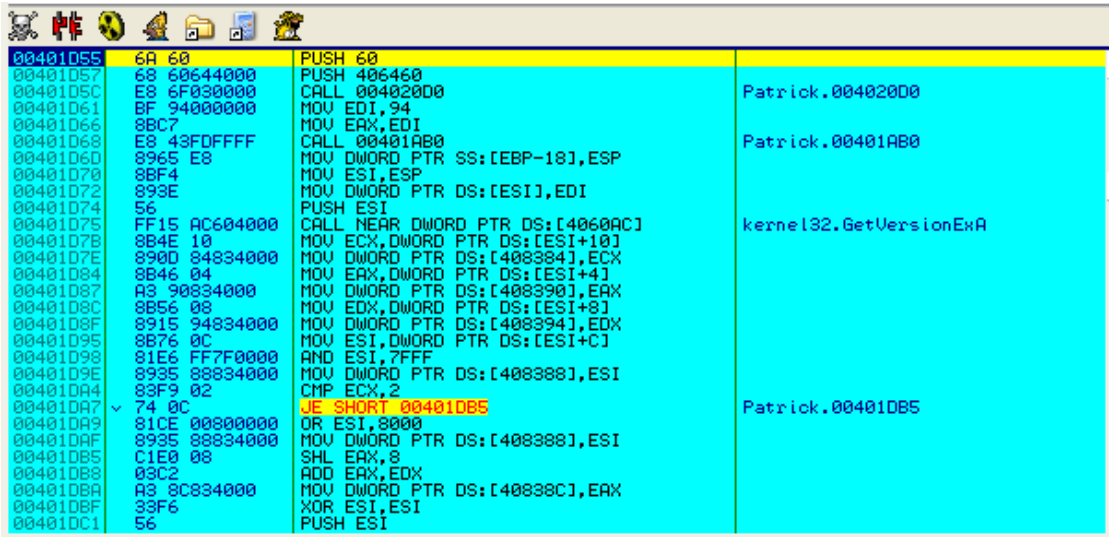
我们定位该线程的起始地址看看,这是是创建的第一个线程,我们继续往下看,看看会不会再创建其他进程。



