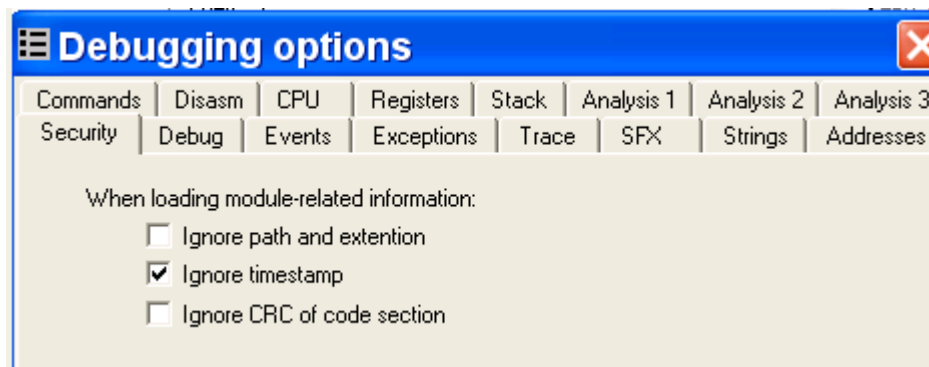


第二十章-OllyDbg 反调试之检测 OD 进程名

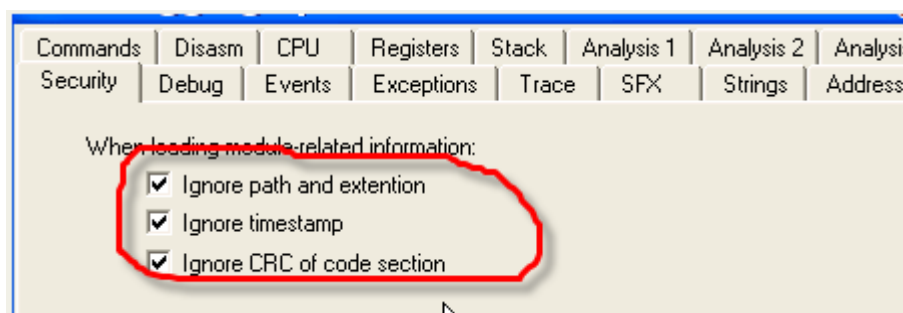
本章我们介绍通过查找 OD 进程名来进行反调试技巧,首先我们有必要对 OD 进行相应的设置。

我们选择主菜单项 Option-Debugging options-Security。



我们可以看到这里有 3 个复选框,我们知道通常情况下,我们给某个 API 函数设置断点,当我们重启 OD 后,刚刚给 API 函数设置的断点就被清除了。但是勾选上这 3 个复选框,再给 API 函数设置断点,重启 OD 后我们会发现刚刚设置的断点依然存在,这就避免了重启 OD 后需要重新给 API 设置断点的麻烦。

实际上,我也不知道 OD 关于这个功能的实现原理,我们只需要知道这 3 个复选框可以让我们设置的 API 断点在 OD 重启后仍然有效。



这个设置我觉得很有必要,避免了很多麻烦,现在我们开始讨论通过查找 OD 进程名来进行反调试的话题吧。

打开 OD,接着通过 Ctrl+ Alt + Delete 快捷键打开任务管理器。

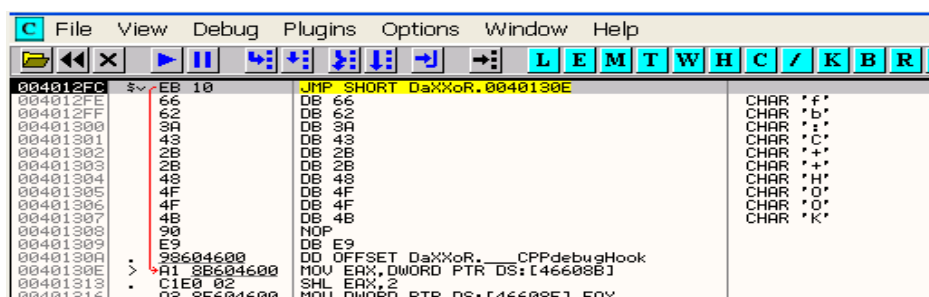
Process Name	PID	Company Name	Private Bytes	Working Set
svchost.exe	1448	SYSTEM	00	56 KB
OLLYDBG.EXE	1412	Ricardo	00	344 KB
lprw.exe	1324	SYSTEM	02	11.540 KB
avdcm.exe	1212	SYSTEM	00	1.128 KB

我们可以看到进程名称列表,我们可以通过检测进程名称是否为 OllyDbg,如果是就结束进程,哈哈。

我们来看一个 CrackMe,这个 CrackMe 运用我们前面介绍的知识暂时还解决不了,我们只是来看看它是如何检测 OllyDbg 的进程名,以及我们如何来绕过它的检测,嘿嘿。

这个 CrackMe 的名字叫做 daxxor,如果我们先运行这个 CrackMe,然后打开 OD,会发现 OD 马上就退出了。如果我们用 OD 加载该 CrackMe 然后运行起来,会发生两者一起退出了。

如何应对这种情况呢,我们先用 OD 加载这个 CrackMe。



停在了入口点处,我们先看看该程序使用了哪些 API 函数。

00472C5C	.idata	Import	OLEAUT32.#8
00472C48	.idata	Import	OLEAUT32.#84
00472C50	.idata	Import	OLEAUT32.#9
00472C38	.idata	Import	OLEAUT32.#94
00402930	.text	Export	@Unit2@Finalize
00402920	.text	Export	@Unit2@Initialize
00472908	.idata	Import	USER32.ActivateKeyboardLayout
0047290C	.idata	Import	USER32.AdjustWindowRectEx
00472910	.idata	Import	USER32.BeginPaint
00472574	.idata	Import	GDI32.BitBlt
00472914	.idata	Import	USER32.CallNextHookEx
00472918	.idata	Import	USER32.CallWindowProcA
0047291C	.idata	Import	USER32.CharLowerA
00472920	.idata	Import	USER32.CharLowerBuffA
00472924	.idata	Import	USER32.CharNextA
00472928	.idata	Import	USER32.CharUpperBuffA
0047292C	.idata	Import	USER32.CheckMenuItem
00472930	.idata	Import	USER32.ClientToScreen
00472934	.idata	Import	USER32.CloseClipboard
00472230	.idata	Import	KERNEL32.CloseHandle
00472234	.idata	Import	KERNEL32.CompareStringA
00472578	.idata	Import	GDI32.CopyEnhMetaFileA
00466098	.data	Export	__CPPdebugHook
0047257C	.idata	Import	GDI32.CreateBitmap
00472580	.idata	Import	GDI32.CreateBrushIndirect
00472584	.idata	Import	GDI32.CreateCompatibleBitmap
00472588	.idata	Import	GDI32.CreateCompatibleDC
00472590	.idata	Import	GDI32.CreateDIBitmap
0047258C	.idata	Import	GDI32.CreateDIBSection
00472238	.idata	Import	KERNEL32.CreateEventA
0047223C	.idata	Import	KERNEL32.CreateFileA
00472594	.idata	Import	GDI32.CreateFontIndirectA
00472598	.idata	Import	GDI32.CreateHalftonePalette
00472938	.idata	Import	USER32.CreateIcon
0047293C	.idata	Import	USER32.CreateMenu
0047259C	.idata	Import	GDI32.CreatePalette
004725A0	.idata	Import	GDI32.CreatePenIndirect
00472940	.idata	Import	USER32.CreatePopupMenu
004725A4	.idata	Import	GDI32.CreateSolidBrush
00472240	.idata	Import	KERNEL32.CreateThread
00472944	.idata	Import	USER32.CreateWindowExA
00472948	.idata	Import	USER32.DefFrameProcA
0047294C	.idata	Import	USER32.DefMDIChildProcA
00472950	.idata	Import	USER32.DefWindowProcA
00472244	.idata	Import	KERNEL32.DeleteCriticalSection
004725A8	.idata	Import	GDI32.DeleteDC

