

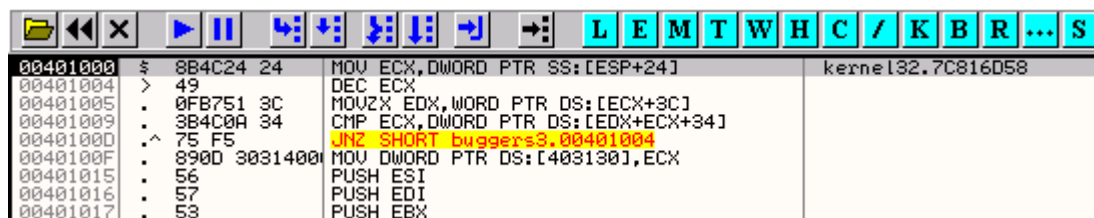
## 第二十一章-OllyDbg 反调试之检测 OD 进程名,窗口类名,窗口标题名

本章我们继续讨论反调试,将我修改过的一个 CrackMe 作为本章的实验对象。

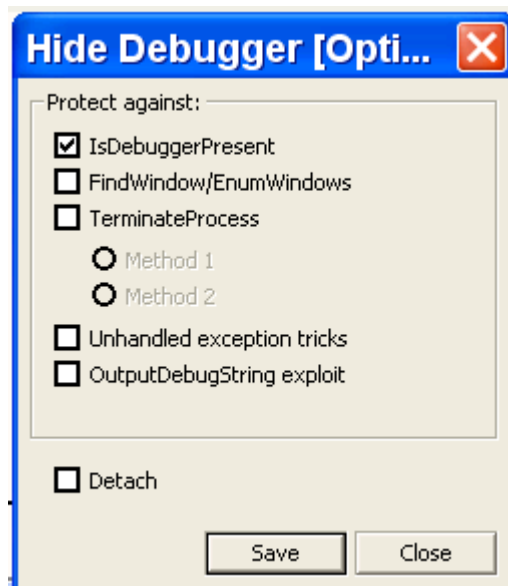
该 CrackMe 的名字叫做 buggers,其中做的一些修改是为了介绍检测 OllyDbg 进程名的其他一些 API 函数,同时该 CrackMe 也涵盖了检测 OllyDbg 窗口标题名以及窗口类名等知识点。

我们打开原始的 OllyDbg 程序,不使用重命名的,因为本章我们将对上一章的检测 OD 的方法进行延伸,因此让 OD 的文件名是 OllyDbg.exe,保证该 CrackMe 可以检测出来 OD。

我们用 OD 加载该 CrackMe,接着将 HideDebugger1.23 版插件的 IsDebuggerPresent 选项勾选上。



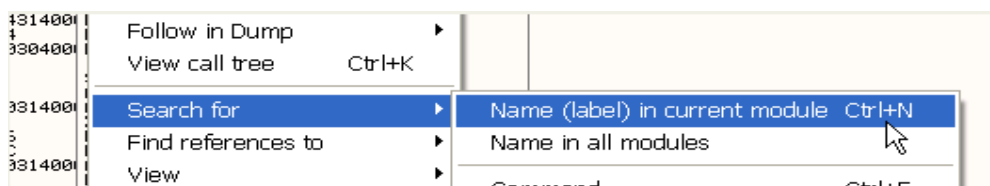
HideDebugger 插件的配置如下:



这里只是为了防止该程序调用 IsDebuggerPresent 对 OD 进行检测。我们打开原始的 OllyDbg.exe,然后打开任务管理器,确保 OD 的进程名为“OLLYDBG.EXE”



好了,我们回到 buggers3,看看该程序使用了哪些 API 函数。



Names in buggers3			
Address	Section	Type	Name
00402000	.rdata	Import	kernel32.ExitProcess
00401000	.text	Export	<ModuleEntryPoint>

我的天啦!API 列表中居然只是唯一的一个函数 ExitProcess,其他 API 函数应该都是通过 GetProcAddress 加载的,但是 GetProcAddress 也不在该列表中。

00403080	73 73 00 6C	73 74 72 63	ss.lstrc
00403088	6D 70 41 00	00 2C 31 40	mpA...1@
00403090	00 46 69 6E	64 57 69 6E	.FindWin

Command: Bp GetProcAddress

Program entry point

我们试试在命令栏中输入 bp GetProcAddress,接着运行起来。

0012FFAC	004010C5	CALL to GetProcAddress from buggers3.004010BF
0012FFB0	7C800000	hModule = 7C800000 (kernel32)
0012FFB4	00403020	ProcNameOrOrdinal = "FreeLibrary"
0012FFB8	7FFDE000	
0012FFBC	7C920738	ntdll.7C920738

断在了 GetProcAddress 函数的入口处,程序调用 GetProcAddress 加载一些 API 函数,如果我们对哪些 API 函数感兴趣,我们可以执行到返回,就知道了该函数的地址了,然后使用 bp EAX 断这个函数,因为 EAX 中保存了 GetProcAddress 获取到的函数地址。这个函数我们不感兴趣,我直接按 F9 键运行起来。

0012FFAC	00401129	CALL to GetProcAddress from buggers3.00401123
0012FFB0	7C800000	hModule = 7C800000 (kernel32)
0012FFB4	00403030	ProcNameOrOrdinal = "CreateToolhelp32Snapshot"
0012FFB8	7FFDE000	
0012FFBC	7C920738	ntdll.7C920738
0012FFC0	FFFFFFFF	

运行几次后,我们找到了一个可疑的 API 函数 CreateToolhelp32Snapshot,你可能会问,你是怎么知道的,因为我知道这种检测方法,所以我介绍它,让大家知道哪些 API 函数还可以用于检测 OD。

好了,现在我们选择主菜单项 Debug-Execute till return 来执行到返回。

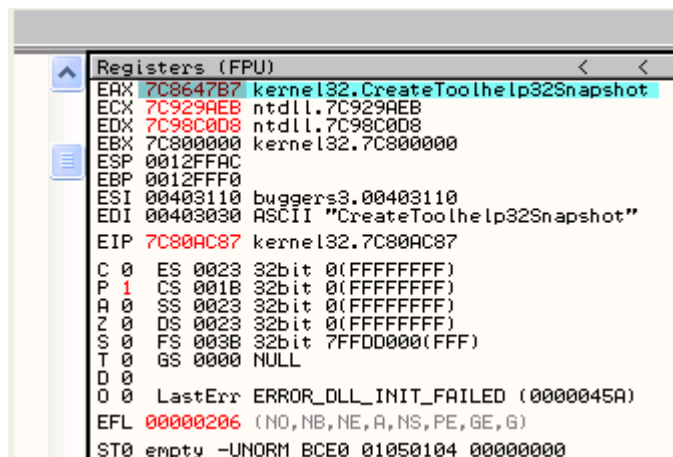
OllyDbg - buggers3.exe - [CPU - main]