这里是创建第二个线程,我们依然将 dwCreationFlags 修改为 0x4。

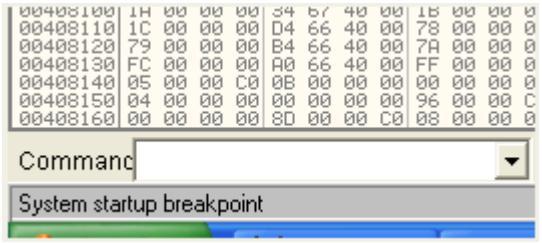


好了,现在区段已经解密完毕了,我们接下来需要顺利断在 OEP 处,我们直接删除之前设置的内存访问断点,然后对主程序代码段设置内存访问断点。接着运行起来。

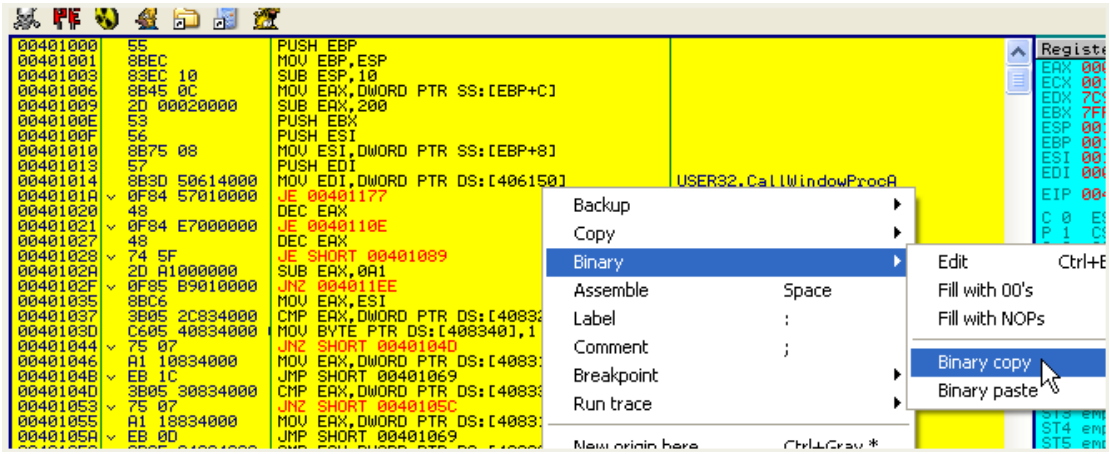


现在我们将成功断在了 OEP 处,下面我们来进行 dump。

现在我们再打开一个 OD,加载 Patrick.exe,断在了系统断点处。



现在我们将子进程代码段的所有字节都拷贝到这个新开的 OD 中。



