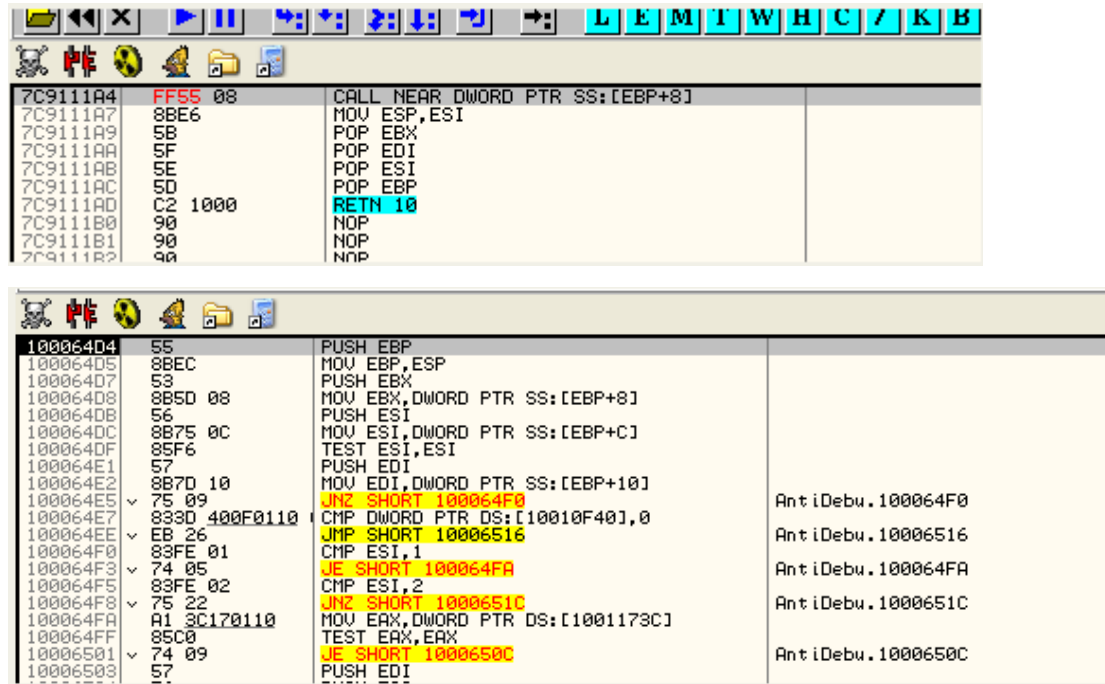


第四十七章-Patrick 的 CrackMe-Part2

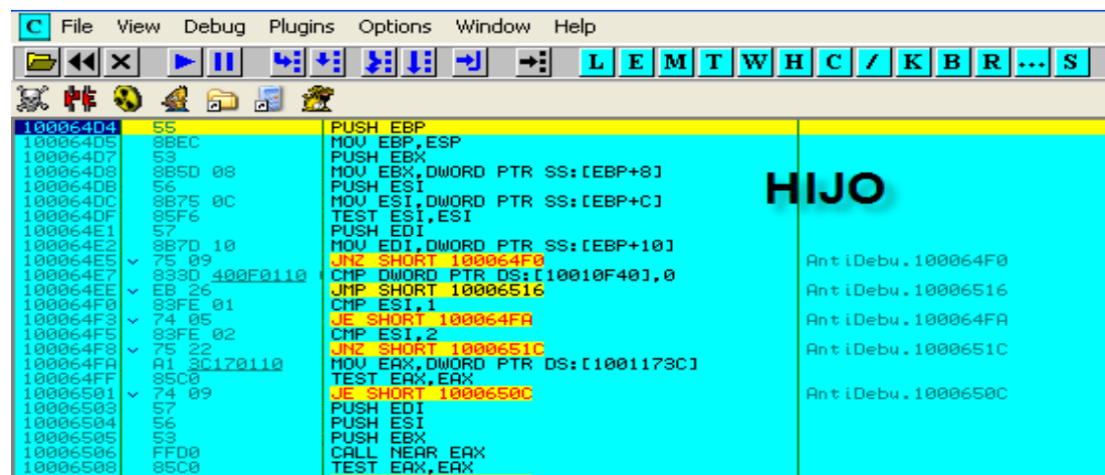
我们接着上一章的继续讲,上一章的结尾处我们是通过如下方式来附加新创建的进程的,首先将 NTDLL.DLL 中调用其他模块的入口点那一条指令设置为一个死循环,然后将 OD 设置为即时调试器(JIT),接着打开任务管理器,选中新创建的进程单击鼠标右键选择调试,这样 OD 就附加了新创建的进程。接着在死循环的指令处设置一个断点,然后将 Patch 过的字节码恢复为原始字节,然后直接按 F9 键运行两次后,模块列表窗口中就会出现 AntiDebugDll.dll 了,接着我们给 AntiDebugDll.dll 的代码段设置内存访问断点,然后删除掉之前设置的 INT 3 断点,运行起来,这样就可以断在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



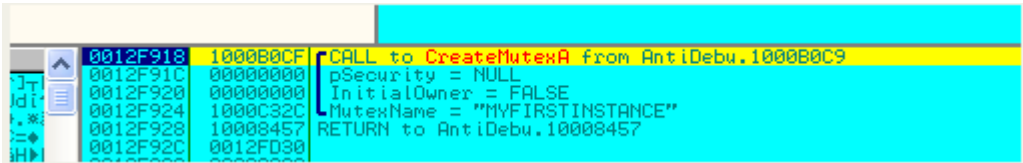
现在父进程处于 CreateProcessA 调用语句的返回地址处。



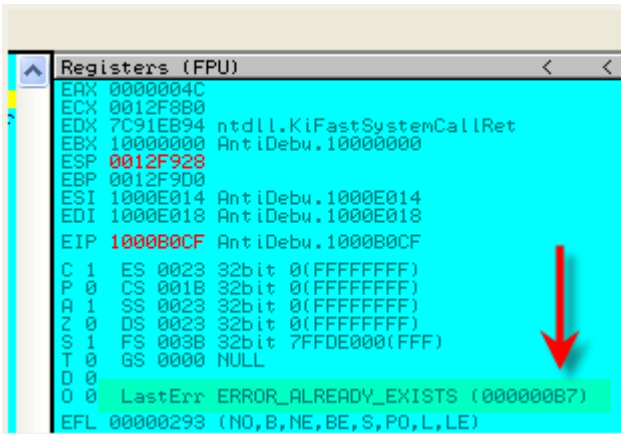
子进程处于 AntiDebugDll.dll 的入口点处。这里我将调试子进程的 OD 换了一种配色方案,这样可以防止大家在阅读的时候将两个 OD 搞混淆了。

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。

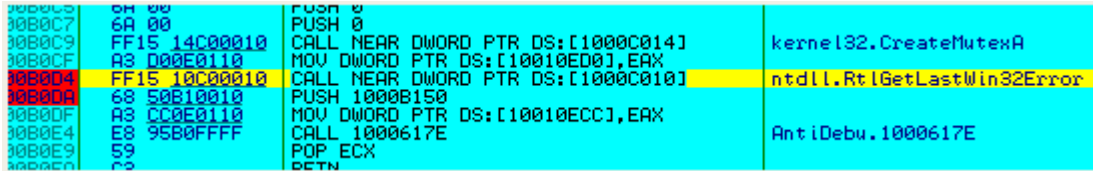
我们知道父进程中调用 CreateMutexA 这个函数,这里我们给子进程的中 CreateMutexA 也设置一个断点。



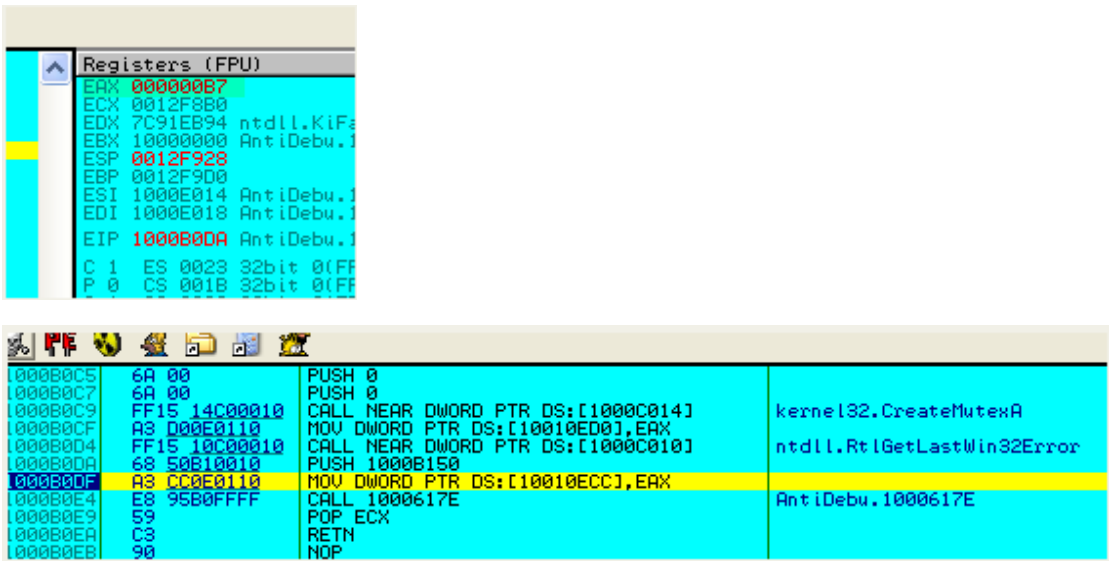
我们运行起来,断在了这里,这里由于父进程已经创建了 MYFIRSTINSTANCE 这个互斥体,所以我们如果执行了该函数,GetLastError 的错误码将返回 0xB7,即 ERROR_ALREADY_EXISTS。这里我们执行到返回验证一下。



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

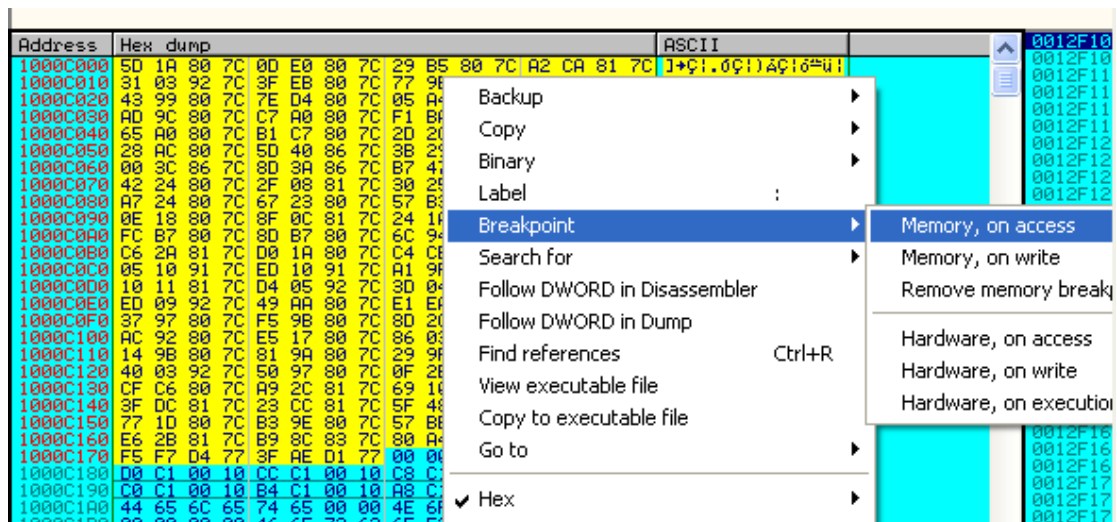


接着这里调用 RtlGetLastWin32Error 获取错误码,这里错误码我们已经知道了是 0xB7 了,即 ERROR_ALREADY_EXISTS。

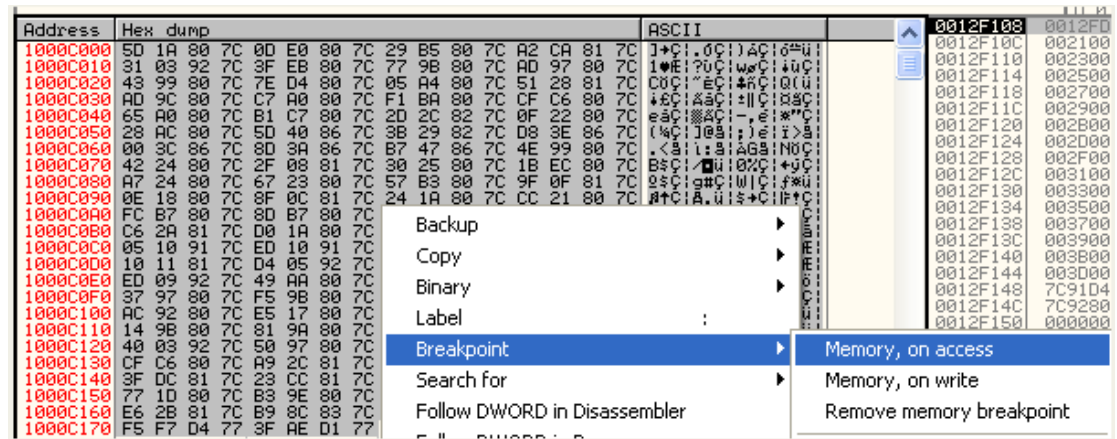


这里将错误码保存到 10010ECC 指向的内存单元中,我们给该内存单元设置一个内存访问断点,运行起来。

断在了这里,这里是读取该错误码进行比较。

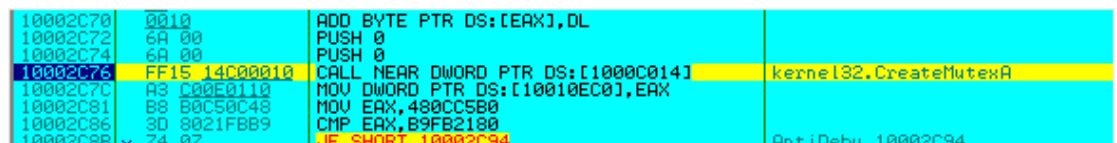


接下来我们同样给父进程的 AntiDebugDll.dll 的整个 IAT 表设置内存访问断点。

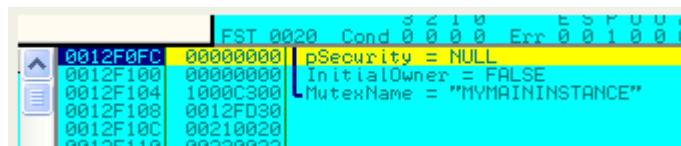


这里我们就给父进程的 AntiDebugDll.dll 的 IAT 都设置了内存访问断点,也就说当程序中调用 AntiDebugDll.dll 模块 IAT 中的 API 函数时就会断下来,这里还需要注意一点,有可能该 DLL 会通过调用 GetProcAddress 来获取其他的 API 函数指针,所以这里我们给 GetProcAddress 也设置一个断点,以防万一。