File View Debug Plugins Options Window Help

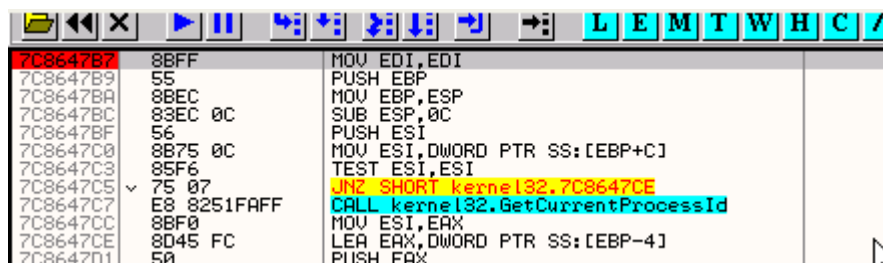
Run F9  
Pause F12  
Restart Ctrl+F2  
Close Alt+F2  
Step into F7  
Step over F8  
Animate into Ctrl+F7  
Animate over Ctrl+F8  
Execute till return Ctrl+F9  
Execute till user code Alt+F9

7C80AC28	8BFF	
7C80AC2A	55	
7C80AC2B	8BEC	
7C80AC2D	51	
7C80AC2E	51	
7C80AC2F	53	
7C80AC30	57	
7C80AC31	8B7D	
7C80AC34	BB F	
7C80AC39	3BF8	
7C80AC3B	0F86	
7C80AC41	57	
7C80AC42	8D45	
7C80AC45	50	
7C80AC46	FF15	
7C80AC4C	8D45	
7C80AC4F	50	
7C80AC50	6A 0	
7C80AC52	8D45	
7C80AC55	50	
7C80AC56	6A 0	
7C80AC58	FF75	
7C80AC5B	50	
7C80AC70	FF75 08	PUSH DWORD PTR SS:[EBP+8]
7C80AC73	E8 AAEFFFFF	CALL kernel32.7C809922
7C80AC78	3945 0C	CMP DWORD PTR SS:[EBP+C],EAX
7C80AC7B	0F84 12600300	JE kernel32.7C840C98
7C80AC81	8B45 0C	MOV EAX,DWORD PTR SS:[EBP+C]
7C80AC84	5F	POP EDI
7C80AC85	5B	POP EBX
7C80AC86	C9	LEAVE
7C80AC87	C2 0800	RETN 8
7C80AC8A	837D 10 00	CMP DWORD PTR SS:[EBP+10],0
7C80AC8E	0F95 81E6FFFF	JNZ kernel32.7C809315
7C80AC94	33FF	XOR EDI,EDI
7C80AC96	E9 81E6FFFF	JMP kernel32.7C80931C
7C80AC9B	8B4E 08	MOV ECX,DWORD PTR DS:[ESI+8]

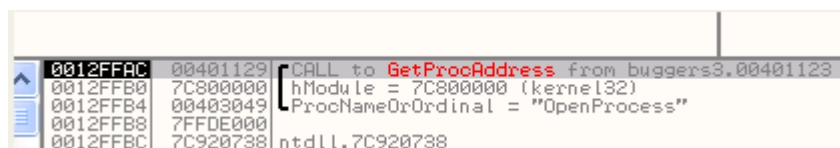
现在我们到了 RET 指令处,并且 EAX 保存了 CreateToolhelp32Snapshot 的函数地址,我们使用 BP EAX 该函数设置一个断点。



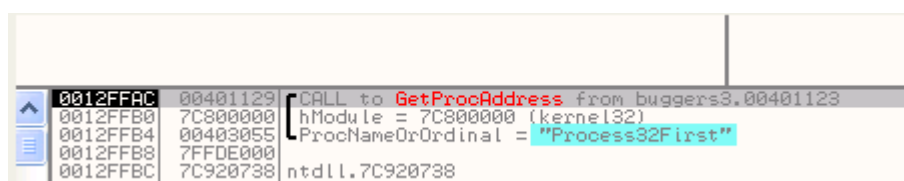
下面是该 API 断点的位置。



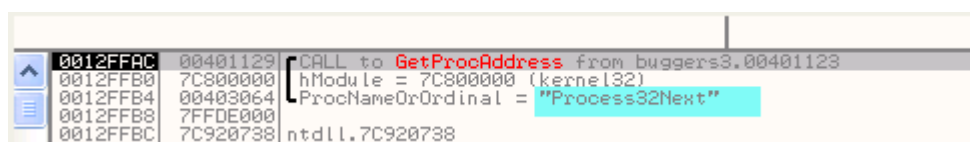
我们继续运行,看看有没有其他的可疑的 API 函数的被加载。



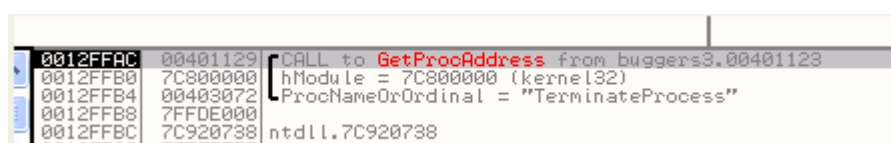
恩,OpenProcess 这个函数也是一个可疑函数,其可以获取进程的句柄(我们上一章节已经讨论过了),我们执行到返回,接着使用 BP EAX 给该函数设置断点。



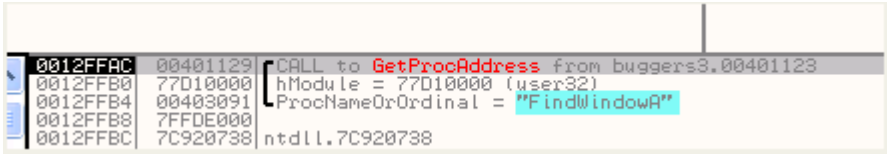
嘿嘿,Process32First 又一个可疑的 API 函数,我们同样执行到返回,然后 BP EAX 给该函数设置断点,接着对下一个可疑的函数 Process32Next 进行同样的操作。