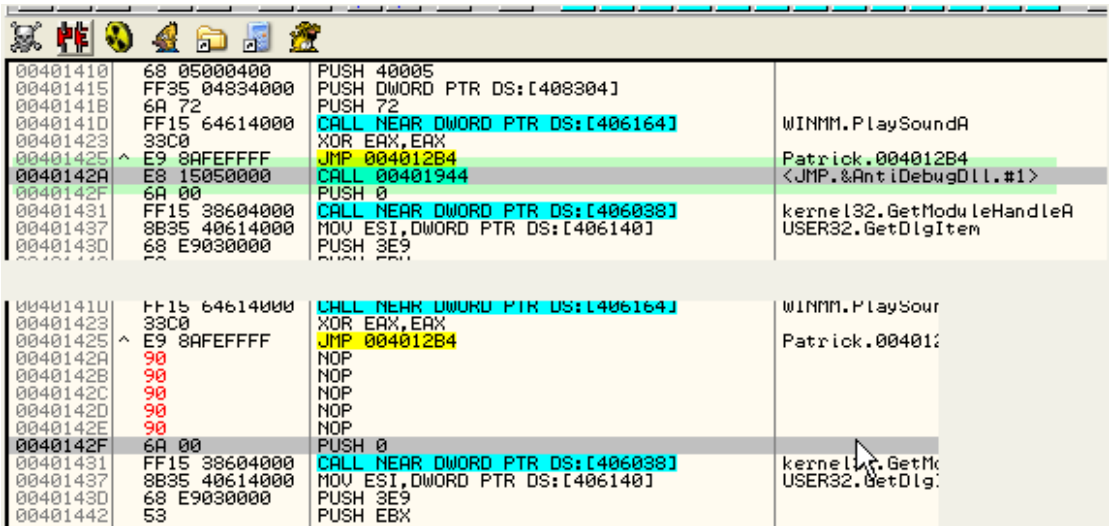
接着将修改保存到文件,我们来看下 IAT,IAT 项都是正常的。

现在我们就不需要执行 AntiDebugDll.dll 这个反调试模块中的内容了,所以我们可以直接将 AntiDebugDll.dll 的入口点处的指令修改为 RET。

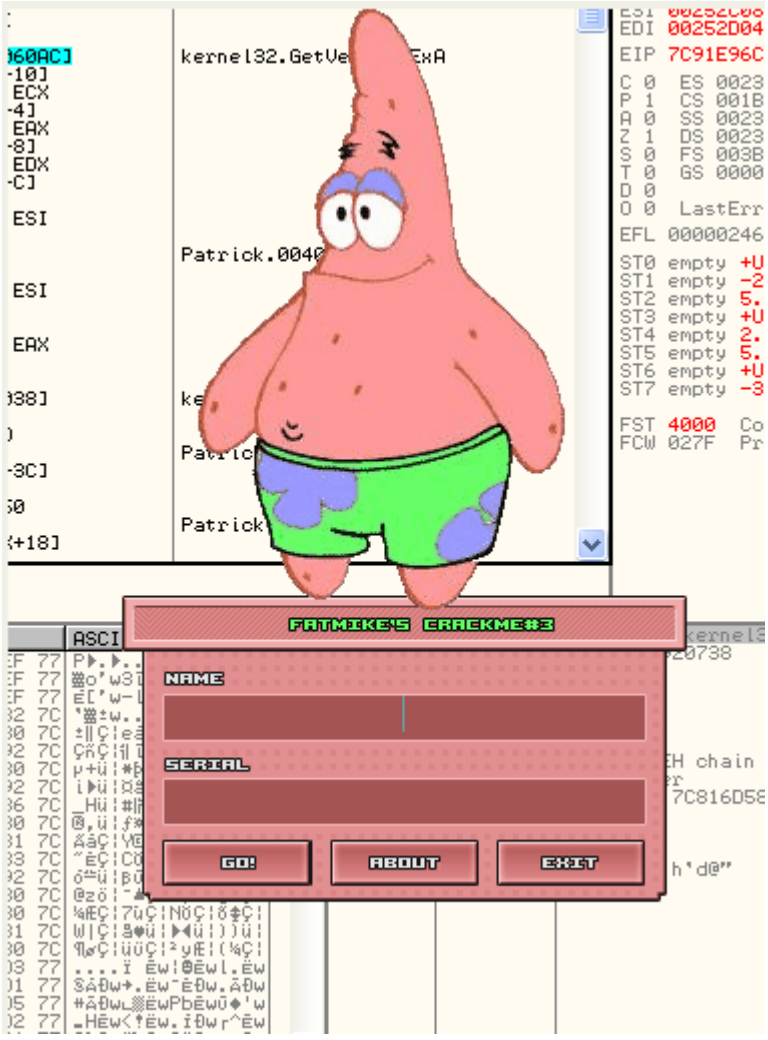
我们再次用 OD 加载刚刚保存的文件,断在了系统断点处,这里我们定位到 AntiDebugDll.dll 的入口地址:100064D4,直接将第一条指令修改为 RET,让其返回。接着保存修改到文件。