我们可以看到有很多 API 函数,但都不是用于检测进程名的,可能这些重要的 API 函数被隐藏起来了,没有出现该列表中。

很显然如果程序不直接导入某些 API 的话,会使用 GetProcAddress 这个 API 函数来获取这些 API 函数的地址进行间接调用。

004725E8	.idata	Import	GDI32.GetPaletteEntries
0047291C	.idata	Import	USER32.GetParent
004725EC	.idata	Import	GDI32.GetPixel
004722A8	.idata	Import	KERNEL32.GetProcAddress
004722AC	.idata	Import	KERNEL32.GetProcessHeap
00472A20	.idata	Import	USER32.GetPropA
00472A24	.idata	Import	USER32.GetScrollInfo

使用 GetProcAddress 函数加载的一些 API 函数并不会出现该 API 函数列表中,我们给 GetProcAddress 设置一个断点。

00472458	.idata	Import	USER32.GetWindowThread
00472604	.idata	Import	GDI32.GetWinMetaFileBi

Command BP a

Program entry point

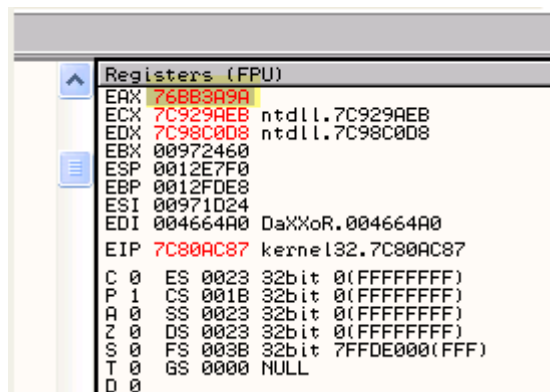
运行起来。

0012FF60	0045C92A	CALL to GetProcAddress from DaXx0R.0045C925
0012FF64	00400000	hModule = 00400000 (DaXx0R)
0012FF68	00467BA8	ProcNameOrOrdinal = "__CPPdebugHook"
0012FF6C	004607C1	RETURN to DaXx0R.004607C1
0012FF70	00000000	
0012FF74	00466034	DaXx0R.00466034
0012FF78	7FFDF000	
0012FF7C	0046C040	DaXx0R.0046C040

断在了 GetProcAddress 的入口点处,当前待获取的函数是 __CPPdebugHook,该函数检测进程名没有关系,继续运行。

0012E7F0	00401C98	CALL to GetProcAddress from DaXx0R.00401C93
0012E7F4	76BB0000	hModule = 76BB0000
0012E7F8	00466275	ProcNameOrOrdinal = "EnumProcesses"
0012E7FC	00140000	
0012E800	0012E8E4	
0012E804	00000000	

我们继续按 F9 键运行直到待获取的 API 是与检测进程名相关的为止,这里待获取的 API 函数是 EnumProcesses,这里通过选择主菜单 Debug-Execute till return 执行到返回,这个时候 EAX 寄存器中保存的就是 EnumProcesses 这个 API 函数的地址了。我们接着就可以给该地址设置断点了。



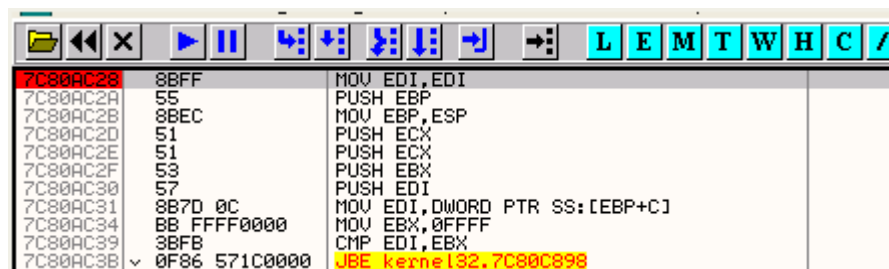
这里 EAX 就保存了 EnumProcesses 的函数地址,我的机器上是 76BB3A9A(可能与你机器上的不一样)。



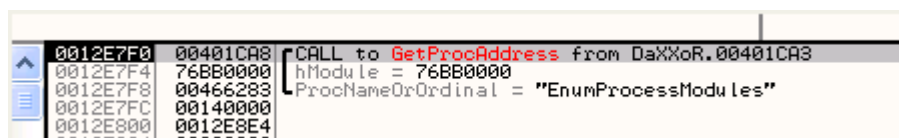
另外,OD 的 API 的函数列表中并没有列出 EnumProcesses 这个名称,所以我们直接 bp EnumProcesses 是设置不了断点的,我们可以给 EnumProcesses 函数的地址设置断点。