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。



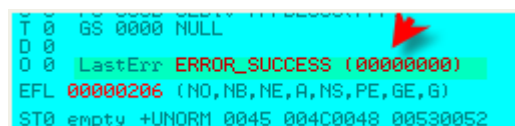
断在了这里,这里要创建 MYMAININSTANCE 这个互斥体,这个互斥体之前未被创建过。



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”。

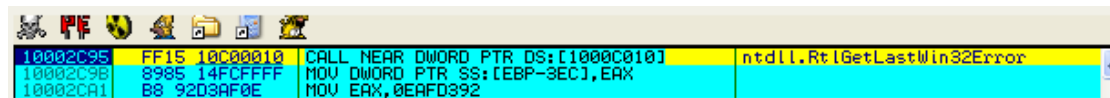
这里要创建的互斥体叫做 MYMAININSTANCE。顾名思义:“主体实例”。

我们按 F8 键执行该 API 函数。

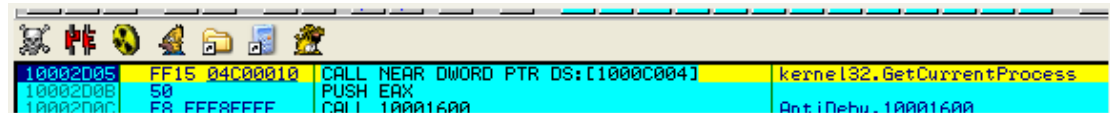


这里我们可以看到错误码为 ERROR_SUCCESS,说明互斥体创建成功。

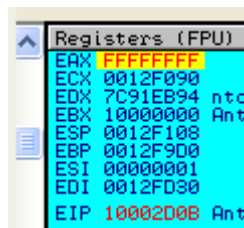
好,我们按 F9 键运行起来,看看下面会调用哪个 API 函数。



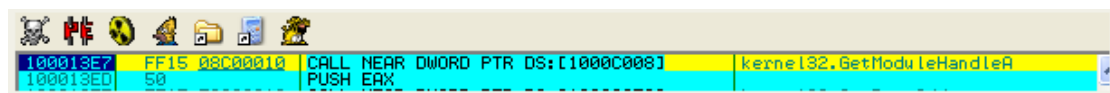
这里又是调用 RtlGetLastWin32Error 获取上次调用 CreateMutexA 这个 API 函数的错误码,我们继续按 F9 键运行。



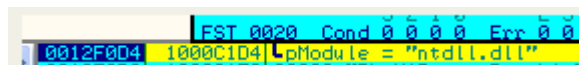
这里是获取进程句柄,我们按 F8 键执行调用 API 函数,会返回(-1)FFFFFFFF.代表当前进程。该句柄不在句柄表中,不是真正的句柄,我们叫它伪句柄。



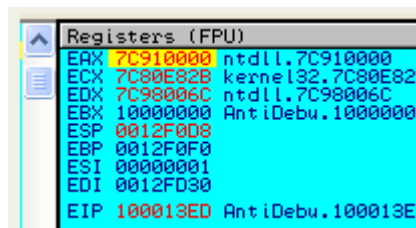
继续按 F9 键运行。



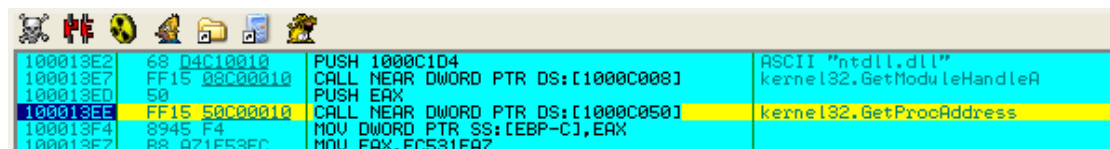
这里调用的是 GetModuleHandleA,这样的跟踪对于我们练习哪些不熟悉的 API 函数是一种极佳的锻炼。我们按 F8 键执行。



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 GetProcAddress 的参数来获取 NTDLL.DLL 中包含的 API 函数指针。

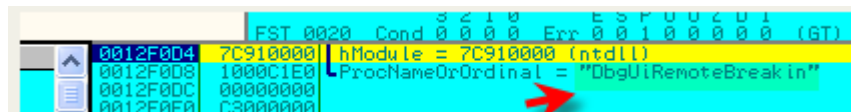


我希望我猜测是正确的,我们按 F9 键运行起来。



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。

10001453	FF15 00C00010	CALL NEAR DWORD PTR DS:[1000C000]	kernel32.VirtualProtectEx
10001459	B8 3ED00DA3	MOV EAX, A38D003E	
1000145E	3D 70CECA5C	CMP EAX, 5CCACE70	
10001463	74 07	JE SHORT 1000146C	AntiDebu.1000146C
10001465	B8 6D140010	MOV EAX, 1000146D	

FST 0020 Cond 0 0 0 0 Err 0 0 1 0 0 0 0 0 (GT)		
0012F0C8	FFFFFFFF	hProcess = FFFFFFFF
0012F0CC	7C96077B	Address = ntdll.DbgUiRemoteBreakin
0012F0D0	00000001	Size = 1
0012F0D4	00000040	NewProtect = PAGE_EXECUTE_READWRITE
0012F0D8	0012F0E8	pOldProtect = 0012F0E8
0012F0DC	00000040	
0012F0E0	C3000000	

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 DbgUiRemoteBreakin 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。

10001478	52	PUSH EAX	
10001479	8B45 08	MOV EAX, DWORD PTR SS:[EBP+8]	
1000147C	50	PUSH EAX	
1000147D	FF15 4CC00010	CALL NEAR DWORD PTR DS:[1000C04C]	kernel32.WriteProcessMemory
10001483	B8 20E306A	MOV EAX, 6A30DE20	
10001488	3D 00A92D96	CMP EAX, 962DA900	
1000148D	74 07	JE SHORT 10001496	AntiDebu.10001496
1000148F	B8 97140010	MOV EAX, 10001497	

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

ST7 empty 0.0		
FST 0020 Cond 0 0 0 0 Err 0 0 1		
0012F0C8	FFFFFFFF	hProcess = FFFFFFFF
0012F0CC	7C96077B	Address = 7C96077B
0012F0D0	0012F0E3	Buffer = 0012F0E3
0012F0D4	00000001	BytesToWrite = 1
0012F0D8	00000000	pBytesWritten = NULL
0012F0DC	00000040	

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。

7C96077B	6A 08	PUSH 8	
7C96077D	68 C807967C	PUSH 7C9607C8	
7C960782	E8 3BE6FBFF	CALL 7C91EDC2	ntdll.7C91EDC2
7C960787	64:A1 18000000	MOV EAX, DWORD PTR FS:[18]	
7C96078D	8B40 30	MOV EAX, DWORD PTR DS:[EAX+30]	
7C960790	8078 02 00	CMP BYTE PTR DS:[EAX+2], 0	
7C960794	75 09	JNZ SHORT 7C96079F	ntdll.7C96079F
7C960796	F605 D402FE7F	TEST BYTE PTR DS:[7FFE02D4], 2	
7C96079D	74 20	JE SHORT 7C9607BF	ntdll.7C9607BF
7C96079F	8365 FC 00	AND DWORD PTR SS:[EBP-4], 0	
7C9607A3	E8 880AFBFF	CALL 7C911230	ntdll.DbgBreakPoint
7C9607A8	EB 11	JMP SHORT 7C9607BB	ntdll.7C9607BB
7C9607AA	90	NOP	
7C9607AB	90	NOP	
7C9607AC	90	NOP	
7C9607AD	90	NOP	
7C9607AE	90	NOP	
7C9607AF	33C0	XOR EAX, EAX	
7C9607B1	40	INC EAX	
7C9607B2	C3	RET	
7C9607B3	90	NOP	

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。

7C96077B	C3	RET	
7C96077C	0868 C8	OR BYTE PTR DS:[EAX-38], CH	
7C96077F	07	POP ES	Modification of segment registers
7C960780	96	XCHG EAX, ESI	
7C960781	7C E8	JL SHORT 7C96076B	ntdll.7C96076B
7C960783	3BE6	CMP ESP, ESI	
7C960785	FB	STI	
7C960786	FF64A1 18	JMP NEAR DWORD PTR DS:[ECX+18]	
7C96078A	0000	ADD BYTE PTR DS:[EAX], AL	
7C96078C	008B 40308078	ADD BYTE PTR DS:[EBX+78803040], CL	
7C960792	0200	ADD AL, BYTE PTR DS:[EAX]	
7C960794	75 09	JNZ SHORT 7C96079F	ntdll.7C96079F
7C960796	F605 D402FE7F	TEST BYTE PTR DS:[7FFE02D4], 2	
7C96079D	74 20	JE SHORT 7C9607BF	ntdll.7C9607BF
7C96079F	8365 FC 00	AND DWORD PTR SS:[EBP-4], 0	
7C9607A3	E8 880AFBFF	CALL 7C911230	ntdll.DbgBreakPoint
7C9607A8	EB 11	JMP SHORT 7C9607BB	ntdll.7C9607BB
7C9607AA	90	NOP	
7C9607AB	90	NOP	
7C9607AC	90	NOP	
7C9607AD	90	NOP	
7C9607AE	90	NOP	
7C9607AF	33C0	XOR EAX, EAX	
7C9607B1	40	INC EAX	
7C9607B2	C3	RET	
7C9607B3	90	NOP	

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。

10001528	FF15 00C00010	CALL NEAR DWORD PTR DS:[1000C000]	kernel32.GetModuleHandleA
10001531	50	PUSH EAX	
10001532	FF15 50C00010	CALL NEAR DWORD PTR DS:[1000C050]	kernel32.GetProcAddress

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。

0012F004	7C910000	hModule = 7C910000 (ntdll)
0012F008	1000C1F4	ProcNameOrOrdinal = "DbgBreakPoint"
0012F0DC	00000040	

10001597	FF15 00C00010	CALL NEAR DWORD PTR DS:[1000C000]	kernel32.VirtualProtectEx
1000159D	B8 116582CD	MOV EAX,CD826511	
100015A2	3D 4816DF32	CMP EAX,32DF1648	

修改 DbgBreakPoint 首字节的访问权限。

100015B0	8B45 08	MOV EAX,DWORD PTR SS:[EBP+8]	
100015C8	50	PUSH EAX	
100015C1	FF15 4CC00010	CALL NEAR DWORD PTR DS:[1000C04C]	kernel32.WriteProcessMemory
100015C7	B8 F3722594	MOV EAX,942572F3	
100015CC	3D D8F0416C	CMP EAX,6C41F0D8	

这里要 Patch DbgBreakPoint 首字节了,我们来看看参数。

0012F0C8	FFFFFFFF	hProcess = FFFFFFFF
0012F0CC	7C911230	Address = 7C911230
0012F0D0	0012F0E3	Buffer = 0012F0E3
0012F0D4	00000001	BytesToWrite = 1
0012F0D8	00000000	pBytesWritten = NULL
0012F0DC	00000040	