接下来是 TerminateProcess。

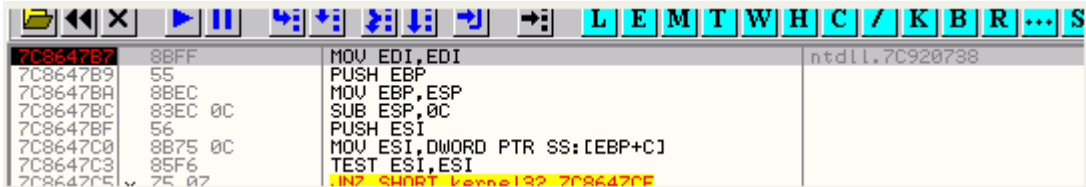


我们知道这个函数是用来关闭 OllyDbg 的。因为必须检测 OD 进程才会执行该函数,所以不必给该函数设置断点,但是为了安全起见我们还是给该函数设置断点吧,嘿嘿。

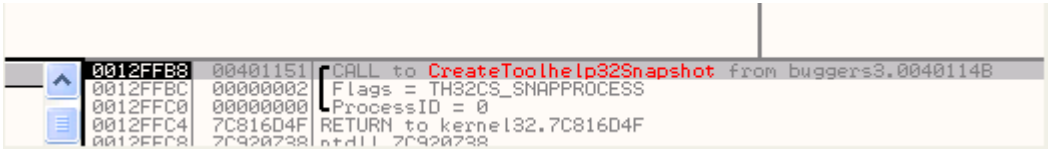


嘿嘿,FindWindowA 又一个可疑的函数,依然按照上面的方法给该函数设置断点。

我们继续 F9 键运行就断在了 CreateToolhelp32Snapshot 的入口处。



堆栈情况如下:



让我们来看看 MSDN 中关于这个函数的说明。

## CreateToolhelp32Snapshot

Takes a snapshot of the processes and the heaps, modules, and threads used by the processes.

**HANDLE** WINAPI CreateToolhelp32Snapshot (DWORD dwFlags,  
          DWORD th32ProcessID);

**Parameters**

*dwFlags*

Flags specifying portions of the system to include in the snapshot. These values are defined:

TH32CS_INHERIT	Indicates that the snapshot handle is to be inheritable.
TH32CS_SNAPALL	Equivalent to specifying the TH32CS_SNAPHEAPLIST, TH32CS_SNAPMODULE, TH32CS_SNAPPROCESS, and TH32CS_SNAPTHREAD values.
TH32CS_SNAPHEAPLIST	Includes the heap list of the specified process in the snapshot.
TH32CS_SNAPMODULE	Includes the module list of the specified process in the snapshot.
TH32CS_SNAPPROCESS	Includes the Win32 process list in the snapshot.
TH32CS_SNAPTHREAD	Includes the Win32 thread list in the snapshot.

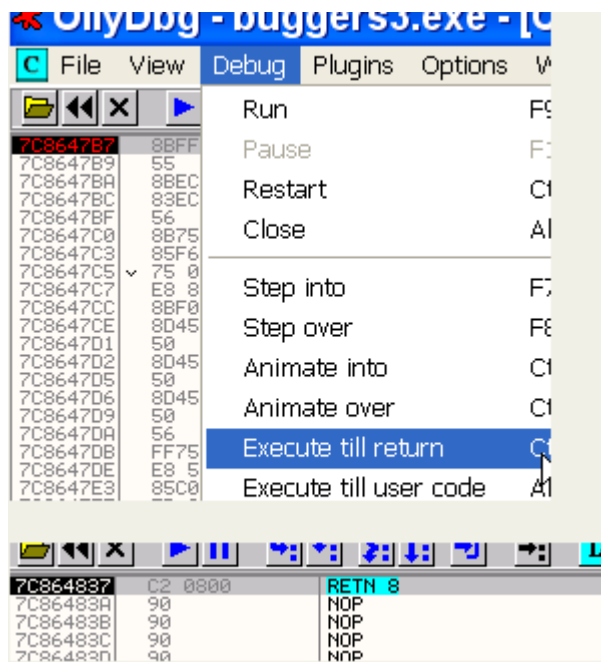
*th32ProcessID*

Process identifier. This parameter can be zero to indicate the current process. This parameter is used when the TH32CS\_SNAPHEAPLIST or TH32CS\_SNAPMODULE value is specified. Otherwise, it is ignored.

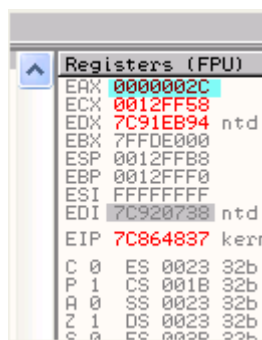
**Return Value**

Returns an open handle to the specified snapshot if successful or - 1 otherwise.

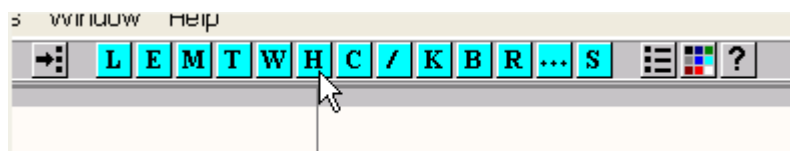
该函数是该当前机器上面运行的所以进程列表创建一个快照,但是返回给我们的仅仅是该快照的句柄,并且没有什么用于保存进程列表的缓冲区之类的参数,我们直接执行到返回。



EAX 中保存了进程快照的句柄。



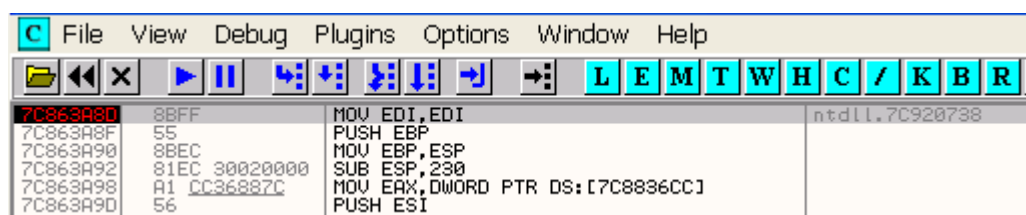
我机器上返回的进程快照句柄是 2C,我们查看一下该程序的句柄列表。



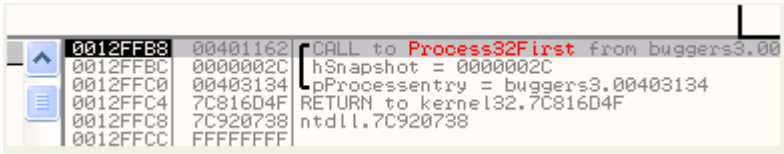
我们单击工具栏上面 H 按钮打开句柄列表窗口。

00000018	Port	3.	001F0001		
00000010	Section	47.	000F001F		
0000002C	Section	2.	000F0007		
00000020	WindowStation	100.	000F0037F		\Windows\WindowStations\WinSta0
00000028	WindowStation	100.	000F0037F		\Windows\WindowStations\WinSta0