设置成功了。

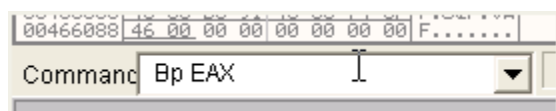


好了,我们已经给 EnumProcesses 设置了断点,继续我们刚才的步骤,看看还有什么 API 函数被加载。

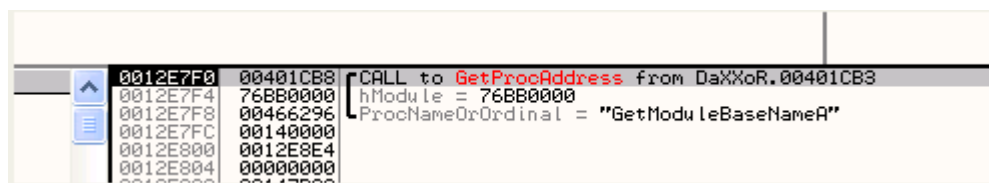


这里是获取枚举进程模块函数的地址,我们还是执行到返回,接着给 EAX 中保存的地址设置断点。

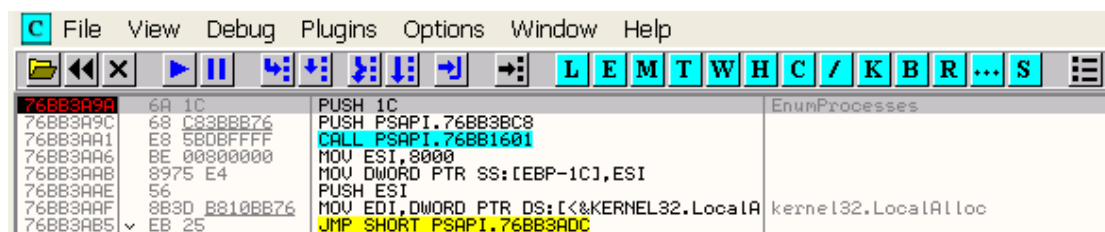
我们直接在命令栏中输入 bp EAX。



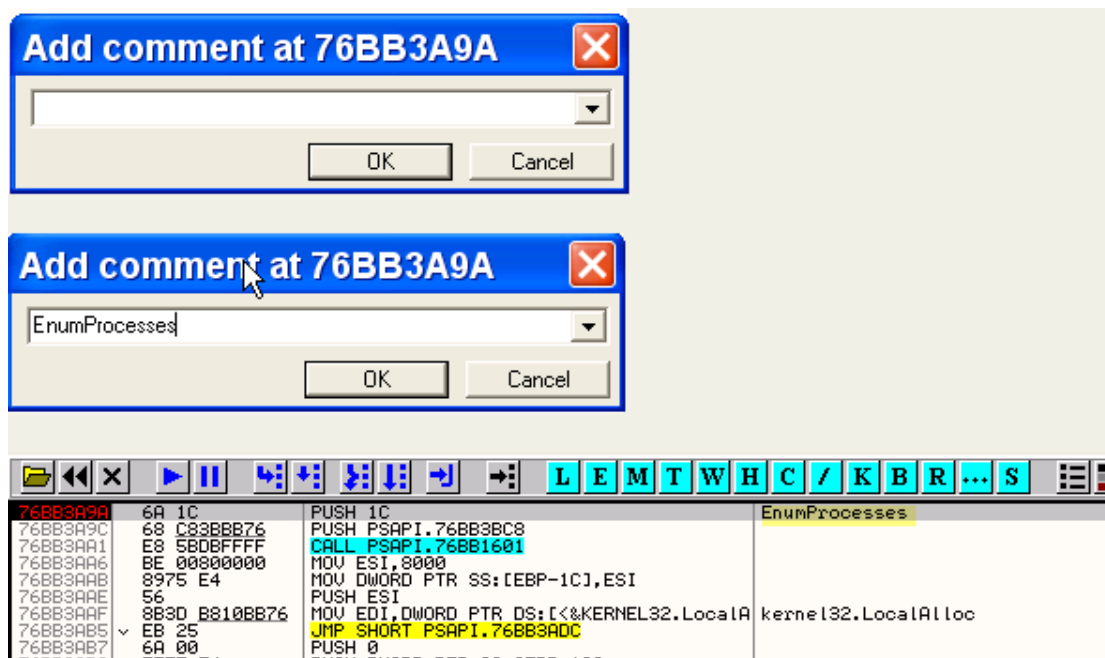
同理,我们给所有与检测进程名相关的 API 函数设置断点。



这里是另外一个可疑的 API 函数 GetModuleBaseNameA,我们跟之前一样给该函数设置断点,然后我们运行起来,断在了 EnumProcesses 函数的入口处。

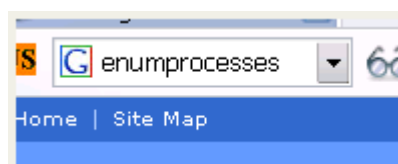


我们通过双击反汇编窗口的注释区域来给该函数添加注释,注释上该函数的名称 EnumProcesses。



我们给这些通过 GetProcAddress 加载的 API 设置断点,并且断下来了,但是 OD 并没有在 API 函数列表中找到与之对应的项,所以 OD 也没有提示该函数相关信息。

我们谷歌一下“EnumProcesses”。



找到一个微软的网站页面:

<http://msdn2.microsoft.com/en-us/library/ms682629.aspx>

http://msdn.microsoft.com/library/default.asp?url=/library/js/enumprocesses

Microsoft.com Home | Site Map

MSDN Home | Developer Centers | Library | Downloads | How to Buy | Subscribers | Worldwide

Search for:

enumproc

- Up One Level
- EmptyWorkingSet
- EnumDeviceDrivers
- EnumPageFiles
- EnumPageFilesProc
- EnumProcesses
- EnumProcessModules
- GetDeviceDriverBaseName
- GetDeviceDriverFileName
- GetMappedFileName
- GetModuleBaseName
- GetModuleFileNameEx
- GetModuleInformation
- GetPerformanceInfo
- GetProcessImageFileName
- GetProcessMemoryInfo
- GetWsChanges
- InitializeProcessForWsv
- QueryWorkingSet
- QueryWorkingSetEx

EnumProcesses

The **EnumProcesses** function retrieves the process identifier for each process object in the system.

```

BOOL EnumProcesses (
    DWORD * pProcessIds ,
    DWORD cb ,
    DWORD * pBytesReturned
);

```

Parameters

pProcessIds
[out] Pointer to an array that receives the list of process identifiers.

cb
[in] Size of the *pProcessIds* array, in bytes.

pBytesReturned
[out] Number of bytes returned in the *pProcessIds* array.

Return Values

If the function succeeds, the return value is nonzero.

上面的解释是该函数返回正在运行的每个进程的标识即 PID。好了,接下来我们看看神马是 PID,嘿嘿。

PID 是系统分配给每个正在运行的进程的标识符-每次启动的时候都会发生变化。我们来看看进程列表。

Administrador de tareas de Windows

Archivo Opciones Ver Apagar Ayuda

Aplicaciones **Procesos** Rendimiento Funciones de red Usuarios

Nombre de imagen	PID	Nombre de us...	CPU	Uso de ...
jusched.exe	1268	Ricardo	00	1.432 KB
vmware-authd.exe	1216	SYSTEM	00	2.672 KB
svchost.exe	1140	Servicio de red	00	3.008 KB
svchost.exe	1084	SYSTEM	00	3.068 KB
explorer.exe	992	Ricardo	00	9.544 KB
lsass.exe	924	SYSTEM	00	1.080 KB
services.exe	912	SYSTEM	00	3.484 KB
winlogon.exe	868	SYSTEM	00	424 KB
csrss.exe	844	SYSTEM	00	3.464 KB
smss.exe	788	SYSTEM	00	272 KB
OLLYDBG.EXE	724	Ricardo	00	4.632 KB
NETFileServerEngine.exe	576	SYSTEM	02	7.480 KB
spoolsv.exe	492	SYSTEM	00	3.792 KB

我们可以看到,这里,OD 的 PID 是 724-十进制。我们用 windows 自带的计算器将该 PID 值转化为十六进制。

Calculadora

Edición Ver Ayuda

724,

☐ Hex
 ☒ Dec
 ☐ Oct
 ☐ Bin
 ☒ Sexagesimal
 ☐ Radián
 ☐ Centesimal

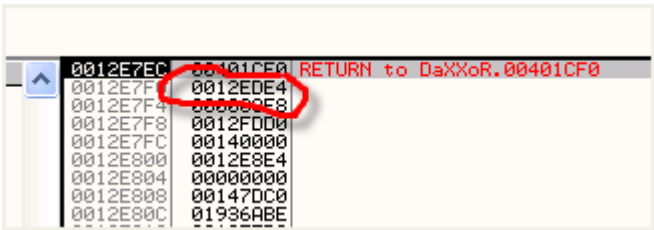
☐ Inv
 ☐ Hyp
 Retroceso
 CE
 C

Sta
 F-E
 (
)
 MC
 7
 8
 9
 /
 Mod
 And

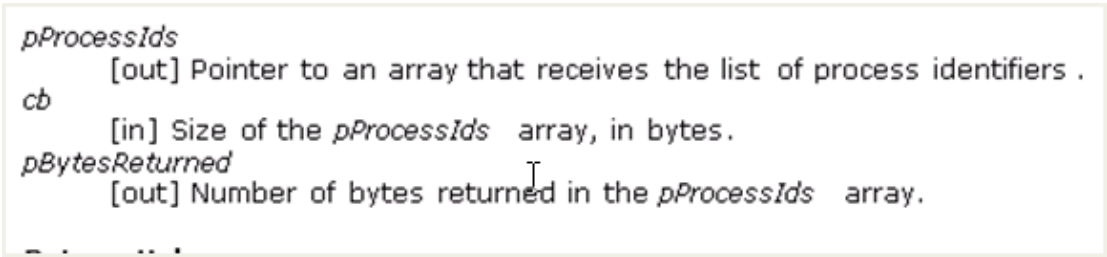
单击 Hex 按钮,就以十六进制显示了。



PID 对应的十六进制为 2D4。可以关闭 OD 然后重新打开一个 OD,会发现 PID 发生了变化。进程每次重新启动,PID 都改变了。
很不走运,API 的参数提示也没有了。