这里我们来看看 DbgBreakPoint 的实现代码。

7C911230	CC	INT3
7C911231	C3	RETN
7C911232	8BFF	MOV EDI,EDI
7C911234	90	NOP
7C911235	90	NOP

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

7C911230	C3	RETN
7C911231	C3	RETN
7C911232	8BFF	MOV EDI,EDI
7C911234	90	NOP

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。

10002034	FF15 7CC00010	CALL NEAR DWORD PTR DS:[1000C07C]	kernel32.OpenMutexA
1000203A	8985 6CFBFFFF	MOV DWORD PTR SS:[EBP-494],EAX	
10002040	B8 CFA00319	MOV EAX,19D3A0CF	
10002045	3D 50000000	CMP EAX,50000000	

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。

0000002C	Key	2.	000F003F		\REGISTRY\MACHINE
00000040	Key	2.	00020019		\REGISTRY\MACHINE\SOFTWARE\Microsoft\Windows
00000044	KeyedEvent	42.	000F0003		\KernelObjects\CritSecOutOfMemoryEvent
00000048	Mutant	4.	001F0001		\BaseNamedObjects\MVFIRSTINSTANCE
00000050	Mutant	3.	001F0001		\BaseNamedObjects\WAIT
0000005C	Mutant	13.	00120001		\BaseNamedObjects\ShimCacheMutex
00000018	Port	3.	001F0001		
00000064	Process	9.	001F0FFF		
00000010	Section	41.	000F001F		
0000003C	Section	2.	000F0007		
0000004C	Section	2.	000F0007		
00000060	Section	13.	00000002		\BaseNamedObjects\ShimSharedMemory
00000034	Semaphore	2.	00100003	Count 0. of	
00000038	Semaphore	2.	00100003	Count 0. of	
00000068	Thread	6.	001F03FF		
00000020	WindowStation	79.	000F037F		\Windows\WindowStations\WinSta0
00000028	WindowStation	79.	000F037F		\Windows\WindowStations\WinSta0

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。

Registers (FPU)	
EAX	00000054
ECX	0012F094
EDX	7C91EB94 ntdll
EBX	10000000 AntiD
ESP	0012F108
EBP	0012F9D0
ESI	00000001
EDI	0012FD30
EIP	10002D3A AntiD
C 0	ES 0023 32bit

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。

00000040	KeyedEvent	42.	000F0003		\KernelObjects\CritSecOutOfMemoryEvent
0000004C	Mutant	3.	001F0001		\BaseNamedObjects\MVMAININSTANCE
00000048	Mutant	4.	001F0001		\BaseNamedObjects\MVFIRSTINSTANCE
00000054	Mutant	4.	001F0001		\BaseNamedObjects\WAIT
0000001C	Port	3.	001F0001		
00000014	Section	41.	000F001F		
00000038	Semaphore	2.	00100003	Count 0. of	
0000003C	Semaphore	2.	00100003	Count 0. of	
00000024	WindowStation	79.	000F037F		\Windows\WindowStations\WinSta0
0000002C	WindowStation	79.	000F037F		\Windows\WindowStations\WinSta0

现在来看看子进程所在 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。

10002DAD	FF15 78C00010	CALL NEAR DWORD PTR DS:[1000C078]	kernel32.WaitForSingleObject
10002DB3	8985 18FCFFFF	MOV DWORD PTR SS:[EBP-3E8],EAX	
10002DB9	B8 2A7A545E	MOV EAX,5E547A2A	

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。

0012F100	00000054	hObject = 00000054 (window)
0012F104	00001388	Timeout = 5000. ms
0012F108	0012FD30	

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。

10002DAD	FF15 78C00010	CALL NEAR DWORD PTR DS:[1000C078]	k
10002DB3	8985 18FCFFFF	MOV DWORD PTR SS:[EBP-3E8],EAX	
10002DB9	B8 2A7A545E	MOV EAX,5E547A2A	
10002DBE	3D 003A00A4	CMP EAX,A4003A00	
10002DC3	74 07	JE SHORT 10002DCC	A
10002DC5	B8 CD2D0010	MOV EAX,10002D00	
10002DCA	FFEB	JMP NEAR EAX	
10002DCC	DA8B 8D18FCFF	FIMUL DWORD PTR DS:[EAX],EAX	Backup
10002DD2	FF89 8D64F7FF	DEC DWORD PTR DS:[EAX]	Copy
10002DD8	FF83 8D64F7FF	INC DWORD PTR DS:[EAX]	Binary
10002DDE	FF80	INC DWORD PTR DS:[EAX]	Assemble
10002DE0	74 02	JE SHORT 10002DE2	Label
10002DE2	EB 58	JMP SHORT 10002DE4	Comment
10002DE4	B8 00A33DB2	MOV EAX,B23DA3D0	Breakpoint
10002DE9	3D 80CA2850	CMP EAX,5028CA80	Run trace
10002DEE	74 07	JE SHORT 10002DF0	
10002DF0	B8 F82D0010	MOV EAX,10002D00	
10002DF5	FFEB	JMP NEAR EAX	
10002DF7	D88B 956CFBFF	FIMUL DWORD PTR DS:[EAX],EAX	
10002DFD	FF52 FF	CALL NEAR DWORD PTR DS:[EAX-1]	
10002E00	15 80C00010	ADC EAX,1000C000	
10002E05	B8 B2B1E078	MOV EAX,78E0B1B2	
10002E0A	3D 10A58B89	CMP EAX,898BA510	
10002E0F	74 07	JE SHORT 10002E11	
10002E11	B8 192E0010	MOV EAX,10002E19	
10002E16	FFEB	JMP NEAR EAX	

10002DCD	8B8D 18FCFFFF	MOV ECX,DWORD PTR SS:[EBP-3E8]	
10002DD3	898D 64F7FFFF	MOV DWORD PTR SS:[EBP-89C],ECX	
10002DD9	83BD 64F7FFFF	CMP DWORD PTR SS:[EBP-89C],0	
10002DE0	74 02	JE SHORT 10002DE4	AntiDebu.10002DE4
10002DE2	EB 58	JMP SHORT 10002E3C	AntiDebu.10002E3C
10002DE4	B8 00A33DB2	MOV EAX,B23DA3D0	

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。

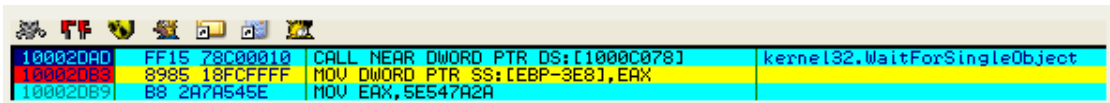
10002DCD	8B8D 18FCFFFF	MOV ECX,DWORD PTR SS:[EBP-3E8]	
10002DD3	898D 64F7FFFF	MOV DWORD PTR SS:[EBP-89C],ECX	
10002DD9	83BD 64F7FFFF	CMP DWORD PTR SS:[EBP-89C],0	
10002DE0	74 02	JE SHORT 10002DE4	AntiDebu.10002DE4
10002DE2	EB 58	JMP SHORT 10002E3C	AntiDebu.10002E3C
10002DE4	B8 00A33DB2	MOV EAX,B23DA3D0	

10002DCD	8B8D 18FCFFFF	MOV ECX,DWORD PTR SS:[EBP-3E8]	
10002DD3	898D 64F7FFFF	MOV DWORD PTR SS:[EBP-89C],ECX	
10002DD9	83BD 64F7FFFF	CMP DWORD PTR SS:[EBP-89C],0	
10002DE0	74 02	JE SHORT 10002DE4	AntiDebu.10002DE4
10002DE2	EB 58	JMP SHORT 10002E3C	AntiDebu.10002E3C
10002DE4	B8 00A33DB2	MOV EAX,B23DA3D0	
10002DE9	3D 80CA2850	CMP EAX,5028CA80	
10002DEE	74 07	JE SHORT 10002DF7	AntiDebu.10002DF7
10002DF0	B8 F82D0010	MOV EAX,10002D00	
10002DF5	FFEB	JMP NEAR EAX	
10002DF7	D88B 956CFBFF	FIMUL DWORD PTR DS:[EAX+FFFB6C95]	
10002DFD	FF52 FF	CALL NEAR DWORD PTR DS:[EDX-1]	
10002E00	15 80C00010	ADC EAX,1000C000	
10002E05	B8 B2B1E078	MOV EAX,78E0B1B2	
10002E0A	3D 10A58B89	CMP EAX,898BA510	
10002E0F	74 07	JE SHORT 10002E18	AntiDebu.10002E18
10002E11	B8 192E0010	MOV EAX,10002E19	
10002E16	FFEB	JMP NEAR EAX	
10002E18	DA8B 856CFBFF	FIMUL DWORD PTR DS:[EAX+FFFB6C85]	
10002E1E	FF50 FF	CALL NEAR DWORD PTR DS:[EAX-1]	
10002E21	15 18C00010	ADC EAX,1000C018	
10002E26	B8 94BF833F	MOV EAX,3F83BF94	
10002E2B	3D A07FEEC2	CMP EAX,C2EE7FA0	
10002E30	74 07	JE SHORT 10002E39	AntiDebu.10002E39
10002E32	B8 3A2E0010	MOV EAX,10002E3A	
10002E37	FFEB	JMP NEAR EAX	
10002E39	DCEB	FSUB ST(3),ST	
10002E3B	30B8 3AE96C93	XOR BYTE PTR DS:[EAX+936CE93A],BH	
10002E41	3D 500F176F	CMP EAX,6F170F50	
10002E46	74 07	JE SHORT 10002E4F	AntiDebu.10002E4F
10002E48	B8 502E0010	MOV EAX,10002E50	
10002E4D	FFEB	JMP NEAR EAX	
10002E4F	DA6A 00	FISUBR DWORD PTR DS:[EDX]	
10002E52	FF15 0CC00010	CALL NEAR DWORD PTR DS:[1000C00C]	kernel32.ExitProcess
10002E58	B8 1CF70F5A	MOV EAX,5A0FF71C	

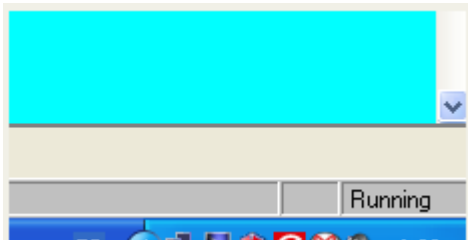
这里为了让子进程不因 WaitForSingleObject 等待超时而退出,我们将 WaitForSingleObject 的超时时间修改为 INFINITE(无穷大)。

0012F100	00000054	hObject = 00000054 (window)
0012F104	FFFFFFFF	Timeout = INFINITE
0012F108	0012FD30	
0012F10C	00210020	

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。



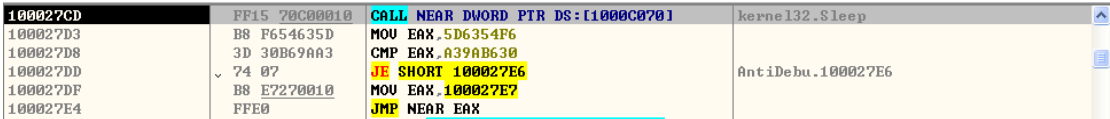
按 F9 键运行起来。



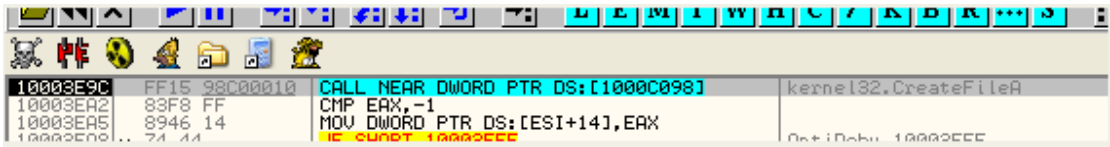
现在我们可以看到子进程已经运行起来了。也就是说子进程正在等待父进程释放 WAIT 这个互斥体的信号量。由于子进程所在 OD 的右下方现在显示的是 Running 状态,所以现在我们不能调试子进程了。转而我们现在来调试父进程。

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。



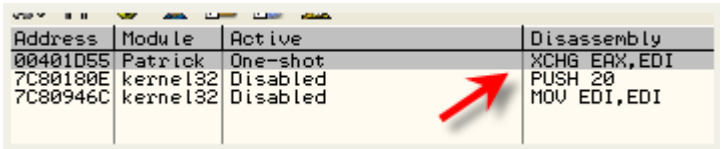
我们可以看到这里调用 Sleep 休眠片刻,我们继续 F9 运行起来,看看还会调用其他的什么重要的 API 函数。



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。

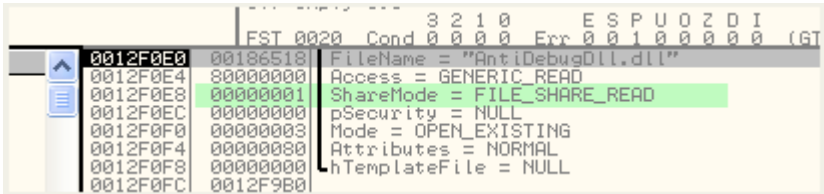
这里调用 CreateFileA 要打开 AntiDebugDll.dll,很可能要检测 AntiDebugDll.dll 中的代码是否被修改,防止被下 INT 3 断点。所以我们这里我们将之前设置的 INT 3 都删除掉。



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。

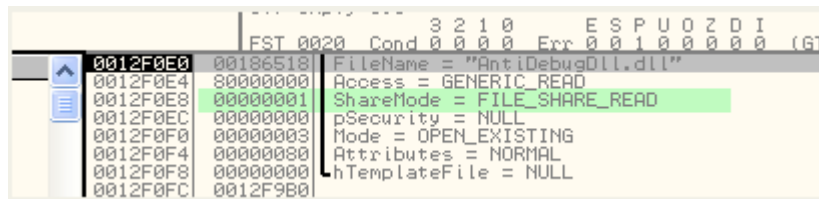
我们来看 CreateFileA 的参数情况:



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

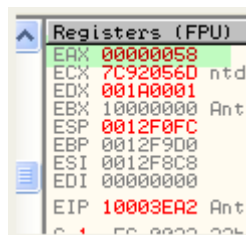
Set of bit flags that specifies how the object can be shared. If dwShareMode is 0, the object cannot be shared. Subsequent open operations on the object will fail, until the handle is closed.