我们发现句柄列表中并没有 2C 这个句柄值,不过还好,我们成功了创建了进程快照并获取到了进程快照的句柄,我们运行起来,看看该程序哪里使用了进程列表。



断在了 Process32First 这个 API 函数的入口处,该函数配合 Process32Next 这个 API 函数可以读取进程快照中所有正在运行的进程的相关信息。

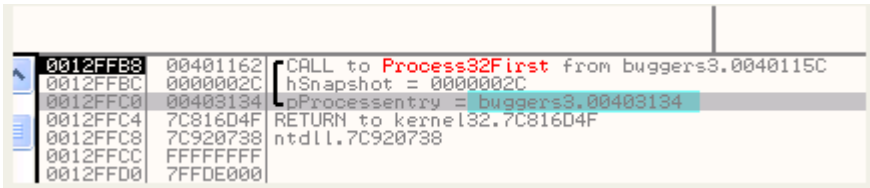


好了,我们来看看 MSDN 中关于这个函数的说明。

Process32First

Retrieves information about the first process encountered in a system snapshot.  
`BOOL WINAPI Process32First(HANDLE hSnapshot, LPPROCESSENTRY32 lppe);`  
**Parameters**  
*hSnapshot*  
Handle of the snapshot returned from a previous call to the [CreateToolhelp32Snapshot](#) function.  
*lppe*  
Address of a [PROCESSENTRY32](#) structure.

该函数用于获取第一个参数也就是进程快照(我这里是 2C)中的第一个进程的信息。第二个参数为 PROCESSENTRY32(进程相关信息)的结构体的指针。



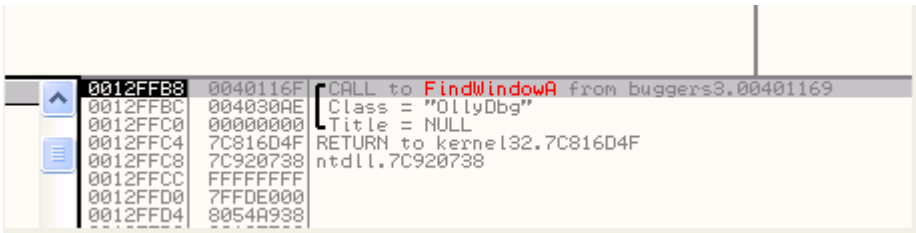
这个函数仅仅是用来获取第一个进程的信息的,Process32Next 才是用来获取后面的进程的信息的。

Address	Hex dump	ASCII
00403134	28 01 00 00 00 00 00 00	{0.....
0040313C	00 00 00 00 00 00 00 00	.....
00403144	00 00 00 00 00 00 00 00	.....
0040314C	00 00 00 00 00 00 00 00	.....
00403154	00 00 00 00 00 00 00 00	.....
0040315C	00 00 00 00 00 00 00 00	.....
00403164	00 00 00 00 00 00 00 00	.....
0040316C	00 00 00 00 00 00 00 00	.....
00403174	00 00 00 00 00 00 00 00	.....
0040317C	00 00 00 00 00 00 00 00	.....

我们在数据窗口中转到 PROCESSENTRY32 结构体的首地址处,接着我们执行到返回就可以获取到第一个进程的相关信息了。

Address	Hex dump	ASCII
00403134	28 01 00 00 00 00 00 00	{0.....
0040313C	00 00 00 00 00 00 00 00	.....
00403144	00 00 00 00 01 00 00 00	...0...
0040314C	00 00 00 00 00 00 00 00	.....
00403154	00 00 00 00 5B 53 79 73	...[Sys
0040315C	74 65 6D 20 50 72 6F 63	tem Proc
00403164	65 73 73 5D 00 00 00 00	ess]....
0040316C	00 00 00 00 00 00 00 00	.....
00403174	00 00 00 00 00 00 00 00	.....

我们可以看到第一个进程的名称,第一个进程总是 System Process,我们继续运行。

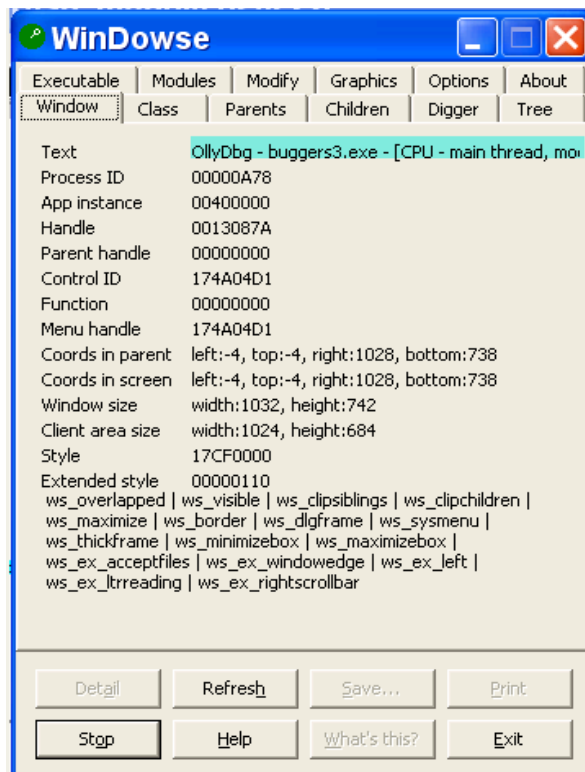


嘿嘿,这里调用了 FindWindowA,由于 OlllyDbg 的窗口标题名和窗口类名是同名的,所以 FindWindowA 也可以指定第一个参数窗口

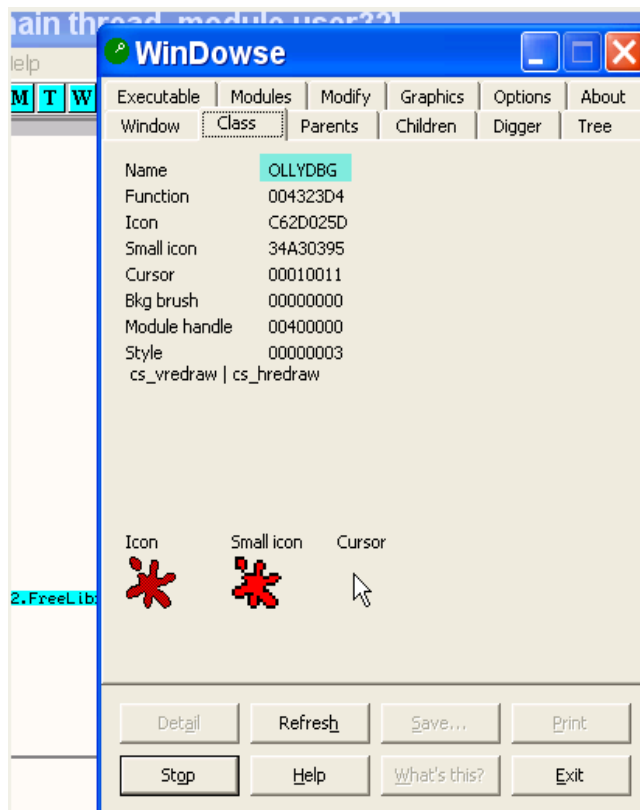
标题名为“OllyDbg”,当前该程序指定是第二个参数窗口类名,同样也是“OllyDbg”。