根据 MSDN 我们知道该函数有 3 个参数。

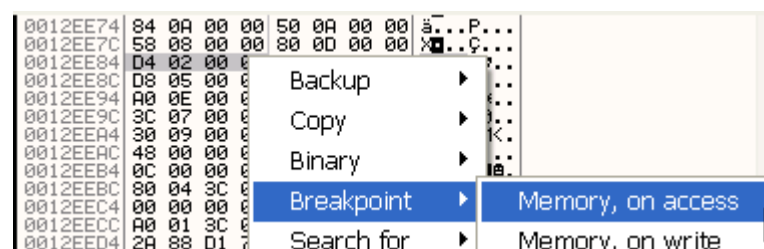


在我们的机器上 12EDE4 该地址指向了保存所有进程 PID 的数组,我们执行到返回看看该 API 函数返回了什么,我们通过数据窗口查看一下。

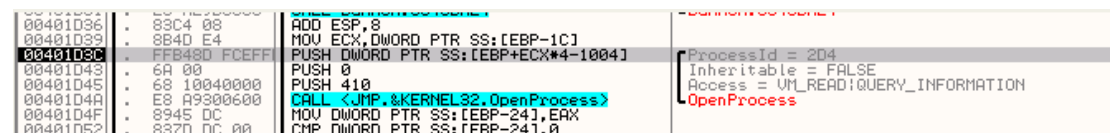
Address	Hex dump	ASCII
0012EDE4	00 00 00 00 04 00 00 00
0012EDEC	14 03 00 00 4C 03 00 00	..L..
0012EDF4	64 03 00 00 90 03 00 00	d...E..
0012EDFC	9C 03 00 00 3C 04 00 00	...<..
0012EE04	74 04 00 00 60 06 00 00	t...^..
0012EE0C	94 06 00 00 E0 06 00 00	...G...
0012EE14	EC 01 00 00 E0 03 00 00	...G...
0012EE1C	08 05 00 00 10 05 00 00	...P...
0012EE24	1C 05 00 00 28 05 00 00	...L...
0012EE2C	38 05 00 00 40 05 00 00	...S...
0012EE34	48 05 00 00 54 05 00 00	...H...
0012EE3C	5C 05 00 00 6C 05 00 00	...I...
0012EE44	B0 01 00 00 04 07 00 00	...E...
0012EE4C	EC 07 00 00 C8 00 00 00	...E...
0012EE54	40 02 00 00 00 05 00 00	...S...
0012EE5C	88 06 00 00 C0 04 00 00	...L...
0012EE64	F8 05 00 00 4C 06 00 00	...L...
0012EE6C	30 08 00 00 08 0C 00 00	...i...
0012EE74	84 0A 00 00 50 0A 00 00	...P...
0012EE7C	58 08 00 00 80 0D 00 00	...C...
0012EE84	04 02 00 00 7C 0B 00 00	...i...
0012EE8C	08 05 00 00 FC 0D 00 00	...F...
0012EE94	A0 0E 00 00 E0 0F 00 00	...O...
0012EE9C	3C 07 00 00 A8 01 00 00	...<...
0012EEA4	30 09 00 00 00 4D 3C 00	...M...
0012EEAC	48 00 00 00 01 00 00 00	...H...
0012EEB4	0C 00 00 00 EB 08 02 00	...U...
0012EEBC	80 04 3C 00 00 4D 3C 00	...M...
0012EEC4	00 00 00 00 67 04 D4 77	...g...w
0012EECC	A0 01 3C 00 FF FF FF FF	...\$...
0012EED4	2A 88 D1 77 06 00 00 00	...\$...
0012EEDC	00 00 00 00 24 51 46 00	...\$...
0012EEF4	3A 0A 0A 0A 7F 0A 0A 0A	...A...

在 PID 列表中我们找到了 OD 的 PID 的值,嘿嘿。

我们给该 PID 设置内存访问断点,看看哪里使用了它。



运行起来。



可以看到这里将要调用 OpenProcess,如果调用成功,就会获取 OD 进程的句柄。

PID 跟句柄有什么区别呢?很简单。PID 是用于区别不同进程的标示符,一个进程被创建后这个进程的 PID 就是不变的,除非进程重新启动。当前,OD 的进程 PID 是 2D4。而句柄实际上是一个指针,它指向一个包含具体数据结构的内存,可能当做索引,所以句柄是你每次访问该进程的时候获取的,使用完毕后要释放,然后通过该句柄可以对该进程进行相应操作。

下面是 OpenProcess 其他参数参数解释,我们并不感兴趣。

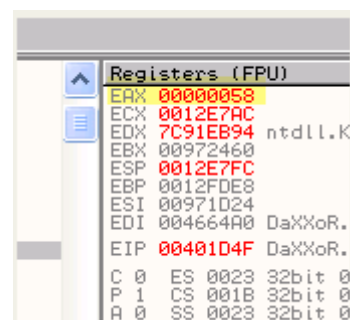
Return Values

If the function succeeds, the return value is an open handle of the specified process.

If the function fails, the return value is NULL. To get extended error information, call [GetLastError](#).

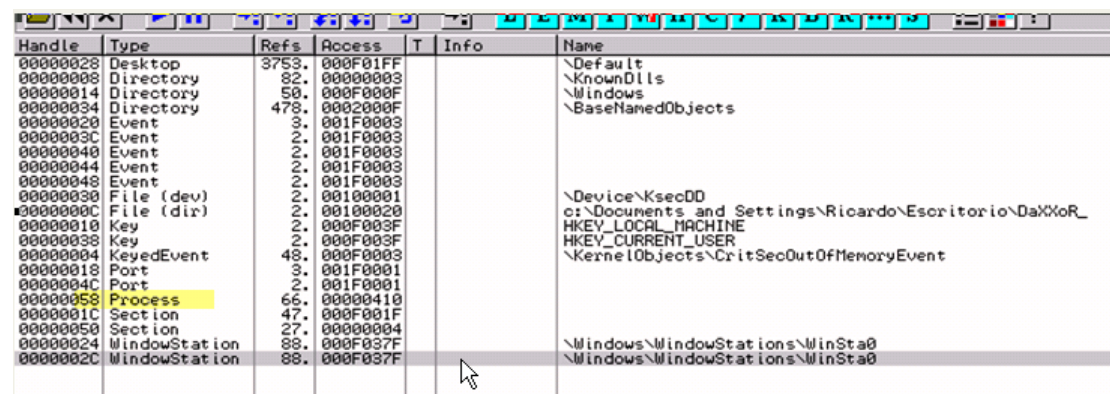
返回值为指定进程的句柄。

我们 F8 单步直到调用该函数。



EAX 中返回了 OllyDbg 进程的句柄,我这里是 58。

我们也可以通过单击工具栏上面的 H 按钮查看 OD 的句柄列表。



我们可以看到 58 对应的类型是 Process,它标识的是 OllyDbg 的进程句柄。

如果另一个进程调用 EnumProcesses 获取 OD 的进程 PID 的话,仍然会是 2D4,但是获取进程句柄的话,会是另外一个值,因为句柄在系统中是独一无二的。

这里,OllyDbg 进程就比较危险了,有了 OD 的进程句柄,该程序就可以做很多事情了(比如结束 OD 进程),以上只是检测 OllyDbg 步骤的一部分,现在还需要来验证一下进程名称是否为 OllyDbg,很明显该程序会对进程列表的所有进程做以上操作,我们跳过这些步骤直接给对应的 PID 设置内存访问断点。

我们继续 F8 键单步。

00401D5E	8B55 00	LEA EAX,DWORD PTR SS:[EBP+40]	
00401D61	52	PUSH EDX	
00401D62	FF75 DC	PUSH DWORD PTR SS:[EBP-24]	
00401D65	FF55 B4	CALL DWORD PTR SS:[EBP-4C]	PSAPI.EnumProcessModules
00401D68	85C0	TEST EAX,EAX	
00401D6A	74 18	JE SHORT DaXx0R.00401D84	
00401D6C	68 04010000	PUSH 104	
00401D71	8D8D F8EEFF	LEA ECX,DWORD PTR SS:[EBP-1108]	

我们看到了 EnumProcessModules 这个函数,我们看看 MSDN 上面关于这个函数的说明。

Search for

MSDN Library

Search

Advanced Search

enumproc

Up One Level

EmptyWorkingSet

EnumDeviceDrivers

EnumPageFiles

EnumPageFilesProc

EnumProcesses

EnumProcessModules

GetDeviceDriverBaseName

GetDeviceDriverFileName

GetMappedFileName

GetModuleBaseName

GetModuleFileNameEx

GetModuleInformation

GetPerformanceInfo

GetProcessImageFileName

GetProcessMemoryInfo

GetWsChanges

InitializeProcessForWsv

QueryWorkingSet

QueryWorkingSetEx

Welcome to the MSDN Library

Platform SDK: Performance Monitoring

EnumProcessModules

The EnumProcessModules function retrieves a handle for each module in the specified process.