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

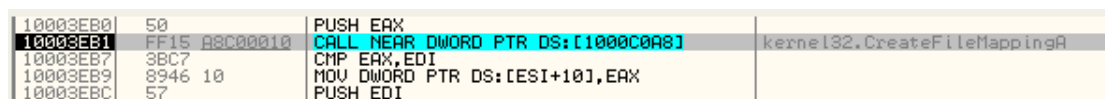
Value	Meaning
FILE_SHARE_DELETE	Windows NT only: Subsequent open operations on the object will succeed only if delete access is requested.
FILE_SHARE_READ	Subsequent open operations on the object will succeed only if read access is requested.
FILE_SHARE_WRITE	Subsequent open operations on the object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

如果是读取操作的话,会返回成功。

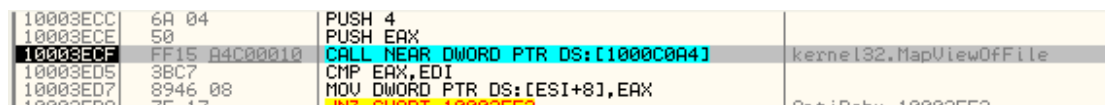
我们按 F8 键执行,可以看到返回的句柄值。



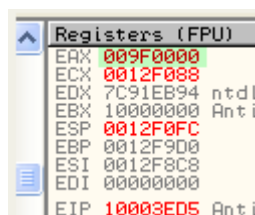
这里打开的 AntiDebugDll.dll 的文件句柄为 0x58。我们继续执行,看看还会调用什么 API 函数。



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	MZ.....♦....
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	!?!?!?!?!?!?!?!Th
009F0050	69 73 20 70 72 6F 67 72 61 60 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 00 00 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	!?!?!?!?!?!?!?!L
009F0090	11 0F FC 4C E5 29 BC 4C 11 0F A0 4C B1 29 BC 4C	!?!?!?!?!?!?!?!L
009F00A0	08 0A A5 4C F7 29 BC 4C F2 29 BD 4C 88 29 BC 4C	!?!?!?!?!?!?!?!L
009F00B0	1A 36 B6 4C F0 29 BC 4C 11 0F A1 4C E3 29 BC 4C	!?!?!?!?!?!?!?!L
009F00C0	11 0F 83 4C F3 29 BC 4C 11 0F 81 4C F3 29 BC 4C	!?!?!?!?!?!?!?!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..L0♦.ëëëD....
009F0100	00 00 00 00 E0 00 0E 21 08 01 07 00 00 B0 00 00ó.!!!.....
009F0110	00 80 00 00 00 00 00 00 04 64 00 00 00 10 00 00	.Ç.....ëd.....
009F0120	00 C0 00 00 00 00 00 10 00 10 00 00 00 10 00 00	.L.....!.....
009F0130	04 00 00 00 00 00 00 04 00 00 00 00 00 00 00 00	♦.....♦.....
009F0140	00 40 01 00 00 10 00 00 00 00 00 00 02 00 00 00	.@0.....@.....
009F0150	00 00 10 00 00 10 00 00 00 10 00 00 00 10 00 00	..!.....!.....
009F0160	00 00 00 00 10 00 00 00 00 DD 00 00 4E 00 00 00!..N....
009F0170	F4 D4 00 00 3C 00 00 00 00 00 00 00 00 00 00 00	!é..<.....

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	MZ.....♦....
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	!?!?!?!?!?!?!?!Th
10000050	69 73 20 70 72 6F 67 72 61 60 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 00 00 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	!?!?!?!?!?!?!?!L
10000090	11 0F FC 4C E5 29 BC 4C 11 0F A0 4C B1 29 BC 4C	!?!?!?!?!?!?!?!L
100000A0	08 0A A5 4C F7 29 BC 4C F2 29 BD 4C 88 29 BC 4C	!?!?!?!?!?!?!?!L
100000B0	1A 36 B6 4C F0 29 BC 4C 11 0F A1 4C E3 29 BC 4C	!?!?!?!?!?!?!?!L
100000C0	11 0F 83 4C F3 29 BC 4C 11 0F 81 4C F3 29 BC 4C	!?!?!?!?!?!?!?!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..L0♦.ëëëD....
10000100	00 00 00 00 E0 00 0E 21 08 01 07 00 00 B0 00 00ó.!!!.....
10000110	00 80 00 00 00 00 00 00 04 64 00 00 00 10 00 00	.Ç.....ëd.....
10000120	00 C0 00 00 00 00 00 10 00 10 00 00 00 10 00 00	.L.....!.....
10000130	04 00 00 00 00 00 00 04 00 00 00 00 00 00 00 00	♦.....♦.....
10000140	00 40 01 00 00 10 00 00 00 00 00 00 02 00 00 00	.@0.....@.....
10000150	00 00 10 00 00 10 00 00 00 10 00 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	
10003F04	0000 0000	MOV EAX,0	

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。

10003FF7	52	PUSH EDX	
10003FF8	FF15 94C00010	CALL NEAR DWORD PTR DS:[1000C094]	kernel32.GetFileSize
10003FFE	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 98C30010	PUSH 1000C398	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 98C30010	PUSH 1000C398	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 98C30010	PUSH 1000C398	ASCII ".DLL"
10003F5C	50	PUSH EAX	
10003F5D	E8 6E230000	CALL 100062D0	AntiDebu.100062D0
10003F62	83C4 08	ADD ESP,8	
10003F65	85C0	TEST EAX,EAX	
10003F67	74 07	JE SHORT 10003F70	AntiDebu.10003F70
10003F69	C746 04 020000	MOV DWORD PTR DS:[ESI+4],2	
10003F70	8BCE	MOV ECX,ESI	
10003F72	E8 69FDFFFF	CALL 10003CE0	AntiDebu.10003CE0
10003F73	74 07	JE SHORT 10003F7A	
DS:[1000C094]=7C810C8F (kernel32.GetFileSize)			

继续:

10003C9F	FF15 A0C00010	CALL NEAR DWORD PTR DS:[1000C0A0]	kernel32.UnmapViewOfFile
10003CA5	897E 08	MOV DWORD PTR DS:[ESI+8],EDI	
10003CA8	8B46 1A	MOV EAX,DWORD PTR DS:[ESI+1A]	

删除 9F0000 地址处的内存映射,说明不再需要这些字节了。

0012F0F0	10003CB8	CALL to CloseHandle from AntiDebu.10003CB6	
0012F0F4	00000054	hObject = 00000054 (window)	
0012F0F8	0012F9B0		

关闭掉文件句柄。

1000388C	FF15 98C00010	CALL NEAR DWORD PTR DS:[1000C098]	kernel32.CreateFileA
10003892	8BF8	MOV EDI,EAX	
10003894	3BFB	CMP EDI,EBX	
10003896	897C24 14	MOV DWORD PTR SS:[ESP+14],EDI	

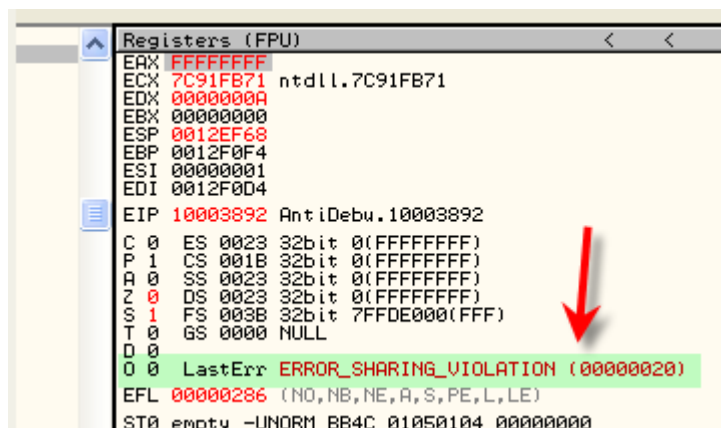
这里又调用了 CreateFileA,参数如下:

0012EF4C	0012EFA8	FileName = "AntiDebugDll.dll"	
0012EF50	80000000	Access = GENERIC_READ	
0012EF54	00000000	ShareMode = 0	
0012EF58	00000000	pSecurity = NULL	
0012EF5C	00000003	Mode = OPEN_EXISTING	
0012EF60	00000000	Attributes = 0	
0012EF64	00000000	hTemplateFile = NULL	
0012EF68	0012F9B0		
0012EF6C	00000001		

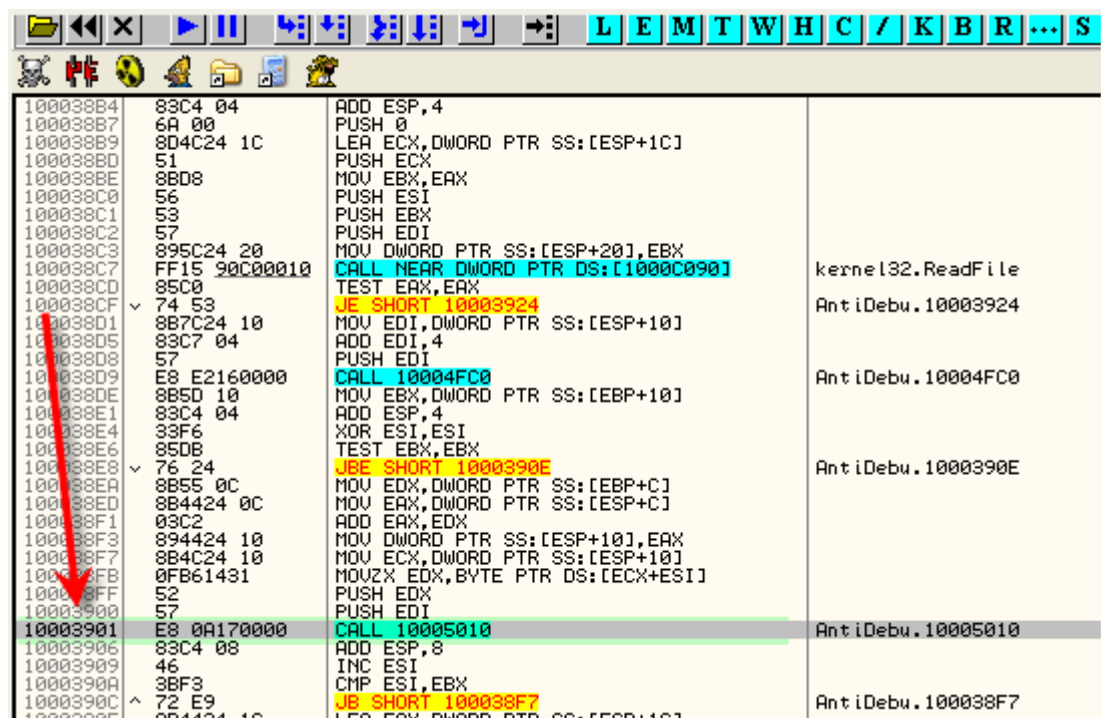
dwDesiredAccess 的值为 GENERIC_READ,但是 dwShareMode 的值被设置为了 0。MSDN 中的解释是如果 dwShareMode 被设置为零,该句柄不共享,再关闭之前不能被再次打开。这里如果我们按 F8 键执行的话,将打开文件失败。EAX 返回-1(即 FFFFFFFF)。

EAX	FFFFFFFF	
ECX	7C91FB71	nto
EDX	0000000A	
EBX	00000000	
ESP	0012EF68	
EBP	0012F0F4	
ESI	00000001	
EDI	0012F004	
EBP	10003892	On

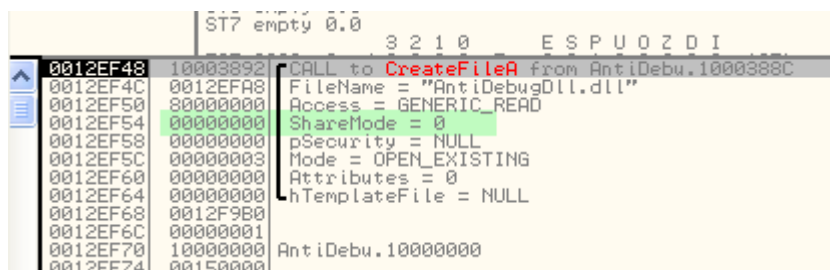
这里我们可以看到返回的句柄为 FFFFFFFF,即打开文件失败,LastError 为 ERROR_SHARING_VIOLATION。