我们可以使用一个实用的小工具 WinDowse 来获取窗口类名(其实 VC 自带的 SPY++也可以,(\*^\_\_^\*) 嘻嘻……)

我们知道 OllyDbg 有对应的插件可以用于查看窗口的相关信息。但是 WinDowse 这款工具获取的信息更加详细一些,我们安装这个工具并运行起来。



我们可以看到 Window 标签页中显示了 OllyDbg 的窗口标题名并且 Class 标签页中显示其窗口类名。



正如我们看到的都是 OllyDbg。

我们可以看到 FindWindowA 返回的是指定窗口的句柄,通过该窗口句柄,我们可以对该窗口进行任何操作。

FindWindow

Quick Info

The **FindWindow** function retrieves the handle to the top-level window whose class name and window name match the specified strings. This function does not search child windows.

**HWND FindWindow(**  
LPCTSTR *lpClassName*, // pointer to class name  
LPCTSTR *lpWindowName* // pointer to window name  
**);**

**Parameters**

*lpClassName*  
Points to a null-terminated string that specifies the class name or is an atom that identifies the class-name string. If this parameter is an atom, it must be a global atom created by a previous call to the [GlobalAddAtom](#) function. The atom, a 16-bit value, must be placed in the low-order word of *lpClassName*; the high-order word must be zero.

*lpWindowName*  
Points to a null-terminated string that specifies the window name (the window's title). If this parameter is NULL, all window names match.

**Return Values**

If the function succeeds, the return value is the handle to the window that has the specified class name and window name.

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

大家没有必要同时设置窗口类名和窗口标题名,你只需要任选择其一,另一个参数赋值为 NULL 即可。

0012FFB8	0040116F	CALL to FindWindowA from buggers3.00401169
0012FFBC	004030AE	Class = "OllyDbg"
0012FFC0	00000000	Title = NULL
0012FFC4	7C816D4F	RETURN to kernel32.7C816D4F

好了,现在我们执行到返回,看看该函数是否会返回 OD 的窗口句柄。

Registers (FPU)

EAX 0013087A UNICODE "WinSxS\  
ECX 7C92056D ntdll.7C92056D  
EDX 00140608  
EBX 7FFDE000  
ESP 0012FFB8  
EBP 0012FFF0  
ESI FFFFFFFF  
EDI 7C920738 ntdll.7C920738  
EIP 77D2E597 user32.77D2E597  
C 0 ES 0023 32bit 0(FFFFFFFF  
P 1 CS 001B 32bit 0(FFFFFFFF  
A 0 SS 0023 32bit 0(FFFFFFFF  
Z 1 DS 0023 32bit 0(FFFFFFFF  
S 0 FS 003B 32bit 7FFDD000(F

hread module user32

WinDowse

Executable Modules Modify Graphics Options About

Window Class Parents Children Digger Tree

Text OllyDbg - buggers3.exe - [CPU - main thread, mo

Process ID 00000A78

App instance 00400000

Handle 0013087A

Parent handle 00000000

Control ID 174A04D1

Function 00000000

Menu handle 174A04D1

Coords in parent left:-4, top:-4, right:1028, bottom:738

Coords in screen left:-4, top:-4, right:1028, bottom:738

Window size width:1032, height:742

Client area size width:1024, height:684

Style 17CF0000



恩,正如我们看到的,返回的窗口句柄值跟 WinDows 上面显示的窗口句柄值一致。

好吧,我们继续跟,看看该程序获取了 OD 的窗口句柄会干些什么。

00401164	. 68 AE304000	PUSH buggers3.004030AE	ASCII "OllyDbg"
00401169	. FF15 28314000	CALL DWORD PTR DS:[403128]	user32.FindWindowA
0040116F	. 83F8 00	CMP EAX,0	
00401172	. 0BC0	OR EAX,EAX	
00401174	. 75 04	JNZ SHORT buggers3.0040117A	
00401176	. 7C 27	JL SHORT buggers3.0040119F	
00401178	. EB 25	JMP SHORT buggers3.0040119F	
0040117A	. 50	PUSH EAX	
0040117B	. 56	PUSH ESI	
0040117C	. 57	PUSH EDI	
0040117D	. BF 01000000	MOV EDI,1	
00401182	. BE 2C314000	MOV ESI,buggers3.0040312C	
00401187	. FF36	PUSH DWORD PTR DS:[ESI]	
00401189	. FF15 0C314000	CALL DWORD PTR DS:[40310C]	kernel32.FreeLibrary
0040118F	. 83C6 04	ADD ESI,4	
00401192	. 4F	DEC EDI	
00401193	. 75 F2	JNZ SHORT buggers3.00401187	
00401195	. 5F	POP EDI	
00401196	. 5E	POP ESI	
00401197	. 58	POP EAX	
00401198	. 6A 00	PUSH 0	
0040119A	. E8 57000000	CALL <JMP.&kernel32.ExitProcess>	[ExitCode = 0 ExitProcess
0040119F	. 68 B6304000	PUSH buggers3.004030B6	ASCII "OLLYDBG.EXE"
004011A4	. 68 58314000	PUSH buggers3.00403158	ASCII "[System Process]"
004011A9	. FF15 24314000	CALL DWORD PTR DS:[403124]	kernel32.lstrcpA
004011AF	. 0BC0	OR EAX,EAX	
004011B1	. 75 2F	JNZ SHORT buggers3.004011E2	
004011B3	. FF35 3C314000	PUSH DWORD PTR DS:[40313C]	