BOOL EnumProcessModules(
HANDLE hProcess,
HMODULE* lphModule,
DWORD cb,
LPDWORD lpcbNeeded
);

Parameters

hProcess

[in] Handle to the process.

lphModule

[out] Pointer to the array that receives the list of module handles.

cb

[in] Size of the lphModule array, in bytes.

lpcbNeeded

[out] Number of bytes required to store all module handles in the lphModule array.

Return Values

If the function succeeds, the return value is nonzero.

该函数是枚举指定进程的模块,我们 F7 键跟进到该函数的入口处。

76BB1F1C	68 88000000	PUSH 88	EnumProcessModules
76BB1F21	68 5820BB76	PUSH PSAPI.76BB2058	
76BB1F26	E8 06F6FFFF	CALL PSAPI.76BB1601	
76BB1F2B	33DB	XOR EBX,EBX	
76BB1F2D	53	PUSH EBX	
76BB1F2E	6A 18	PUSH 18	
76BB1F30	8D45 B8	LEA EAX,DWORD PTR SS:[EBP-48]	
76BB1F33	50	PUSH EAX	
76BB1F34	E3	PUSH EBX	

我们在堆栈窗口中来看看参数情况:

0012E7E8	00401D68	RETURN to DaXx0R.00401D68
0012E7EC	00000058	
0012E7F0	0012FDA8	
0012E7F4	00000004	
0012E7F8	0012FDD0	
0012E7FC	00140000	
0012E800	0012E8E4	
0012E804	00000000	

58 是 OD 的进程句柄作为第一个参数。

这里,你需要清楚一点:当我们请求获取进程中模块句柄的时候,系统会返回该模块在进程内存中基地址(模块起始地址),这里,系统给我返回的是 400000,该基地址对应的 OllyDbg 进程中的。

Address	Hex dump	ASCII
0012FDA8	00 00 40 00 01 00 00 00	..@.0...
0012FDB0	0B 00 00 00 0B 00 00 00	3...3...
0012FDB8	00 00 BB 76 F0 FD 12 00	..UV-2...
0012FDC0	67 04 D4 77 58 00 00 00	gEWX...
0012FDC8	00 00 00 00 28 00 00 00(...
0012FDD0	70 00 00 00 E6 07 0A 00	p...p...
0012FDD8	31 00 00 00 F8 03 00 00	1...b...

继续跟,我们可以看到另一个 API 函数。

00401071	808D F8EEFF	LEA ECX,DWORD PTR SS:[EBP-1108]	
00401077	51	PUSH ECX	
00401079	FF75 C0	PUSH DWORD PTR SS:[EBP-40]	
0040107B	FF75 DC	PUSH DWORD PTR SS:[EBP-24]	
0040107E	FF55 00	CALL DWORD PTR SS:[EBP-50]	PSAPI.GetModuleBaseNameA
00401081	8945 C4	MOV DWORD PTR SS:[EBP-3C],EAX	
00401084	FF75 DC	PUSH DWORD PTR SS:[EBP-24]	hObject

GetModuleBaseNameA

GetModuleBaseName

The **GetModuleBaseName** function retrieves the base name of the specified module.

```
DWORD GetModuleBaseName (  
    HANDLE hProcess,  
    HMODULE hModule,  
    LPSTR lpBaseName,  
    DWORD nSize  
);
```

Parameters

hProcess

[in] Handle to the process that contains the module. If this parameter is NULL, **GetModuleBaseName** uses the current process. The handle must have the **PROCESS_QUERY_INFORMATION** and **PROCESS_VM_READ** access rights. For more information, see [Process Security and Access Rights](#).

hModule

[in] Handle to the module. If this parameter is NULL, this function returns the name of the file used to create the calling process.

lpBaseName

[out] Pointer to the buffer that receives the base name of the module. If the base name is longer than maximum number of characters specified by the **nSize** parameter, the base name is truncated.

nSize

[in] Size of the **lpBaseName** buffer, in characters.

Return Values

If the function succeeds, the return value specifies the length of the string copied to the buffer, in characters.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

该函数是获取指定进程模块的名称,lpBaseName 这个参数是用来保存模块名称的缓冲区首地址,我们执行到返回。

76BB215A	8BFF	MOV EDI,EDI	GetModuleBaseNameA
76BB215C	55	PUSH EBP	
76BB215D	8BEC	MOV EBP,ESP	
76BB215F	53	PUSH EBX	
76BB2160	56	PUSH ESI	
76BB2161	8B75 14	MOV ESI,DWORD PTR SS:[EBP+14]	
76BB2164	8D0436	LEA EAX,DWORD PTR DS:[ESI+ESI]	
76BB2167	50	PUSH EAX	
76BB2168	33DB	XOR EBX,EBX	
76BB216A	53	PUSH EBX	
76BB216B	FF15 B810BB76	CALL DWORD PTR DS:[<&KERNEL32.LocalAlloc	kernel32.LocalAlloc
76BB2171	3BC3	CMP EAX,EBX	
76BB2173	8945 14	MOV DWORD PTR SS:[EBP+14],EAX	

堆栈窗口中参数情况如下:

0012E7E8	00401081	RETURN to DaXXoR.00401081
0012E7EC	00000053	
0012E7F0	00400000	ASCII "MZP"
0012E7F4	0012ECE0	ASCII "Unknown"
0012E7F8	00000104	
0012E7FC	00140000	
0012E800	0012E8E4	
0012E804	00000000	

58 是 OllyDbg 的进程句柄,400000 是 OD 主模块的基地址,用于保存模块名称的缓冲区首地址为 12ECE0,我们在数据窗口中定位到这个地址。

Address	Hex dump	ASCII
0012ECE0	4F 4C 4C 59 44 42 47 2E	OLLYDBG.
0012ECE8	45 58 45 00 48 ED 12 00	EXE.HY+
0012ECF0	B7 2C 92 7C 00 3E 00 00	A,El>..
0012ECF8	28 65 47 00 28 ED 12 00	(eG.(Y+
0012ED00	00 00 00 00 8E A0 80 7C	...âäç!
0012ED08	00 00 40 00 28 65 47 00	..@.(eG.
0012ED10	28 ED 12 00 A5 A0 80 7C	(Y+.âäç!