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。

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	7C920560 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]	
100038D5	83C7 04	ADD EDI,4	

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	8B08	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 E2100000	CALL 10004FC0	AntiDebu.10004FC0
100038DE	8B5D 10	MOV EBX,DWORD PTR SS:[EBP+10]	
100038E0	83C4 04	ADD ESP,4	
100038E2	3F6 04	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	JR SHORT 100038F7	AntiDebu.100038F7

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
10003928	53	PUSH EBX	
1000392C	E8 5F280000	CALL 10006190	AntiDebu.10006190
10003931	83C4 04	ADD ESP,4	
10003934	33DB	XOR EBX,EBX	
10003937	8B45 04000000	MOV EBX,DWORD PTR DS:[10000000]	

关闭句柄。

这里又到了 CreateFileA 处。

10003E9C	FF15 98C00010	CALL NEAR DWORD PTR DS:[1000C098]	kernel32.CreateFileA
10003EA2	83F8 FF	CMP EAX,-1	
10003EA5	8946 14	MOV DWORD PTR DS:[ESI+14],EAX	
10003EA8	74 44	JE SHORT 10003EEE	AntiDebu.10003EEE
10003EAA	57	PUSH EDI	
10003EAB	57	PUSH EDI	

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。

01A24 (kernel32.CreateFileA)		
0012F0E0	00153E20	FileName = "C:\Documents and Settings\Ricardo\Escritorio\CONCURSO 87\carpeta sin t+itulo\Patrickori\Patrick.exe"
0012F0E4	80000000	Access = GENERIC_READ
0012F0E8	00000001	ShareMode = FILE_SHARE_READ
0012F0EC	00000000	Security = NULL
0012F0F0	00000003	Mode = OPEN_EXISTING
0012F0F4	00000000	Attributes = NORMAL
0012F0F8	00000000	hTemplateFile = NULL
0012F0FC	0012F0E0	
0012F100	00000001	

Registers (FPU)	
EAX	00000058
ECX	7C92056D ntdll.7C92056D
EDX	00150608
EBX	10000000 AntiDebu.10000000
ESP	0012F0FC
EBP	0012F9D0
ESI	0012F904
EDI	00000000
EIP	10003EA2 AntiDebu.10003EA2
CPU	FS:0023:32bit 0(FFFFFFFF)

这里由于是第一次打开,所以成功获取到了文件句柄。

10003EB0	50	PUSH EAX	
10003EB1	FF15 A8C00010	CALL NEAR DWORD PTR DS:[1000C0A8]	kernel32.CreateFileMappingA
10003EB7	3BC7	CMP EAX,EDI	
10003EB9	8946 10	MOV DWORD PTR DS:[ESI+10],EAX	
10003EBC	57	PUSH EDI	
10003EBD	75 08	JNZ SHORT 10003ECA	AntiDebu.10003ECA
10003EBF	FF15 18C00010	CALL NEAR DWORD PTR DS:[1000C018]	kernel32.CloseHandle
10003EC5	5F	POP EDI	
10003EC6	32C0	XOR AL,AL	
10003EC8	5E	POP ESI	
10003EC9	C3	RETN	

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

10003ECC	6A 04	PUSH 4	
10003ECE	50	PUSH EAX	
10003ECF	FF15 A4C00010	CALL NEAR DWORD PTR DS:[1000C0A4]	kernel32.MapViewOfFile
10003ED5	3BC7	CMP EAX,EDI	
10003ED7	8946 08	MOV DWORD PTR DS:[ESI+8],EAX	
10003EDA	75 17	JNZ SHORT 10003EF3	AntiDebu.10003EF3
10003EDC	8B46 10	MOV EAX,DWORD PTR DS:[ESI+10]	
10003EDF	8B3D 18C00010	MOV EDI,DWORD PTR DS:[1000C018]	kernel32.CloseHandle
10003EE5	50	PUSH EAX	
10003EE7	50	PUSH EAX	

创建内存映射,映射到了 AF0000 这个地址处。

Registers (FPU)	
EAX	00AF0000
ECX	0012F088
EDX	7C91EB94 ntd
EBX	10000000 Ant
ESP	0012F0FC
EBP	0012F9D0
ESI	0012F904
EDI	00000000
EIP	10003ED5 Ant

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。

判断文件扩展名。

10003C9F	FF15 00C00010	CALL NEAR DWORD PTR DS:[1000C0A0]	kernel32.UnmapViewOfFile
10003CA5	897E 08	MOV DWORD PTR DS:[ESI+8],EDI	
10003CA8	8B46 10	MOV EAX,DWORD PTR DS:[ESI+10]	
10003CAB	3BC7	CMP EAX,EDI	
10003CAD	8B1D 18C00010	MOV EBX,DWORD PTR DS:[1000C018]	kernel32.CloseHandle

删除内存映射,貌似没有做文件内容的检测呀。

100036D2	FF15 00C00010	CALL NEAR DWORD PTR DS:[1000C000]	kernel32.VirtualProtectEx
100036D8	C1ED 04	SHR EBP,4	
100036DB	3BEB	CMP EBP,EBX	
100036DD	895C24 10	MOV DWORD PTR SS:[ESP+10],EBX	

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。

0012EE74	00000064	hProcess = 00000064 (window)
0012EE78	00401000	Address = Patrick.00401000
0012EE7C	00005000	Size = 5000 (20480.)
0012EE80	00000040	NewProtect = PAGE_EXECUTE_READWRITE
0012EE84	0012EE9C	pOldProtect = 0012EE9C
0012FF88	0012F900	

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。

100036E1	895C24 18	MOV DWORD PTR SS:[ESP+18],EBX	
100036E5	7E 45	JLE SHORT 10003720	AntiDebu.1000372C
100036E7	8B1D 9CC00010	MOV EBX,DWORD PTR DS:[1000C09C]	kernel32.ReadProcessMemory
100036ED	8D49 00	LEA ECX,DWORD PTR DS:[ECX]	
100036F0	8D4C24 10	LEA ECX,DWORD PTR SS:[ESP+10]	
100036F4	51	PUSH ECX	
100036F5	6A 10	PUSH 10	
100036F7	8D5424 24	LEA EDX,DWORD PTR SS:[ESP+24]	
100036FB	52	PUSH EDX	
100036FC	56	PUSH ESI	
100036FD	57	PUSH EDI	
100036FE	FFD3	CALL NEAR EBX	
10003700	8D4424 1C	LEA EAX,DWORD PTR SS:[ESP+1C]	

这里调用 ReadProcessMemory 读取进程内容。

0012EE70	10003700	CALL to ReadProcessMemory from AntiDebu
0012EE74	00000064	hProcess = 00000064 (window)
0012EE78	00401000	pBaseAddress = 401000
0012EE7C	0012EEA4	Buffer = 0012EEA4
0012EE80	00000010	BytesToRead = 10 (16.)
0012EE84	0012EE98	pBytesRead = 0012EE98
0012EE88	0012F9D0	

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。

10003720	FF15 4CC00010	CALL NEAR DWORD PTR DS:[1000C04C]	kernel32.WriteProcessMemory
10003726	83C6 10	ADD ESI,10	
10003729	4D	DEC EBP	
1000372A	75 C4	JNZ SHORT 100036F0	AntiDebu.100036F0
1000372C	8D4C24 68	LEA ECX,DWORD PTR SS:[ESP+68]	
10003730	51	PUSH ECX	
10003731	F8 3D180000	CALL 10004F70	AntiDebu.10004F70

0012EE74	00000064	hProcess = 00000064 (window)
0012EE78	00401000	Address = 401000
0012EE7C	0012EEA4	Buffer = 0012EEA4
0012EE80	00000010	BytesToWrite = 10 (16.)
0012EE84	0012EEA0	pBytesWritten = 0012EEA0
0012EE88	0012F9D0	

003E0000	0000E000			Map	Rw	Rw	
003F0000	00001000			Priv	Rw	Rw	
00400000	00001000	Patrick	.text	PE header	Imag	RwE	
00401000	00005000	Patrick	.rdata	code	Imag	RwE	
00406000	00002000	Patrick	.data	imports	Imag	RwE	
00408000	00001000	Patrick	.rsrsc	data	Imag	RwE	
00409000	00005000	Patrick		resources	Imag	RwE	
00470000	00007000				Map	R E	
00530000	00002000				Map	R E	
00540000	00103000				Map	R	

起始地址为 401000 的这个区段是主程序的代码段。

我们先来看看 401000 地址处的代码。

00401011	6F	OUTS DX,DWORD PTR ES:[EDI]	I/O command
00401012	DCAB 5C10061	FSUBR QUORD 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 DS:[EBP+9B6595ED],EDX	
00401024	01B1 28FD033A	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:[ESI]	
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 50FDC4BF	CMP BYTE PTR DS:[EAX+FFC4FD50],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	BytesWritten = 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:[1000C000]	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	LhMutex = 00000050 (window)		
0012F108	0012FD30			
0012F10C	00150640			
0000000C	File (UI)	2.	00100020	\\Device\\HarddiskVolume1\\Documents and Settings\\Administrator\\My Recent Documents\\
0000002C	Key	2.	000F003F	\\REGISTRY\\MACHINE
00000040	Key	2.	00020019	\\REGISTRY\\MACHINE\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Policies\\
00000044	KeyedEvent	37.	000F0003	\\KernelObjects\\CriticalSectionOfMemoryEvent
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.	001F00FF	

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
10002BB8	B8 3611C531	MOV EAX,31C51136	
10002BC0	3D 3008F6CF	CMP EAX,CFF60830	
10002BC5	74 07	JE SHORT 10002BCE	AntiDebu.10002BCE

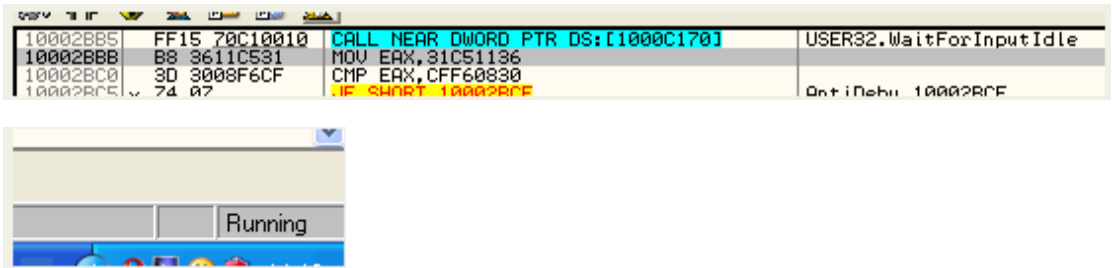
这里调用的是 WaitForInputIdle 这个函数,我们来看看 MSDN 中的说明。

Remarks

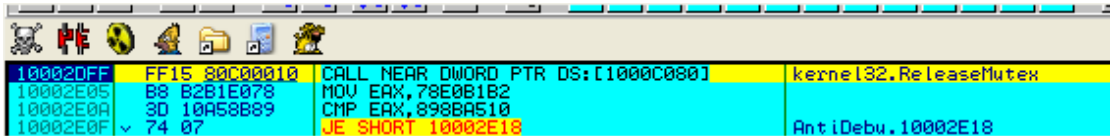
The **WaitForInputIdle** function enables a thread to suspend its execution until a specified process has finished its initialization and is waiting for user input with no input pending. This can be useful for synchronizing a parent process and a newly created child process. When a parent process creates 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。