这里判断获取到的窗口句柄是否为空,如果窗口句柄为空,说明不存在 OllyDbg 窗口,如果返回的窗口句柄非空,该程序就会调用 ExitProcess 退出进程。

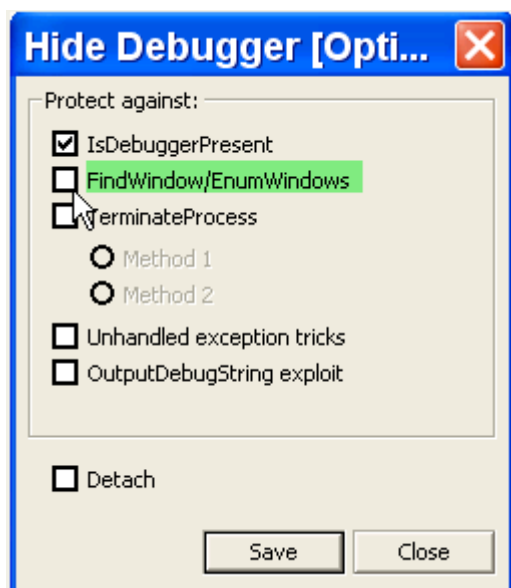
00401164	. 68 AE304000	PUSH buggers3.004030AE	ASCII "OllyDbg"
00401169	. FF15 28314000	CALL DWORD PTR DS:[403128]	user32.FindWindowA
0040116F	. 83F8 00	CMP EAX,0	
00401172	. 0BC0	OR EAX,EAX	
00401174	. 75 04	JNZ SHORT buggers3.0040117A	
00401176	. 7C 27	JL SHORT buggers3.0040119F	
00401178	. EB 25	JMP SHORT buggers3.0040119F	
0040117A	. 50	PUSH EAX	
0040117B	. 56	PUSH ESI	
0040117C	. 57	PUSH EDI	
0040117D	. BF 01000000	MOV EDI,1	

直接跳转到退出进程的代码块并且不显示任何东西出来。

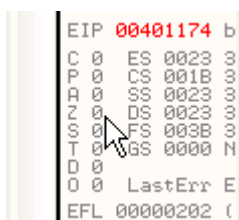
00401169	. FF15 28314000	CALL DWORD PTR DS:[403128]	user32.FindWindowA
0040116F	. 83F8 00	CMP EAX,0	
00401172	. 0BC0	OR EAX,EAX	
00401174	. 75 04	JNZ SHORT buggers3.0040117A	
00401176	. 7C 27	JL SHORT buggers3.0040119F	
00401178	. EB 25	JMP SHORT buggers3.0040119F	
0040117A	. 50	PUSH EAX	
0040117B	. 56	PUSH ESI	
0040117C	. 57	PUSH EDI	
0040117D	. BF 01000000	MOV EDI,1	
00401182	. BE 2C314000	MOV ESI,buggers3.0040312C	
00401187	. FF36	PUSH DWORD PTR DS:[ESI]	
00401189	. FF15 0C314000	CALL DWORD PTR DS:[40310C]	kernel32.FreeLibrary
0040118F	. 83C6 04	ADD ESI,4	
00401192	. 4F	DEC EDI	
00401193	. 75 F2	JNZ SHORT buggers3.00401187	
00401195	. 5F	POP EDI	
00401196	. 5E	POP ESI	
00401197	. 58	POP EAX	
00401198	. 6A 00	PUSH 0	
0040119A	. E8 57000000	CALL <JMP.&kernel32.ExitProcess>	[ExitCode = 0 ExitProcess
0040119F	. 68 B6304000	PUSH buggers3.004030B6	ASCII "OLLYDBG.EXE"
004011A4	. 68 58314000	PUSH buggers3.00403158	ASCII "[System Process]"
004011A9	. FF15 24314000	CALL DWORD PTR DS:[403124]	kernel32.lstrcpA
004011AF	. 0BC0	OR EAX,EAX	
004011B1	. 75 2F	JNZ SHORT buggers3.004011E2	
004011B3	. FF35 3C314000	PUSH DWORD PTR DS:[40313C]	

所以,我们需要 FindWindowA 返回值 EAX 为空。

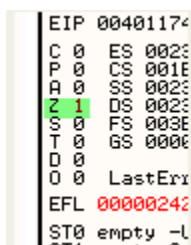
好了,我们现在知道如何手工绕过该反调试了,下面直接使用 HideDebugger1.23 版插件来绕过该反调试吧,我们来看看该插件的配置吧。



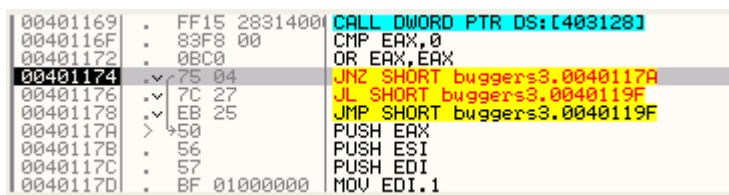
我们可以看到第二个选项,可以绕过 FindWindow 和 EnumWindows 检测 OD 窗口方法,首先我们还是要知道如何手工绕过该反调试以及其原理是什么。好了,我们现在不设置该选项,直接重新启动 OD,手工来实现跳过 ExitProcess 代码块并且继续执行。



我们双击零标志 Z 将其修改为 1,这样 JNZ 条件跳转就不会实现了。



现在 JNZ 指令不会跳转了。



接下来将会执行 JMP 指令跳过 ExitProcess 的调用代码。

00401172	00BC0	OR EAX,EAX	
00401174	75 04	JNZ SHORT buggers3.0040117A	
00401176	7C 27	JL SHORT buggers3.0040119F	
00401178	EB 25	JMP SHORT buggers3.0040119F	
0040117A	50	PUSH EAX	
0040117B	56	PUSH ESI	
0040117C	57	PUSH EDI	
0040117D	BF 01000000	MOV EDI,1	
00401182	BE 2C314000	MOV ESI,buggers3.0040312C	
00401187	FF36	PUSH DWORD PTR DS:[ESI]	
00401189	FF15 0C314000	CALL DWORD PTR DS:[40310C]	kernel32.FreeLibrary
0040118F	83C6 04	ADD ESI,4	
00401192	4F	DEC EDI	
00401193	75 F2	JNZ SHORT buggers3.00401187	
00401195	5F	POP EDI	
00401196	5E	POP ESI	
00401197	58	POP EAX	
00401198	6A 00	PUSH 0	
0040119A	E8 57000000	CALL <JMP.&kernel32.ExitProcess>	ExitCode = 0 ExitProcess
0040119F	68 B6304000	PUSH buggers3.004030B6	ASCII "OLLYDBG.EXE"
004011A4	68 58314000	PUSH buggers3.00403158	ASCII "[System Process
004011A9	FF15 24314000	CALL DWORD PTR DS:[403124]	kernel32.lstrcpA
004011AF	00BC0	OR EAX,EAX	
004011B1	75 2F	JNZ SHORT buggers3.004011E2	
004011B3	FF35 3C314000	PUSH DWORD PTR DS:[40313C]	