呵呵,很显然,我们执行到返回以后获取到了进程的模块名称,同理,该程序会依次判断每个进程的名称是否为"OLLYDBG.exe"。

00401D7B	. FF75 DC	PUSH DWORD PTR SS:[EBP-24]	
00401D7E	. FF55 B0	CALL DWORD PTR SS:[EBP-50]	
00401D81	. 8945 C4	MOV DWORD PTR SS:[EBP-3C],EAX	
00401D84	> FF75 DC	PUSH DWORD PTR SS:[EBP-24]	
00401D87	. E8 E02E0600	CALL <JMP.&KERNEL32.CloseHandle>	hObject CloseHandle
00401D8C	. 8D85 F4EDFFF	LEA EAX,DWORD PTR SS:[EBP-120C]	

好了,现在看到了 CloseHandle 这个 API 函数,该函数用来关闭指定的句柄,即 58 将从句柄列表中消失。

Handle	Type	Refs	Access	T	Info	Name
00000028	Desktop	3944	000F01FF			\Default
00000008	Directory	83	00000003			\KnownDlls
00000014	Directory	51	000F000F			\Windows
00000034	Directory	484	0002000F			\BaseNamedObjects
00000020	Event	3	001F0003			
0000003C	Event	2	001F0003			
00000040	Event	2	001F0003			
00000044	Event	2	001F0003			
00000048	Event	2	001F0003			
00000030	File (dev)	2	00100001			\Device\KsecDD
0000000C	File (dir)	2	00100020			c:\Documents and Settings\Ricardo\Escritorio\DaXxoR_
00000018	Key	2	000F003F			HKEY_LOCAL_MACHINE
00000038	Key	2	000F003F			HKEY_CURRENT_USER
00000004	KeyedEvent	49	000F0003			\KernelObjects\CritSecOutOfMemoryEvent
00000018	Port	3	001F0001			
0000004C	Port	2	001F0001			
0000001C	Section	48	000F001F			
00000050	Section	28	00000004			
00000024	WindowStation	91	000F037F			\Windows\WindowStations\WinSta0
0000002C	WindowStation	91	000F037F			\Windows\WindowStations\WinSta0

好了,现在进程句柄被关闭了,我们不能再依赖它了。

00401D8C	. 8D85 F4EDFFF	LEA EAX,DWORD PTR SS:[EBP-120C]	
00401D92	. 50	PUSH EAX	
00401D93	. 8D95 F8EEFFF	LEA EDX,DWORD PTR SS:[EBP-1108]	
00401D99	. 52	PUSH EDX	
00401D9A	. E8 21BD0500	CALL DaXxoR.0045DAC0	Arg1 DaXxoR.0045DAC0
00401D9F	. 59	POP ECX	
00401DA0	. 50	PUSH EAX	
00401DA1	. E8 F29C0500	CALL DaXxoR.0045BA98	
00401DA6	. 83C4 08	ADD ESP,8	
00401DA9	. 85C0	TEST EAX,EAX	
00401DAB	. 75 74	JNZ SHORT DaXxoR.00401E21	

这里有一个 CALL 指令,我们按 F7 键跟进。

0045DACF	. 8A03	MOV AL,BYTE PTR DS:[EBX]	
0045DAD1	. 50	PUSH EAX	
0045DAD2	. E8 0D000000	CALL DaXxoR.0045DAE4	Arg1 DaXxoR.0045DAE4
0045DAD7	. 59	POP ECX	
0045DAD8	. 8A03	MOV BYTE PTR DS:[EBX],AL	
0045DADA	. 84C0	TEST AL,AL	
0045DADC	. 75 EE	JNZ SHORT DaXxoR.0045DACC	
0045DB17	. C3	RETN	

Stack DS:[0012ECE0]=4F ('O')
AL=00

这里,我们可以看到将要压入堆栈的是"OLLYDBG"字符串的第一个字母(4F),然后调用一个 CALL 指令,我们继续跟进。

我们可以看到这个 CALL 里面没有什么特别的,所以我们跟出来,然后跟到下一个 CALL 指令处。

00401D9F	. 59	POP ECX	
00401DA0	. 50	PUSH EAX	
00401DA1	. E8 F29C0500	CALL DaXxoR.0045BA98	
00401DA6	. 83C4 08	ADD ESP,8	
00401DA9	. 85C0	TEST EAX,EAX	
00401DAB	. 75 74	JNZ SHORT DaXxoR.00401E21	
00401DAD	. C745 E0 0100	MOV DWORD PTR SS:[EBP-20],1	
00401DB4	. 8B4D E4	MOV ECX,DWORD PTR SS:[EBP-1C]	

跟进。

0045BA98	8B4C24 04	MOV ECX,DWORD PTR SS:[ESP+4]
0045BA9C	8B5424 08	MOV EDX,DWORD PTR SS:[ESP+8]
0045BAA0	53	PUSH EBX
0045BAA1	33C0	XOR EAX,EAX
0045BAAB	33DB	XOR EBX,EBX
0045BAAB	8A01	MOV AL,BYTE PTR DS:[ECX]
0045BAAB	8A1A	MOV BL,BYTE PTR DS:[EDX]
0045BAAB	2BC3	SUB EAX,EBX
0045BAAB	75 34	JNZ SHORT DaXXoR.0045BAE1
0045BAAD	84DB	TEST BL,BL
0045BAAF	74 30	JE SHORT DaXXoR.0045BAE1
0045BAB1	8A41 01	MOV AL,BYTE PTR DS:[ECX+1]
0045BAB4	8A5A 01	MOV BL,BYTE PTR DS:[EDX+1]
0045BAB7	2BC3	SUB EAX,EBX
0045BAB9	75 26	JNZ SHORT DaXXoR.0045BAE1
0045BAB8	84DB	TEST BL,BL
0045BABD	74 22	JE SHORT DaXXoR.0045BAE1
0045BAF7	8A41 02	MOV AL,BYTE PTR DS:[ECX+2]
0045BAC2	8A5A 02	MOV BL,BYTE PTR DS:[EDX+2]
0045BAC5	2BC3	SUB EAX,EBX
0045BAC7	75 18	JNZ SHORT DaXXoR.0045BAE1
0045BAC9	84DB	TEST BL,BL
0045BACB	74 14	JE SHORT DaXXoR.0045BAE1
0045BACD	8A41 03	MOV AL,BYTE PTR DS:[ECX+3]
0045BAD0	8A5A 03	MOV BL,BYTE PTR DS:[EDX+3]
0045BAD3	2BC3	SUB EAX,EBX
0045BAD5	75 0A	JNZ SHORT DaXXoR.0045BAE1
0045BAD7	83C1 04	ADD ECX,4
0045BADA	83C2 04	ADD EDX,4
0045BADD	84DB	TEST BL,BL
0045BADF	75 C4	JNZ SHORT DaXXoR.0045BAAB
0045BAE1	5B	POP EBX
0045BAE2	C3	RETN

Registers (FPU)	
EAX	00000000
ECX	0012ECE0 ASCII "OLLYDBG.EXE"
EDX	0012EBDC ASCII "OLLYDBG.EXE"
EBX	00000000
ESP	0012E7EC
EBP	0012FDE8
ESI	00971D24
EDI	004664A0 DaXXoR.004664A0
EIP	0045BAAB DaXXoR.0045BAAB
C 0	ES 0023 32bit 0(FFFFFFFF)