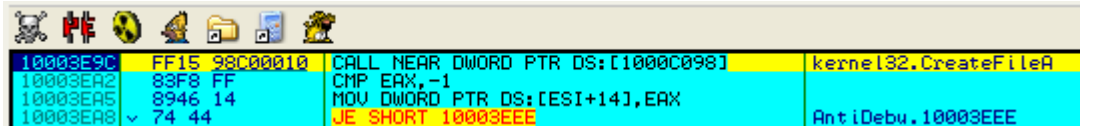
我们直接对下一条指令处设置一个断点,然后运行起来。



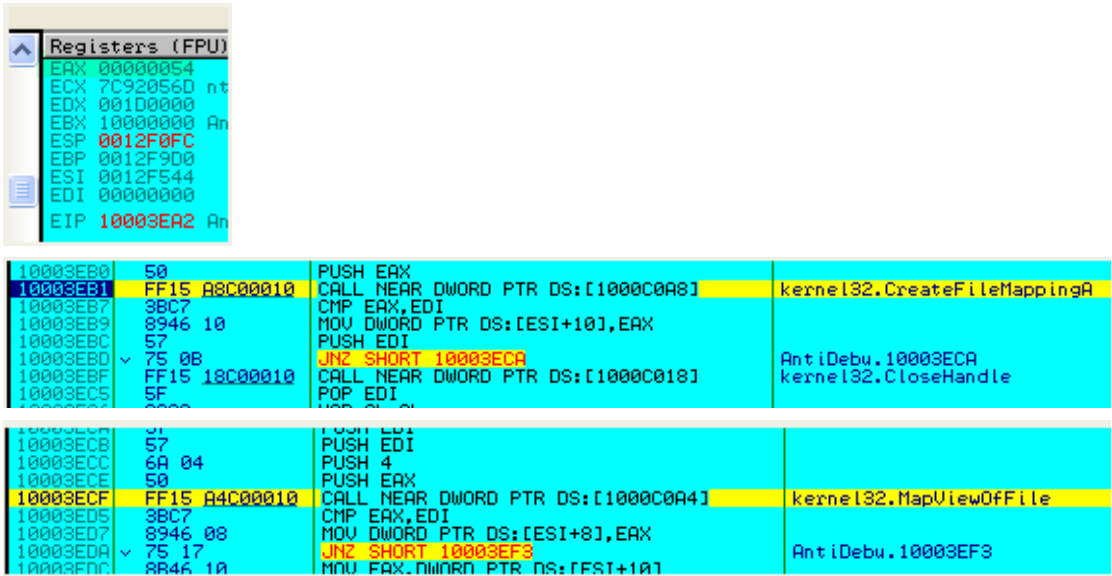
我们可以看到父进程现在已经处于运行状态了,我们现在接着来看子进程。



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。



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	8B06	MOV EAX,DWORD PTR DS:[ESI]	
10003F03	68 90C30010	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 1000C398	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	
10003F67	74 07	JE SHORT 10003F70	AntiDebu.10003F70

10003C9F	FF15 A0C00010	CALL NEAR DWORD PTR DS:[1000C0A0]	kernel32.UnmapViewOfFile
10003CA5	897E 08	MOV DWORD PTR DS:[ESI+8],EDI	
10003CA8	8B46 10	MOV EAX,DWORD PTR DS:[ESI+10]	
10003CAB	8B47	MOV EAX,DWORD PTR DS:[ESI+11]	

现在到了关键部位了。

1000388C	FF15 98C00010	CALL NEAR DWORD PTR DS:[1000C098]	kernel32.CreateFileA
10003892	8BF8	MOV EDI,EAX	
10003894	3BFB	CMP EDI,EBX	

0012EF4C	0012EFA8	FileName = "AntiDebugDll.dll"
0012EF50	80000000	Access = GENERIC_READ
0012EF54	00000000	ShareMode = 0
0012EF58	00000000	Security = NULL
0012EF5C	00000003	Mode = OPEN_EXISTING
0012EF60	00000000	Attributes = 0
0012EF64	00000000	hTemplateFile = NULL
0012EF68	0012FD30	
0012EF6C	00000001	

跟之前一样我们将 dwShareMode 修改为 1。

Registers (FPU)	
EAX	00000054
ECX	7C92056D ntd
EDX	001E0000
EBX	00000000
ESP	0012EF68
EBP	0012F0F4
ESI	00000001
EDI	0012F0D4
EIP	10003892 Ant

成功返回句柄。

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]	

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。

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	8B47	MOV EAX,DWORD PTR DS:[ESI+11]	

接着这里关闭文件句柄。

10003E9C	FF15 98C00010	CALL NEAR DWORD PTR DS:[1000C098]	kernel32.CreateFileA
10003EA2	83F8 FF	CMP EAX,-1	
10003EA5	8946 14	MOV DWORD PTR DS:[ESI+14],EAX	

这里再次调用 CreateFileA 打开 Patrick.exe。

0012F0E0	00154C00	FileName = "C:\Documents and Settings\Ricardo\Escritorio\ESCR	
0012F0E4	80000000	Access = GENERIC_READ	
0012F0E8	00000001	ShareMode = FILE_SHARE_READ	
0012F0EC	00000000	pSecurity = NULL	
0012F0F0	00000003	Mode = OPEN_EXISTING	
0012F0F4	00000000	Attributes = NORMAL	
0012F0F8	00000000	hTemplateFile = NULL	
0012F0FC	0012FD30		

成功返回句柄。

10003EAF	57	PUSH EDI	
10003EB0	50	PUSH EAX	
10003EB1	FF15 A8C00010	CALL NEAR DWORD PTR DS:[1000C0A8]	kernel32.CreateFileMappingA
10003EB7	3BC7	CMP EAX,EDI	
10003EB9	8946 10	MOV DWORD PTR DS:[ESI+10],EAX	
10003EBB	57	PUSH EDI	

这部分与父进程是一样的,我们不再赘述。

10003ECC	6A 04	PUSH 4	
10003ECE	50	PUSH EAX	
10003ECF	FF15 A4C00010	CALL NEAR DWORD PTR DS:[1000C0A4]	kernel32.MapViewOfFile
10003ED5	3BC7	CMP EAX,EDI	
10003ED7	8946 08	MOV DWORD PTR DS:[ESI+8],EAX	

10003C9F	FF15 A0C00010	CALL NEAR DWORD PTR DS:[1000C0A0]	kernel32.UnmapViewOfFile
10003CA5	897E 08	MOV DWORD PTR DS:[ESI+8],EDI	
10003CA8	8B46 10	MOV EAX,DWORD PTR DS:[ESI+10]	
10003CAB	3BC7	CMP EAX,EDI	
10003CAD	8B46 10	MOV EAX,DWORD PTR DS:[ESI+10]	

0012EF4C	0012EFA8	FileName = "C:\Documents and Settings\Ricardo\Escritorio\ESCRITORIO\CONCURSO 87\carpet	
0012EF50	80000000	Access = GENERIC_READ	
0012EF54	00000000	ShareMode = 0	
0012EF58	00000000	pSecurity = NULL	
0012EF5C	00000003	Mode = OPEN_EXISTING	
0012EF60	00000000	Attributes = 0	
0012EF64	00000000	hTemplateFile = NULL	
0012EF68	0012FD30		
0012EF6C	00000001		

这里我们依然将 dwShareMode 修改为 1。

Registers (FPU)	
EAX	00000054
ECX	7C920560 ntdll
EDX	00200000
EBX	00000000
ESP	0012EF68
EBP	0012F0F4
ESI	00000001
EDI	0012F0D4
EIP	10003892 AntiDebu.10003924

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。

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]	

0012EF4C	0012EFA8	FileName = "C:\Documents and Settings\Ricardo\Escritorio\ESCRITORIO\CONCURSO 87\carpet	
0012EF50	80000000	Access = GENERIC_READ	
0012EF54	00000000	ShareMode = 0	
0012EF58	00000000	pSecurity = NULL	
0012EF5C	00000003	Mode = OPEN_EXISTING	
0012EF60	00000000	Attributes = 0	
0012EF64	00000000	hTemplateFile = NULL	
0012EF68	0012FD30		
0012EF6C	00000001		
0012EF70	10000000	AntiDebu.10000000	

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。

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

又是调用 ReadFile 进行检查。

100032E6	FF15 04C00010	CALL NEAR DWORD PTR DS:[1000C004]	kernel32.GetCurrentProcess
100032EC	50	PUSH EAX	
100032ED	8D8D FCF9FFFF	LEA ECX,DWORD PTR SS:[EBP-604]	
100032F3	EB 00020000	CALL 100035D0	AntiDebu.100035D0

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。

Registers (FPU)	
EAX	FFFFFFFF
ECX	00154D44
EDX	00000001
EBX	10000000
ESP	0012F0F8
EBP	0012F9D0
ESI	00000001
EDI	0012FD30
EIP	100032EC

100036D2	FF15 00C00010	CALL NEAR DWORD PTR DS:[1000C000]	kernel32.VirtualProtectEx
100036D8	C1ED 04	SHR EBP,4	
100036DB	3EBE	CMP EBP,EBX	
100036DD	895C24 10	MOV DWORD PTR SS:[ESP+10],EBX	

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,WriteProcessMemory 读写内存做铺垫。

100036D0	895C24 10	MOV DWORD PTR SS:[ESP+10],EBX	
100036E1	895C24 18	MOV DWORD PTR SS:[ESP+18],EBX	
100036E5	7E 45	JLE SHORT 10003720	AntiDebu.10003720
100036E7	8B10 9CC00010	MOV EBX,DWORD PTR DS:[1000C09C]	kernel32.ReadProcessMemory
100036ED	8D49 00	LEA ECX,DWORD PTR DS:[ECX]	
100036F0	8D4C24 10	LEA ECX,DWORD PTR SS:[ESP+10]	
100036F4	51	PUSH ECX	
100036F5	6A 10	PUSH 10	
100036F7	8D5424 24	LEA EDX,DWORD PTR SS:[ESP+24]	
100036FB	52	PUSH EDX	
100036FC	56	PUSH ESI	
100036FD	57	PUSH EDI	
100036FE	FFD3	CALL NEAR EBX	
10003700	8D4424 1C	LEA EAX,DWORD PTR SS:[ESP+1C]	

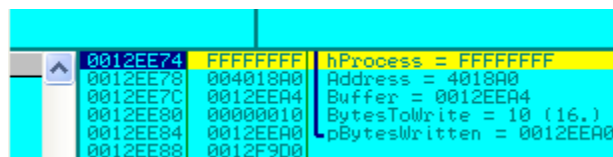
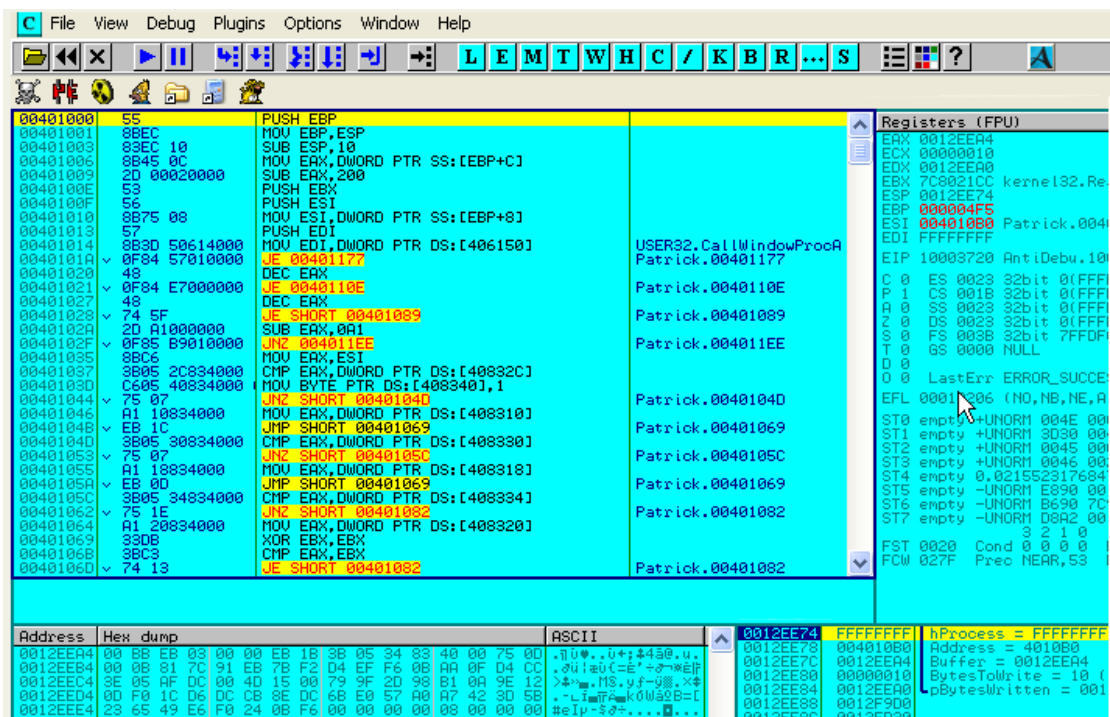
接着又是 WriteProcessMemory。

1000371E	56	PUSH ESI	
1000371F	57	PUSH EDI	
10003720	FF15 4CC00010	CALL NEAR DWORD PTR DS:[1000C04C]	kernel32.WriteProcessMemory
10003726	83C6 10	ADD ESI,10	
10003729	40	DEC EBP	
1000372A	75 C4	JNZ SHORT 100036F0	AntiDebu.100036F0
1000372C	8D4C24 68	LEA ECX,DWORD PTR SS:[ESP+68]	
10003730	51	PUSH ECX	

这里又是写入起始地址为 401000,长度为 16 的内容。

00401000	55	PUSH EBP	
00401001	8BEC	MOV EBP,ESP	
00401003	83EC 10	SUB ESP,10	
00401006	8B45 0C	MOV EAX,DWORD PTR SS:[EBP+C]	
00401009	2D 00020000	SUB EAX,200	
0040100E	53	PUSH EBX	
0040100F	52	PUSH ESI	