好了,我们继续,介绍如何绕过 FindWindowA 了,现在继续讨论绕过检测 OD 进程名的方法,运行起来。

0012FFB8	004011F3	CALL to Process32Next from buggers3.004011ED
0012FFBC	0000002C	hSnapshot = 0000002C
0012FFC0	00403134	pProcessentry = buggers3.00403134
0012FFC4	7C816D4F	RETURN to kernel32.7C816D4F
0012FFC8	7C920738	ntdll.7C920738
0012FFCC	FFFFFFFF	
0012FFD0	7FFDE000	

断了下来,现在调用的是 Process32Next,获取进程快照中第二个进程的相关信息,并且该进程的相关信息会保存在 403134 指向缓冲区中。

我们执行到返回看看保存了什么。

Address	Hex dump	ASCII
00403134	28 01 00 00 00 00 00 00	(0.....
0040313C	04 00 00 00 00 00 00 00	.....
00403144	00 00 00 00 41 00 00 00	...A...
0040314C	00 00 00 00 08 00 00 00	...8...
00403154	00 00 00 00 53 79 73 74	...Syst
0040315C	65 6D 00 20 50 72 6F 63	em. Proc
00403164	65 73 73 5D 00 00 00 00	ess]....
0040316C	00 00 00 00 00 00 00 00	.....
00403174	00 00 00 00 00 00 00 00	.....
0040317C	00 00 00 00 00 00 00 00	.....

现在获取到的是 System 进程,PID 为 4,我们结合任务管理器来看。

NETFileServerEngine.exe	148	SYSTEM	00	18.644 KB
System	4	SYSTEM	00	216 KB
Proceso inactivo del sistema	0	SYSTEM	98	16 KB

同理,我们就可以看到获取到的每个进程以及其相关信息。

0040126D	00	DB 00	
0040126E	00	DB 00	
0040126F	00	DB 00	
00401270	68 58314000	PUSH buggers3.00403158	ASCII "System"
00401275	68 A0314000	PUSH buggers3.004031A0	ASCII "buggers3.exe"
0040127A	FF15 24314000	CALL DWORD PTR DS:[403124]	kernel32.lstrcpA
00401280	85C0	TEST EAX,EAX	
00401282	0F85 17FFFFFF	JNZ buggers3.0040119F	
00401288	68 CC314000	PUSH buggers3.004031CC	ASCII "MessageBoxA"
0040128D	FF35 2C314000	PUSH DWORD PTR DS:[40312C]	user32.77D10000
00401293	FF15 08314000	CALL DWORD PTR DS:[403108]	kernel32.GetProcAddress
00401299	6A 00	PUSH 0	
0040129B	6A 00	PUSH 0	
0040129D	68 CD304000	PUSH buggers3.004030CD	ASCII "not debugged!"
004012A2	68 CD304000	PUSH buggers3.004030CD	ASCII "not debugged!"
004012A7	6A 00	PUSH 0	
004012A9	FFD0	CALL EAX	
004012AB	E9 E5FEFFFF	JMP buggers3.00401195	
004012B0	0000	ADD BYTE PTR DS:[EAX],AL	
004012B2	0000	ADD BYTE PTR DS:[EAX],AL	
004012B4	0000	ADD BYTE PTR DS:[EAX],AL	
004012B6	0000	ADD BYTE PTR DS:[EAX],AL	

这里我们可以看到 lstrcpA 这个 API 函数,它将 "System" 与 "buggers3.exe" 两个字符串进行比较,即比较当前获取的进程名与

该 CrackMe 名称,如果它们相等,将会调用 MessageBoxA 弹出 not debugged!没有被调试的信息。这里,两者并不相等,所以我们继续跟。

0040126B	00	DB 00	
0040126C	00	DB 00	
0040126D	00	DB 00	
0040126E	00	DB 00	
0040126F	00	DB 00	
00401270	> 68 58314000	PUSH buggers3.00403158	ASCII "System"
00401275	. 68 00314000	PUSH buggers3.004031A0	ASCII "buggers3.exe"
0040127A	. FF15 24314000	CALL DWORD PTR DS:[403124]	kernel32.lstrcmpA
00401280	. 85C0	TEST EAX,EAX	
00401282	^ 0F85 17FFFFFF	JNZ buggers3.0040119F	
00401288	. 68 CC314000	PUSH buggers3.004031CC	ASCII "MessageBoxA"
0040128D	. FF35 2C314000	PUSH DWORD PTR DS:[40312C]	user32.77D10000
00401293	. FF15 00314000	CALL DWORD PTR DS:[4031A8]	kernel32.GetProcAddress

上面两个字符串不相等,所以比较结果为 FFFFFFFF。

Registers (FPU)	
EAX	FFFFFFFF
ECX	0000A6B8
EDX	0000000E
EBX	7FFDE000
ESP	0012FFC4
EBP	0012FFFF
ESI	FFFFFFFF
EDI	7C920738 ntdll
EIP	00401282 bugge
C 0	ES 0023 32bit
P 1	CS 001B 32bit
A 0	SS 0023 32bit
Z 0	DS 0023 32bit