嘿嘿,这里在比较进程名称是否为"OLLYDBG.EXE",如果是,就会...嘿嘿,我们现在执行到返回,看看会发生什么。

50	PUSH EAX	
E8 F29C0500	CALL DaXXoR.0045BA98	
83C4 08	ADD ESP,8	
85C0	TEST EAX,EAX	
75 74	JNZ SHORT DaXXoR.00401E21	
C745 E0 0100	MOV DWORD PTR SS:[EBP-20],1	
8B4D E4	MOV ECX,DWORD PTR SS:[EBP-1C]	
FFB48D FCEFF	PUSH DWORD PTR SS:[EBP+ECX*4-1004]	ProcessId
6A 00	PUSH 0	Inheritable = FALSE
6A 01	PUSH 1	Access = TERMINATE
E8 31300600	CALL <JMP.&KERNEL32.OpenProcess>	OpenProcess
8945 DC	MOV DWORD PTR SS:[EBP-24],EAX	
837D DC 00	CMP DWORD PTR SS:[EBP-24],0	
74 3F	JE SHORT DaXXoR.00401E0F	ExitCode = 0
6A 00	PUSH 0	hProcess
FF75 DC	PUSH DWORD PTR SS:[EBP-24]	TerminateProcess
E8 78300600	CALL <JMP.&KERNEL32.TerminateProcess>	
85C0	TEST EAX,EAX	
74 17	JE SHORT DaXXoR.00401DF5	
FF75 DC	PUSH DWORD PTR SS:[EBP-24]	hObject
E8 862E0600	CALL <JMP.&KERNEL32.CloseHandle>	CloseHandle
FF75 D0	PUSH DWORD PTR SS:[EBP-30]	hLibModule
E8 CC2E0600	CALL <JMP.&KERNEL32.FreeLibrary>	FreeLibrary
33C0	XOR EAX,EAX	
E9 AF020000	JMP DaXXoR.004020A4	
FF75 DC	PUSH DWORD PTR SS:[EBP-24]	hObject
E8 6F2E0600	CALL <JMP.&KERNEL32.CloseHandle>	CloseHandle
FF75 D0	PUSH DWORD PTR SS:[EBP-30]	hLibModule
E8 B52E0600	CALL <JMP.&KERNEL32.FreeLibrary>	FreeLibrary
B8 5A020000	MOV EAX,25A	
E9 95020000	JMP DaXXoR.004020A4	
FF75 D0	PUSH DWORD PTR SS:[EBP-30]	hLibModule
E8 A32E0600	CALL <JMP.&KERNEL32.FreeLibrary>	FreeLibrary
B8 5C020000	MOV EAX,25C	
E9 83020000	JMP DaXXoR.004020A4	
FF45 E4	INC DWORD PTR SS:[EBP-1C]	

Taken
aXXoR.00401E21

如果进程名不为"OLLYDBG.EXE",EAX 就不等于 0,JNZ 条件跳转将会发生,如果 EAX 为 0,条件跳转不会发生,将执行后面结束 OD 进程等一系列操作。

Registers (FPU)	
EAX	00000000
ECX	0012ECEC
EDX	0012EBE8 AS
EBX	00972460
ESP	0012E7FC
EBP	0012FDE8
ESI	00971D24
EDI	004664A0 Da
EIP	00401DAB Da
C 0	ES 0023 32I
P 1	CS 001B 32I

跳转不会发生,将会再次调用 `OpenProcess` 打开 OD 进程并且获得其句柄,然后调用 `TerminateProcess` 结束掉该 OD 进程。

00401DB4	8B4D E4	MOV ECX,DWORD PTR SS:[EBP-1C]	ProcessId Inheritable = FALSE Access = TERMINATE OpenProcess
00401DB7	FFB48D FCEFF	PUSH DWORD PTR SS:[EBP+ECX*4-1004]	
00401DBE	6A 00	PUSH 0	
00401DC0	6A 01	PUSH 1	
00401DC2	E8 31300600	CALL <JMP.&KERNEL32.OpenProcess>	
00401DC7	8945 DC	MOV DWORD PTR SS:[EBP-24],EAX	
00401DCA	837D DC 00	CMP DWORD PTR SS:[EBP-24],0	

0012E7F0	00000001	Access = TERMINATE
0012E7F4	00000000	Inheritable = FALSE
0012E7F8	000002D4	ProcessId = 2D4
0012E7FC	00140000	

我们按 F8 键单步。

Registers (FPU)	
EAX	00000058
ECX	0012E7AC
EDX	7C91EB94 ntdll.K
EBX	00972460
ESP	0012E7FC
EBP	0012FDE8
ESI	00971D24
EDI	004664A0 DaXXoR.
EIP	00401DC7 DaXXoR.
C 0	ES 0023 32bit 0
P 1	CS 001B 32bit 0

返回的是 58,我们继续。

00401DCA	837D DC 00	CMP DWORD PTR SS:[EBP-24],0	ExitCode = 0 hProcess TerminateProcess
00401DCE	74 3F	JE SHORT DaXXoR.00401E0F	
00401DD0	6A 00	PUSH 0	
00401DD2	FF75 DC	PUSH DWORD PTR SS:[EBP-24]	
00401DD5	E8 78300600	CALL <JMP.&KERNEL32.TerminateProcess>	
00401DDA	85C0	TEST EAX,EAX	

正如你所看到的,调用 `TerminateProcess` 这个 API 函数,指定进程的句柄,结束调用 OllyDbg 进程。

0012E7F4	00000058	hProcess = 00000058 (window)
0012E7F8	00000000	ExitCode = 0
0012E7FC	00140000	
0012E800	0012E8E4	
0012F8A4	AAAAAAAA	