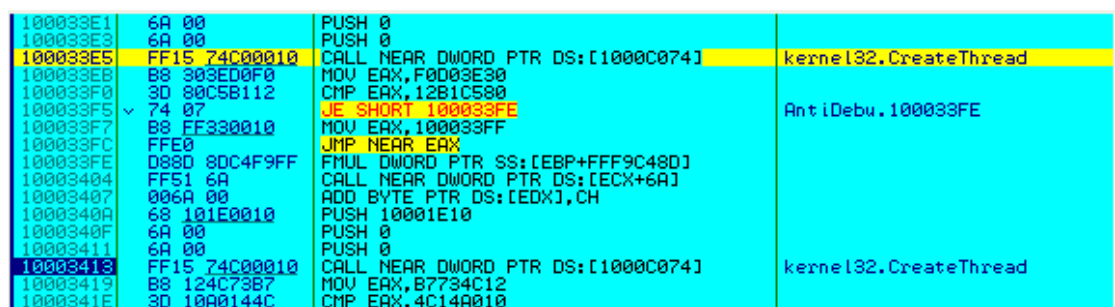
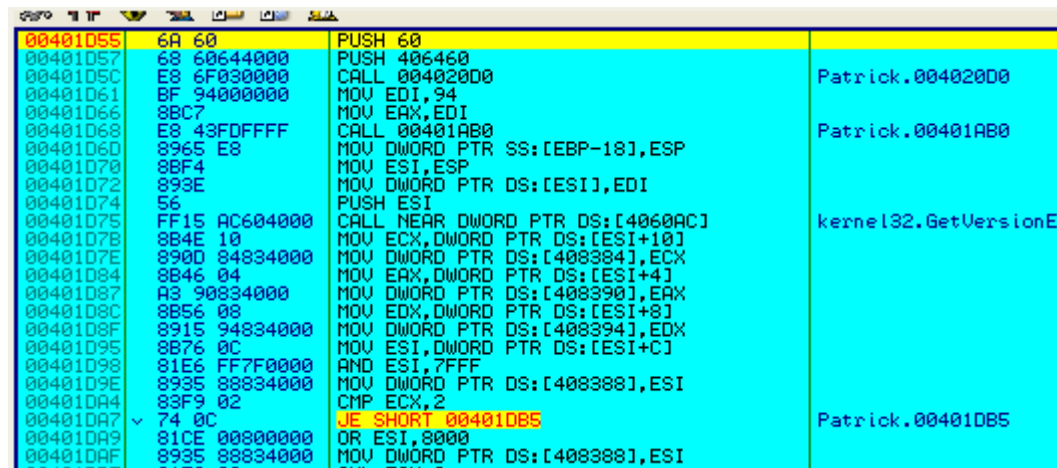
我们定位到 401000 看看。



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。

可能看到的的确是入口点的样子,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。

10002140	51	PUSH ECX	
10002141	56	PUSH ESI	
10002142	8B35 70C00010	MOV ESI,DWORD PTR DS:[1000C070]	kernel32.Sleep
10002148	68 C8000000	PUSH 0C8	
1000214D	FFD6	CALL NEAR ESI	
1000214F	C74424 04 0000	MOV DWORD PTR SS:[ESP+4],0	
10002157	50	PUSH EAX	
10002158	64:A1 18000000	MOV EAX,DWORD PTR FS:[18]	
1000215E	3E:8B40 30	MOV EAX,DWORD PTR DS:[EAX+30]	
10002162	3E:0FB640 02	MOVZX EAX,BYTE PTR DS:[EAX+2]	
10002167	894424 08	MOV DWORD PTR SS:[ESP+8],EAX	
10002168	58	POP EAX	
1000216C	8B4424 04	MOV EAX,DWORD PTR SS:[ESP+4]	
10002170	85C0	TEST EAX,EAX	
10002172	74 D4	JE SHORT 10002148	AntiDebu.10002148
10002174	6A 00	PUSH 0	
10002176	FF15 0CC00010	CALL NEAR DWORD PTR DS:[1000C00C]	kernel32.ExitProcess
1000217C	CC	INT3	
1000217D	90	NOP	
1000217E	90	NOP	
1000217F	90	NOP	
10002180	68 0E000780	PUSH 8007000E	
10002185	E8 F6FFFFFF	CALL 10001180	AntiDebu.10001180
1000218A	CC	INT3	
1000218B	90	NOP	
1000218C	90	NOP	
1000218D	90	NOP	

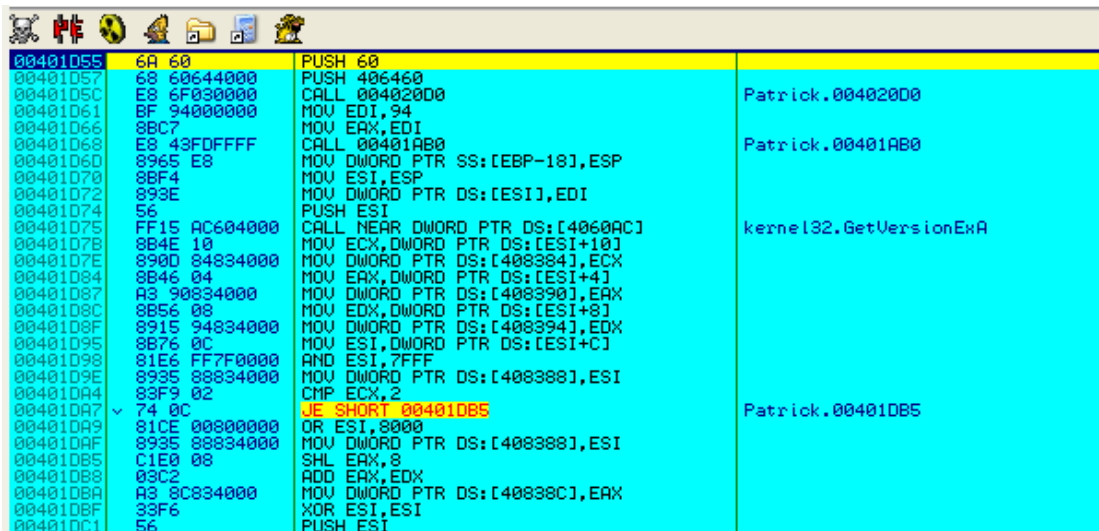
我们定位该线程的起始地址看看,这是是创建的第一个线程,我们继续往下看,看看会不会再创建其他进程。

10001E10	55	PUSH EBP	
10001E11	8BEC	MOV EBP,ESP	
10001E13	83E4 F8	AND ESP,FFFFFFF8	
10001E16	81EC 5C030000	SUB ESP,35C	
10001E1C	A1 8C050110	MOV EAX,DWORD PTR DS:[100105BC]	
10001E21	3345 04	XOR EAX,DWORD PTR SS:[EBP+4]	
10001E24	53	PUSH EBX	
10001E25	56	PUSH ESI	
10001E26	57	PUSH EDI	
10001E27	898424 64030000	MOV DWORD PTR SS:[ESP+364],EAX	
10001E2E	33C0	XOR EAX,EAX	
10001E30	B9 49000000	MOV ECX,49	
10001E35	8D7C24 14	LEA EDI,DWORD PTR SS:[ESP+14]	
10001E39	F3:AB	REP STOS DWORD PTR ES:[EDI]	
10001E3B	B9 88000000	MOV ECX,88	
10001E40	8DBC24 3C010000	LEA EDI,DWORD PTR SS:[ESP+13C]	
10001E47	898424 38010000	MOV DWORD PTR SS:[ESP+138],EAX	
10001E4E	F3:AB	REP STOS DWORD PTR ES:[EDI]	
10001E50	C74424 10 2801	MOV DWORD PTR SS:[ESP+10],128	
10001E58	EB 06	JMP SHORT 10001E60	AntiDebu.10001E60
10001E5A	8D9B 00000000	LEA EBX,DWORD PTR DS:[EBX]	
10001E60	68 C8000000	PUSH 0C8	
10001E65	FF15 70C00010	CALL NEAR DWORD PTR DS:[1000C070]	kernel32.Sleep
10001E68	6A 00	PUSH 0	
10001E6D	6A 02	PUSH 2	
10001E6F	E8 94340000	CALL 10005308	<JMP.&KERNEL32.CreateToolhelp32S
10001E74	8D4C24 10	LEA ECX,DWORD PTR SS:[ESP+10]	
10001E78	51	PUSH ECX	
10001E79	50	PUSH EAX	
10001E7A	894424 14	MOV DWORD PTR SS:[ESP+14],EAX	
10001E7E	E8 7F340000	CALL 10005302	<JMP.&KERNEL32.Process32First>
10001E83	85C0	TEST EAX,EAX	
10001E85	74 D9	JE SHORT 10001E60	AntiDebu.10001E60

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

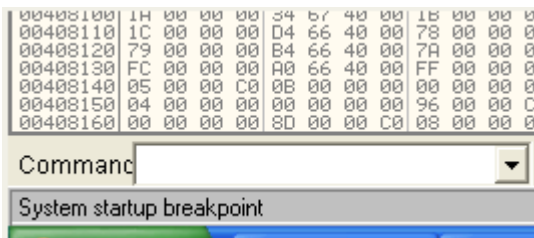
0012F0EC	10003419	CALL to CreateThread from AntiDebu.10003413
0012F0F0	00000000	pSecurity = NULL
0012F0F4	00000000	StackSize = 0
0012F0F8	10001E10	ThreadFunction = AntiDebu.10001E10
0012F0FC	00000000	pThreadParam = NULL
0012F100	00000004	CreationFlags = CREATE_SUSPENDED
0012F104	0012F394	pThreadId = 0012F394
0012F108	0012FD30	
0012F10C	0012F0F8	