接着我们打开 PEEditor 程序的所有区段访问权限都修改为 E0000020(可读可写可执行)。

这里我们还可以看到一个主程序代码段中还有一处 CALL 是指向的 AntiDebugDll.dll 的,我们直接把它 NOP 掉。



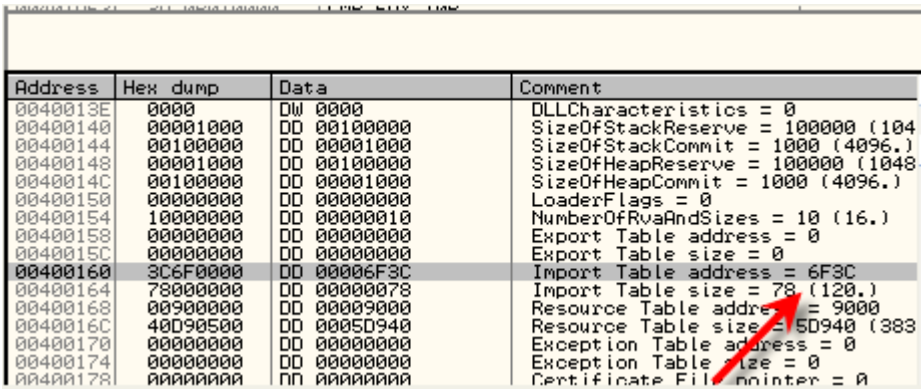
我们保存修改到文件,直接运行修改好的 CrackMe。



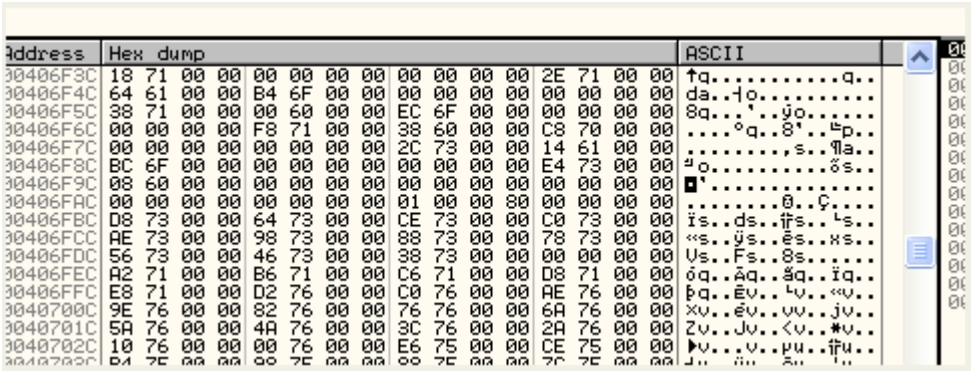
我们现在运行起来了,我们可以看到程序正常运行。

下面我们需要进一步干掉这个 AntiDebugDll.dll 这个模块,让主程序无需加载这个 DLL。

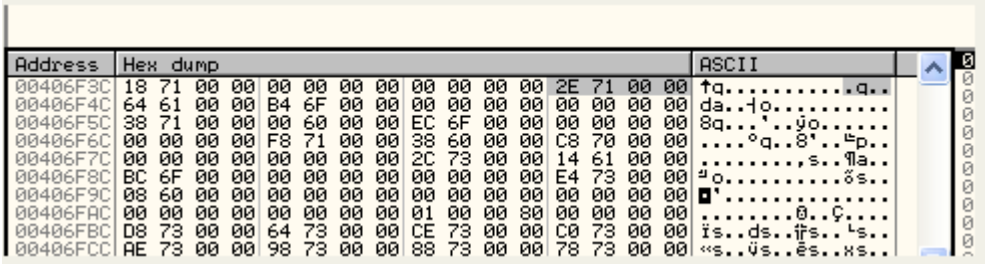
为了 DLL 不加载,我们应该进行如下处理:



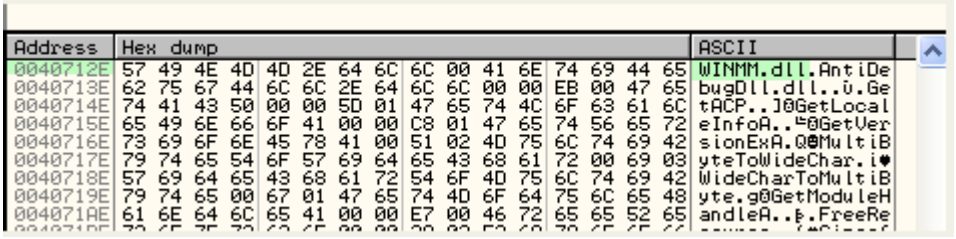
我们定位到导入表(IT)的起始地址为 406F3C。



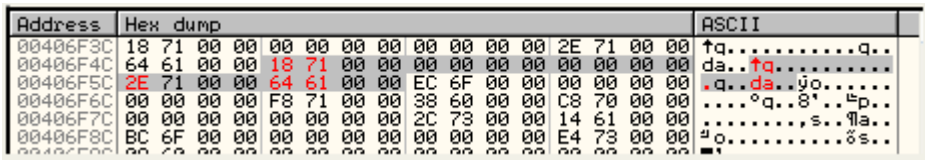
大家应该可以记得吧,每个 DLL 项占 5 个 DWORD,第 4 个 DWORD 为 DLL 名称字符串指针。



这里第一个 DLL 名称字符串指针为 40712E,我们一起来看一看。



是 WINMM.DLL,第二个 DLL 为 AntiDebugDll.dll,这里我们为了剔除掉 AntiDebugDll.dll 这个模块,我们可以将 AntiDebugDll.dll 对应的 DLL 项修改得与 WINMM.DLL 这一个 DLL 项一致。



保存修改到文件。

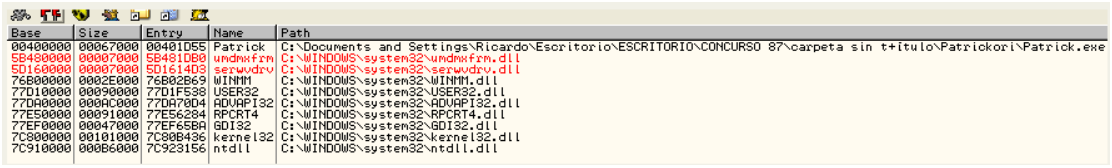
我们重启 OD,再来看看导入表。

```
00400154   10000000   DD 00000010   NumberOfRvaAndSizes = 10 (16.
00400158   00000000   DD 00000000   Export Table address = 0
0040015C   00000000   DD 00000000   Export Table size = 0
00400160   3C6F0000   DD 00006F3C   Import Table address = 6F3C
00400164   78000000   DD 00000078   Import Table size = 78 (120.)
00400168   00900000   DD 00009000   Resource Table address = 9000
```