我们按 F8 键,OD 被关闭了。这就是我们研究的如何检测 OllyDbg 进程名的原理。

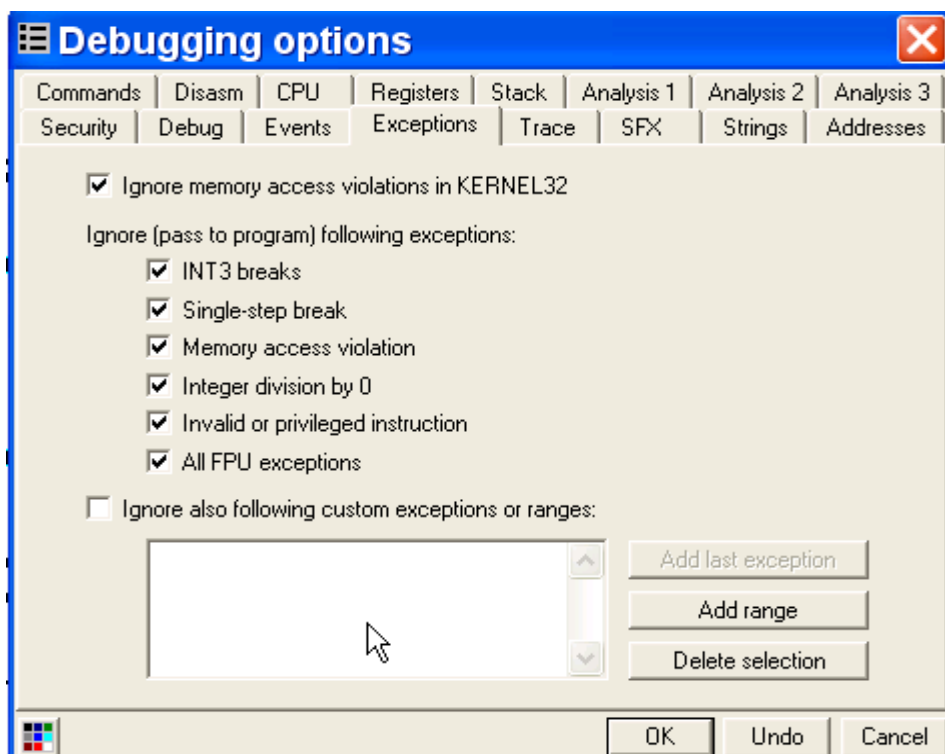
好吧,我们重新启动 OD,然后在 `OpenProcess` 入口处设置断点。

Address	Disassembly	Comment
7C81E079	8BFF	MOV EDI,EDI
7C81E07B	55	PUSH EBP
7C81E07C	8BEC	MOV EBP,ESP
7C81E07E	83EC 20	SUB ESP,20
7C81E081	8B45 10	MOV EAX,DWORD PTR SS:[EBP+10]
7C81E084	8945 F8	MOV DWORD PTR SS:[EBP-8],EAX
7C81E087	8B45 0C	MOV EAX,DWORD PTR SS:[EBP+C]
7C81E08A	56	PUSH ESI
7C81E08B	33F6	XOR ESI,ESI
7C81E08D	F7D8	NEG EAX
7C81E08F	1BC0	SBB EAX,EAX
7C81E091	83E0 02	AND EAX,2
7C81E094	8945 EC	MOV DWORD PTR SS:[EBP-14],EAX
7C81E097	8D45 F8	LEA EAX,DWORD PTR SS:[EBP-8]
7C81E09A	50	PUSH EAX
7C81E09B	8D45 E0	LEA EAX,DWORD PTR SS:[EBP-20]
7C81E09E	50	PUSH EAX
7C81E09F	FF75 08	PUSH DWORD PTR SS:[EBP+8]
7C81E0A2	8D45 10	LEA EAX,DWORD PTR SS:[EBP+10]
7C81E0A5	50	PUSH EAX
7C81E0A6	8975 FC	MOV DWORD PTR SS:[EBP-4],ESI
7C81E0A9	C745 E0 180000	MOV DWORD PTR SS:[EBP-20],18
7C81E0B0	8975 E4	MOV DWORD PTR SS:[EBP-1C],ESI
7C81E0B3	8975 E8	MOV DWORD PTR SS:[EBP-18],ESI
7C81E0B6	8975 F0	MOV DWORD PTR SS:[EBP-10],ESI
7C81E0B9	8975 F4	MOV DWORD PTR SS:[EBP-C],ESI
7C81E0BC	FF15 0C11807C	CALL DWORD PTR DS:[<<ntdll.NtOpenProcess
7C81E0C2	3BC6	CMP EAX,ESI
7C81E0C4	5E	POP ESI
7C81E0C5	0F8C FBAA0100	JL .kernel32.7C838BC6
7C81E0C8	8B45 10	MOV EAX,DWORD PTR SS:[EBP+10]
7C81E0CE	C9	LEAVE
7C81E0CF	C2 0C00	RETN 0C
7C81E0D2	50	PUSH EAX

断了下来,我们可以修改该 API 函数,让它总是返回 0,程序就会认为没有其他程序正在运行,获取不到任何进程句柄,要做到这一点,我们修改最后一行。

Address	Disassembly	Comment
7C81E0BC	FF15 0C11807C	CALL DWORD PTR DS:[<<ntdll.NtOpenProcess
7C81E0C2	3BC6	CMP EAX,ESI
7C81E0C4	5E	POP ESI
7C81E0C5	90	NOP
7C81E0C6	90	NOP
7C81E0C7	90	NOP
7C81E0C8	90	NOP
7C81E0C9	90	NOP
7C81E0CA	90	NOP
7C81E0CB	33C0	XOR EAX,EAX
7C81E0CD	90	NOP
7C81E0CE	C9	LEAVE
7C81E0CF	C2 0C00	RETN 0C
7C81E0D2	50	PUSH EAX

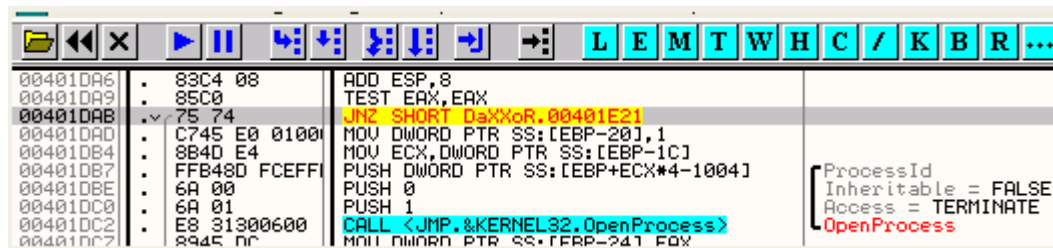
这样,这个 API 函数就总是返回 0 了。现在我们删除之前设置的所有断点。接着设置一下主菜单项中的调试选项中的异常,如下:



这里,我们勾选上忽略所以异常,因为这些异常中(有的必须按 Shift + F9 键,才能忽略异常继续执行,这里我们直接忽略掉)运行起来。

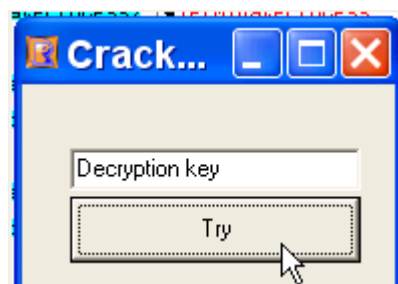


正常运行,其实你也可以这么做:



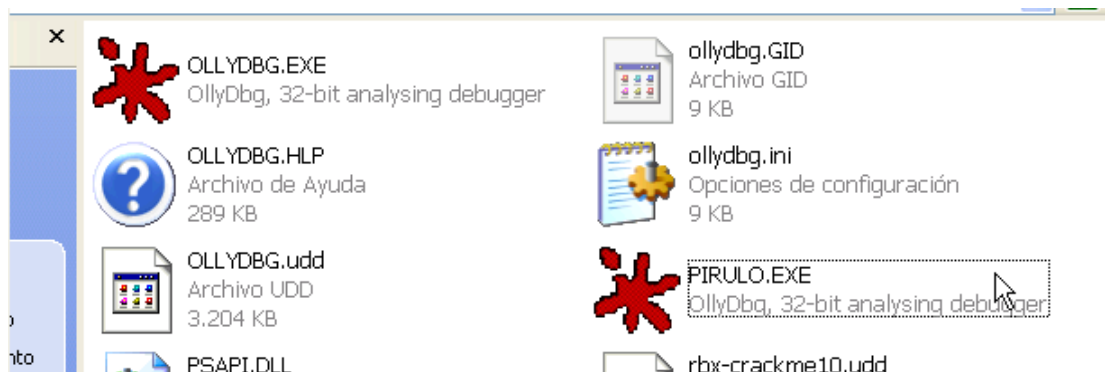
将 JNZ 修改为 JMP,让其直接跳过下面的关闭 OllyDbg 进程的代码。

运行起来,主窗口显示出来了,我们单击 Try 按钮。

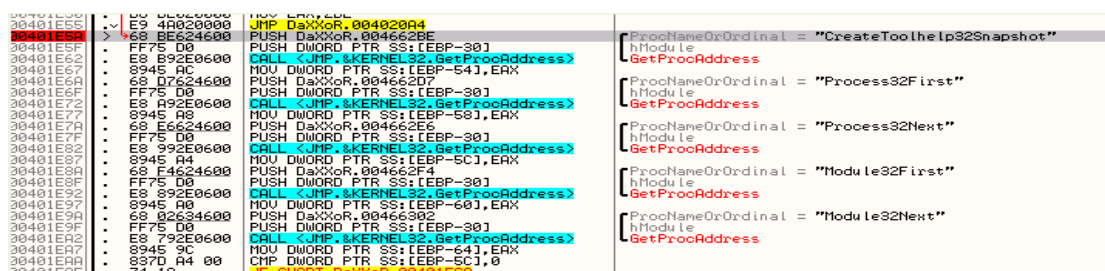


会弹出一个错误信息框。显示您已成功绕过该反调试。

以上的方法并不一定能绕过这个反调试,其实我们可以通过将 OLLYDBG.EXE 创建一个副本,将其重命名为 PIRULO.EXE,然后运行该程序,被其调试的程序将无法找到一个进程名为 OLLYDBG 的进程了。



有一点很重要就是在 OD 的文件夹下面留一个原始的 OD,避免 OD 的插件出现问题。



以上只是该程序检测 OllyDbg 的一部分,如果我们输入正确的序列号,该程序还会调用其他的一些 API 函数来检测是否被 OD 调用,我们下一章再来详细讨论。