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。

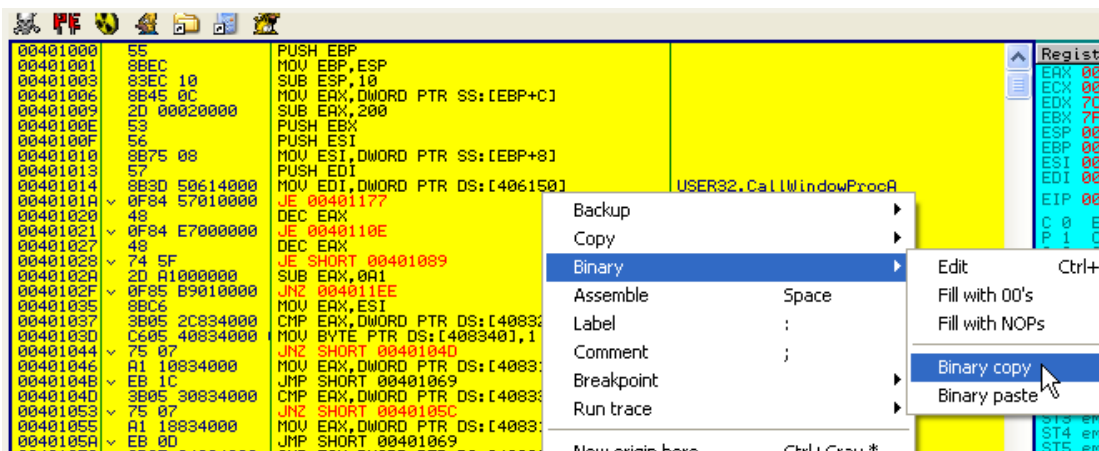


现在我们将成功断在了 OEP 处,下面我们将进行 dump。

现在我们将再打开一个 OD,加载 Patrick.exe,断在了系统断点处。



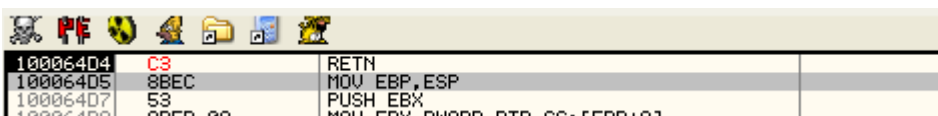
现在我们将子进程代码段的所有字节都拷贝到这个新开的 OD 中。



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。

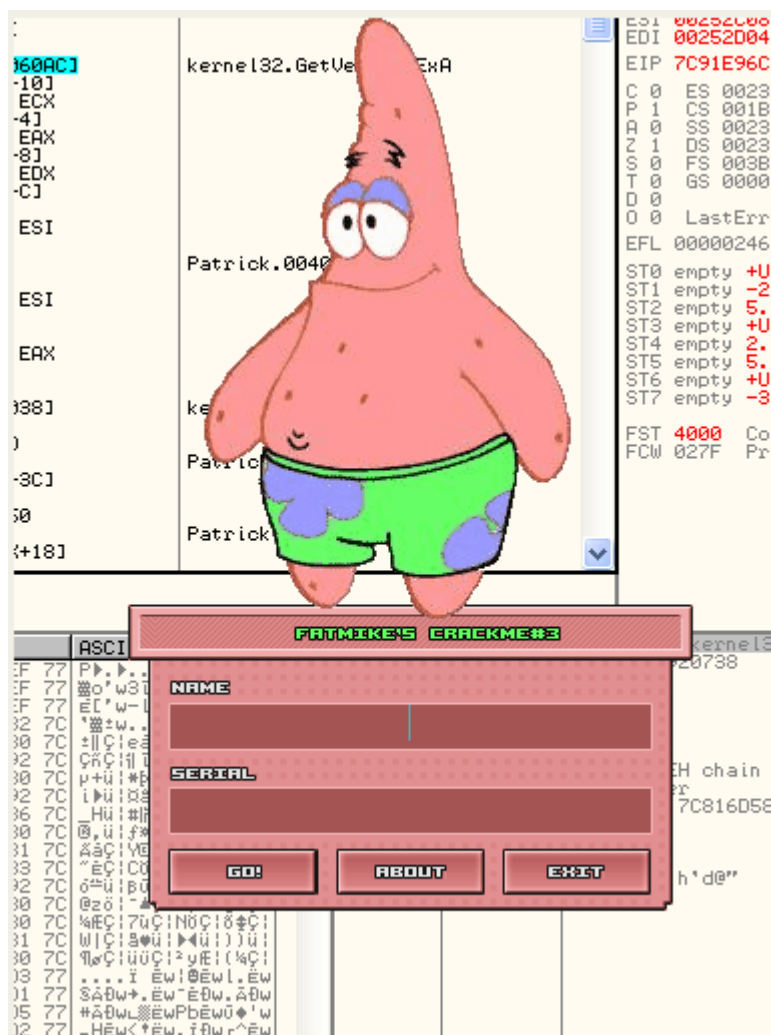


接着我们打开 PEEditor 程序的所有区段访问权限都修改为 E0000020(可读可写可执行)。

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。

00401410	68 0500400	PUSH 40005	
00401415	FF35 04834000	PUSH DWORD PTR DS:[408304]	
00401418	6A 72	PUSH 72	
0040141D	FF15 64614000	CALL NEAR DWORD PTR DS:[406164]	WINMM.PlaySoundA
00401423	33C0	XOR EAX,EAX	
00401425	E9 8AFEFFFF	JMP 004012B4	Patrick.004012B4
0040142A	E8 15050000	CALL 00401944	<JMP.&AntiDebugDll.#1>
0040142F	6A 00	PUSH 0	
00401431	FF15 38604000	CALL NEAR DWORD PTR DS:[406038]	kernel32.GetModuleHandleA
00401437	8B35 40614000	MOV ESI,DWORD PTR DS:[406140]	USER32.GetDlgItem
0040143D	68 E9030000	PUSH 3E9	
00401442	53	PUSH EBX	
00401410	FF15 64614000	CALL NEAR DWORD PTR DS:[406164]	WINMM.PlaySoundA
00401423	33C0	XOR EAX,EAX	
00401425	E9 8AFEFFFF	JMP 004012B4	Patrick.004012B4
0040142A	90	NOP	
0040142B	90	NOP	
0040142C	90	NOP	
0040142D	90	NOP	
0040142E	90	NOP	
0040142F	6A 00	PUSH 0	
00401431	FF15 38604000	CALL NEAR DWORD PTR DS:[406038]	kernel32.GetModuleHandleA
00401437	8B35 40614000	MOV ESI,DWORD PTR DS:[406140]	USER32.GetDlgItem
0040143D	68 E9030000	PUSH 3E9	
00401442	53	PUSH EBX	

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:

Address	Hex dump	Data	Comment
0040013E	0000	00 0000	DLLCharacteristics = 0
00400140	00001000	00 00100000	SizeOfStackReserve = 100000 (104
00400144	00100000	00 00001000	SizeOfStackCommit = 1000 (4096.)
00400148	00001000	00 00100000	SizeOfHeapReserve = 100000 (1048
0040014C	00100000	00 00001000	SizeOfHeapCommit = 1000 (4096.)
00400150	00000000	00 00000000	LoaderFlags = 0
00400154	10000000	00 00000010	NumberOfRvaAndSizes = 10 (16.)
00400158	00000000	00 00000000	Export Table address = 0
0040015C	00000000	00 00000000	Export Table size = 0
00400160	3C6F0000	00 00006F3C	Import Table address = 6F3C
00400164	78000000	00 00000078	Import Table size = 78 (120.)
00400168	00900000	00 00009000	Resource Table address = 9000
0040016C	40090500	00 00050940	Resource Table size = 50940 (383
00400170	00000000	00 00000000	Exception Table address = 0
00400174	00000000	00 00000000	Exception Table size = 0
00400178	00000000	00 00000000	Certificate File pointer = 0

我们定位到导入表(IT)的起始地址为 406F3C。

Address	Hex dump	ASCII
00406F3C	18 71 00 00 00 00 00 00 00 00 00 00 00 00 00 00 2E 71 00 00	↑q.....q..
00406F4C	64 61 00 00 B4 6F 00 00 00 00 00 00 00 00 00 00 00 00 00 00	da..fo.....
00406F5C	38 71 00 00 00 60 00 00 EC 6F 00 00 00 00 00 00 00 00 00 00	8q...'.ýo.....
00406F6C	00 00 00 00 F8 71 00 00 38 60 00 00 C8 70 00 00°q..8'..p..
00406F7C	00 00 00 00 00 00 00 00 2C 73 00 00 14 61 00 00s..la..
00406F8C	BC 6F 00 00 00 00 00 00 00 00 00 00 00 E4 73 00 00	#o.....s..
00406F9C	08 60 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	█'.....ø..ç..
00406FAC	00 00 00 00 00 00 00 00 01 00 00 00 80 00 00 00 00 00 00 00ø..ç..
00406FBC	D8 73 00 00 64 73 00 00 CE 73 00 00 C0 73 00 00	is..ds..fs..ls..
00406FCC	AE 73 00 00 98 73 00 00 88 73 00 00 78 73 00 00	«s..ýs..és..xs..
00406FDC	56 73 00 00 46 73 00 00 38 73 00 00 00 00 00 00	Us..Fs..8s..
00406FEC	A2 71 00 00 B6 71 00 00 C6 71 00 00 D8 71 00 00	óq..Áq..áq..iq..
00406FFC	E8 71 00 00 D2 76 00 00 C0 76 00 00 AE 76 00 00	þq..ëv..lv..«v..
0040700C	9E 76 00 00 82 76 00 00 76 76 00 00 6A 76 00 00	xv..ëv..vv..jv..
0040701C	5A 76 00 00 4A 76 00 00 3C 76 00 00 2A 76 00 00	Zv..Jv..<v..*v..
0040702C	10 76 00 00 00 76 00 00 E6 75 00 00 CE 75 00 00	þv..v..pu..fu..
0040703C	04 75 00 00 00 75 00 00 00 75 00 00 7C 75 00 00	Ju..üu..su..tu..

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。

Address	Hex dump	ASCII
00406F3C	18 71 00 00 00 00 00 00 00 00 00 00 00 00 00 00 2E 71 00 00	↑q.....q..
00406F4C	64 61 00 00 B4 6F 00 00 00 00 00 00 00 00 00 00 00 00 00 00	da..fo.....
00406F5C	38 71 00 00 00 60 00 00 EC 6F 00 00 00 00 00 00 00 00 00 00	8q...'.ýo.....
00406F6C	00 00 00 00 F8 71 00 00 38 60 00 00 C8 70 00 00°q..8'..p..
00406F7C	00 00 00 00 00 00 00 00 2C 73 00 00 14 61 00 00s..la..
00406F8C	BC 6F 00 00 00 00 00 00 00 00 00 00 00 E4 73 00 00	#o.....s..
00406F9C	08 60 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	█'.....ø..ç..
00406FAC	00 00 00 00 00 00 00 00 01 00 00 00 80 00 00 00 00 00 00 00ø..ç..
00406FBC	D8 73 00 00 64 73 00 00 CE 73 00 00 C0 73 00 00	is..ds..fs..ls..
00406FCC	AE 73 00 00 98 73 00 00 88 73 00 00 78 73 00 00	«s..ýs..és..xs..

这里第一个 DLL 名称字符串指针为 40712E,我们一起来看看。

Address	Hex dump	ASCII
0040712E	57 49 4E 40 4D 2E 64 6C 6C 00 41 6E 74 69 44 65	WINMM.dll.AntiDe
0040713E	62 75 67 44 6C 6C 2E 64 6C 6C 00 00 EB 00 47 65	bugDll.dll..v.Ge
0040714E	74 41 43 50 00 00 5D 01 47 65 74 4C 6F 63 61 6C	tACP..J0GetLocal
0040715E	65 49 6E 66 6F 41 00 00 C8 01 47 65 74 56 65 72	eInfoA..00GetVer
0040716E	73 69 6F 6E 45 78 41 00 51 02 4D 75 6C 74 69 42	sionExA..00MultiB
0040717E	79 74 65 54 6F 57 69 64 65 43 68 61 72 00 69 03	yteToWideChar.i
0040718E	57 69 64 65 43 68 61 72 54 6F 4D 75 6C 74 69 42	WideCharToMultiB
0040719E	79 74 65 00 67 01 47 65 74 4D 6F 64 75 6C 65 48	yte.g0GetModuleH
004071AE	61 6E 64 6C 65 41 00 00 E7 00 46 72 65 65 52 65	andleA..p.FreeRe

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。

Address	Hex dump	ASCII
00406F3C	18 71 00 00 00 00 00 00 00 00 00 00 00 00 00 00 2E 71 00 00	↑q.....q..
00406F4C	64 61 00 00 18 71 00 00 00 00 00 00 00 00 00 00 00 00 00 00	da..↑q.....
00406F5C	2E 71 00 00 EC 6F 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.q..da..ýo.....
00406F6C	00 00 00 00 F8 71 00 00 38 60 00 00 C8 70 00 00°q..8'..p..
00406F7C	00 00 00 00 00 00 00 00 2C 73 00 00 14 61 00 00s..la..
00406F8C	BC 6F 00 00 00 00 00 00 00 00 00 00 00 E4 73 00 00	#o.....s..

保存修改到文件。

我们重启 OD,再来看看导入表。

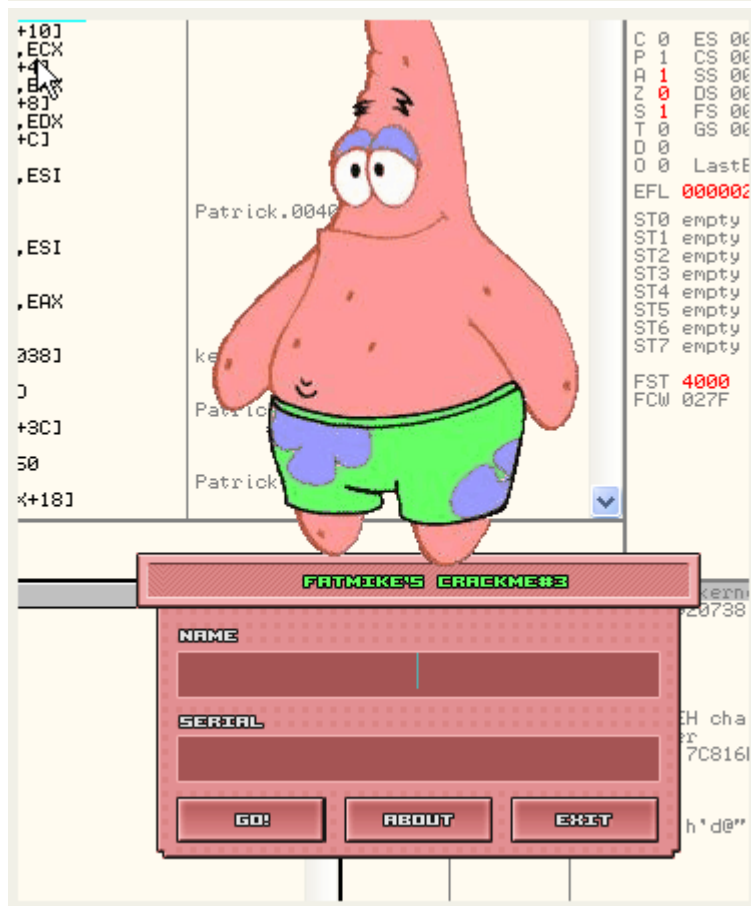
00400154	10000000	UU 00000010	NumberOfRvaAndSizes = 10 (16.)
00400158	00000000	DD 00000000	Export Table address = 0
0040015C	00000000	DD 00000000	Export Table size = 0
00400160	3C6F0000	DD 00006F3C	Import Table address = 6F3C
00400164	78000000	DD 00000078	Import Table size = 78 (120.)
00400168	00900000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

如下:

00400148	00001000	UU 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	40090500	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	00401055	Patrick	C:\Documents and Settings\Ricardo\Escritorio\ESCRITORIO\CONCURSO 87\carpeta sin t+itulo\Patrick\Patrick.exe
5B480000	00007000	5B4810B0	undwxfm	C:\WINDOWS\system32\undwxfm.dll
5D160000	00007000	5D161403	serwudrv	C:\WINDOWS\system32\serwudrv.dll
76B00000	00002000	76B02659	WINMM	C:\WINDOWS\system32\WINMM.dll
77D10000	00090000	77D1F538	USER32	C:\WINDOWS\system32\USER32.dll
77DB0000	0004C000	77DB70D4	ADVAPI32	C:\WINDOWS\system32\ADVAPI32.dll
77E50000	00091000	77E56284	RPCRT4	C:\WINDOWS\system32\RPCRT4.dll
77EF0000	00047000	77EF658A	GDI32	C:\WINDOWS\system32\GDI32.dll
7C800000	00101000	7C80B436	kernel32	C:\WINDOWS\system32\kernel32.dll
7C910000	000B0000	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。