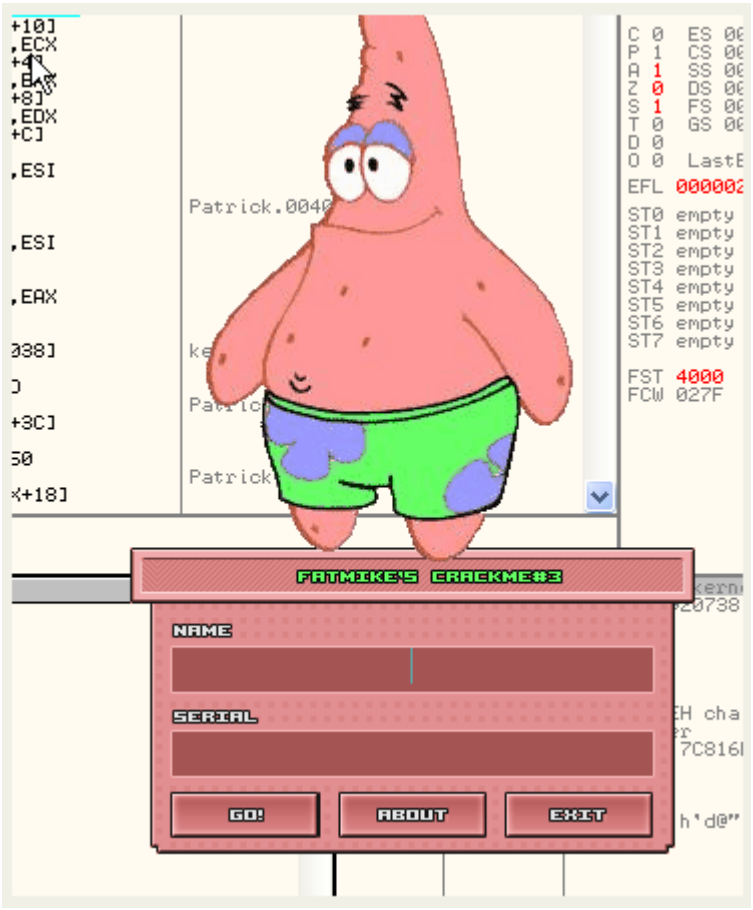
如下:

```
00400148   00001000   DD 00100000   SizeOfHeapReserve = 100000 (1048
0040014C   00100000   DD 00001000   SizeOfHeapCommit = 1000 (4096.)
00400150   00000000   DD 00000000   LoaderFlags = 0
00400154   10000000   DD 00000010   NumberOfRvaAndSizes = 10 (16.)
00400158   00000000   DD 00000000   Export Table address = 0
0040015C   00000000   DD 00000000   Export Table size = 0
00400160   506F0000   DD 00006F50   Import Table address = 6F50
00400164   78000000   DD 00000078   Import Table size = 78 (120.)
00400168   00900000   DD 00009000   Resource Table address = 9000
0040016C   40D90500   DD 0005D940   Resource Table size = 5D940 (383
00400170   00000000   DD 00000000   Exception Table address = 0
```

现在我们会发现并没有加载 AntiDebugDll.dll 了。



Base	Size	Entry	Name	Path
00400000	00067000	00401D55	Patrick	C:\Documents and Settings\Ricardo\Escritorio\ESCRITORIO\CONCURSO 87\carpeta sin t+itulo\Patrickori\Patrick.exe
5B480000	00007000	5B481DB0	wmdmxfrm	C:\WINDOWS\system32\wmdmxfrm.dll
5D160000	00007000	5D1614D3	serwvdrv	C:\WINDOWS\system32\serwvdrv.dll
76B00000	0002E000	76B02B69	WINMM	C:\WINDOWS\system32\WINMM.dll
77D10000	00090000	77D1F538	USER32	C:\WINDOWS\system32\USER32.dll
77DA0000	000AC000	77DA7D04	ADVAPI32	C:\WINDOWS\system32\ADVAPI32.dll
77E50000	00091000	77E56284	RPCRT4	C:\WINDOWS\system32\RPCRT4.dll
77EF0000	00047000	77EF65BA	GDI32	C:\WINDOWS\system32\GDI32.dll
7C800000	00101000	7C80B436	kernel32	C:\WINDOWS\system32\kernel32.dll
7C910000	000B6000	7C923156	ntdll	C:\WINDOWS\system32\ntdll.dll



好了,程序完美运行。

嘿嘿,这个 CrackMe 我们就搞定了。