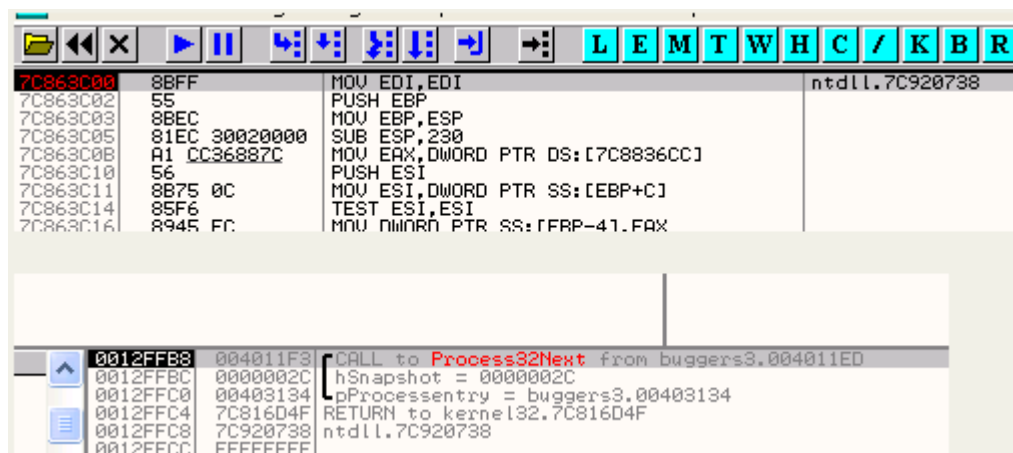
由于结果不为零,将会跳转至 40119F 地址处。

0040119F	> 68 B6304000	PUSH buggers3.004030B6	ASCII "OLLYDBG.EXE"
004011A4	. 68 58314000	PUSH buggers3.00403158	ASCII "System"
004011A9	. FF15 24314000	CALL DWORD PTR DS:[403124]	kernel32.lstrcmpA
004011AF	. 0BC0	OR EAX,EAX	
004011B1	. 75 2F	JNZ SHORT buggers3.004011E2	
004011B3	. FF35 3C314000	PUSH DWORD PTR DS:[40313C]	
004011B9	. 6A 01	PUSH 1	
004011BB	. 68 FF0F1F00	PUSH 1F0FFF	
004011C0	. FF15 14314000	CALL DWORD PTR DS:[403114]	kernel32.OpenProcess
004011C6	. A3 64324000	MOV DWORD PTR DS:[403264],EAX	
004011CB	. 6A 00	PUSH 0	
004011CD	. FF35 64324000	PUSH DWORD PTR DS:[403264]	
004011D3	. FF15 20314000	CALL DWORD PTR DS:[403120]	kernel32.TerminateProcess
004011D9	. 6A 00	PUSH 0	ExitCode = 0
004011DB	. E8 16000000	CALL <JMP.&kernel32.ExitProcess>	ExitProcess
004011E0	. EB 11	JMP SHORT buggers3.004011F3	

这里,到了比较关键的地方了,比较获取到的进程名是否为 OLLYDBG.EXE,如果是,结果为零并且 JNZ 条件跳转将不会实现,将会调用 OpenProcess 获取 OD 进程的句柄,然后通过 TerminateProcess 结束掉 OD 进程。跟上一章我们遇到的情况差不多。

00401197	. 58	POP EAX	
00401198	. 6A 00	PUSH 0	ExitCode = 0
0040119A	. E8 57000000	CALL <JMP.&kernel32.ExitProcess>	ExitProcess
0040119F	> 68 B6304000	PUSH buggers3.004030B6	ASCII "OLLYDBG.EXE"
004011A4	. 68 58314000	PUSH buggers3.00403158	ASCII "System"
004011A9	. FF15 24314000	CALL DWORD PTR DS:[403124]	kernel32.lstrcmpA
004011AF	. 0BC0	OR EAX,EAX	
004011B1	. 75 2F	JNZ SHORT buggers3.004011E2	
004011B3	. FF35 3C314000	PUSH DWORD PTR DS:[40313C]	
004011B9	. 6A 01	PUSH 1	
004011BB	. 68 FF0F1F00	PUSH 1F0FFF	
004011C0	. FF15 14314000	CALL DWORD PTR DS:[403114]	kernel32.OpenProcess
004011C6	. A3 64324000	MOV DWORD PTR DS:[403264],EAX	
004011CB	. 6A 00	PUSH 0	
004011CD	. FF35 64324000	PUSH DWORD PTR DS:[403264]	
004011D3	. FF15 20314000	CALL DWORD PTR DS:[403120]	kernel32.TerminateProcess
004011D9	. 6A 00	PUSH 0	ExitCode = 0
004011DB	. E8 16000000	CALL <JMP.&kernel32.ExitProcess>	ExitProcess
004011E0	. EB 11	JMP SHORT buggers3.004011F3	
004011E2	> 68 34314000	PUSH buggers3.00403134	
004011E7	. FF35 5C324000	PUSH DWORD PTR DS:[40325C]	
004011ED	. FF15 1C314000	CALL DWORD PTR DS:[40311C]	kernel32.Process32Next
004011F3	. EB 7B	JMP SHORT buggers3.00401270	

我们可以看到当前进程名并不是 OllYdbg.exe,所以会继续执行 Process32Next 获取下一个进程的相关信息。



我们同样是执行到返回,看看获取到的信息。

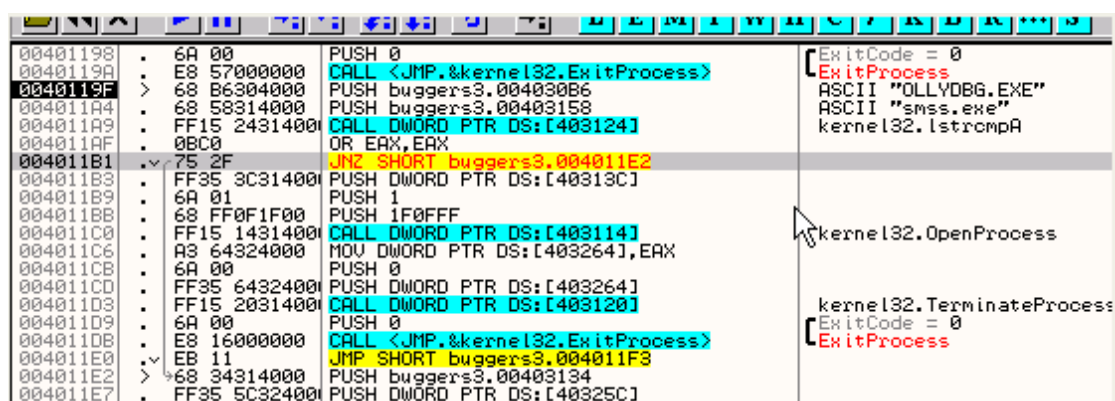
Address	Hex dump	ASCII
00403134	2E 01 00 00 00 00 00 00	(0.....
0040313C	6C 02 00 00 00 00 00 00	l0.....
00403144	00 00 00 00 03 00 00 00	...♦...
0040314C	04 00 00 00 0B 00 00 00	♦...0...
00403154	00 00 00 00 73 6D 73 73	...smss
0040315C	2E 65 78 65 00 72 6F 63	...exe.roc
00403164	65 73 73 5D 00 00 00 00	ess]....
0040316C	00 00 00 00 00 00 00 00	.....
00403174	00 00 00 00 00 00 00 00	.....
0040317C	00 00 00 00 00 00 00 00	.....

现在获取到的进程名称为 smss.exe,其 PID 为 26C。我们结合任务管理器来看。

csrss.exe	676	SYSTEM	00	3.116 KB
smss.exe	620	SYSTEM	00	100 KB
winhlp32.exe	580	Ricardo	00	1.984 KB
ComproScheduler.exe	520	Ricardo	00	1.020 KB
fdm.exe	484	Ricardo	00	5.860 KB
GoogleDesktop.exe	468	Ricardo	00	640 KB

任务管理器中显示的 smss.exe 进程的 PID,十进制为 620,十六进制即 26C。

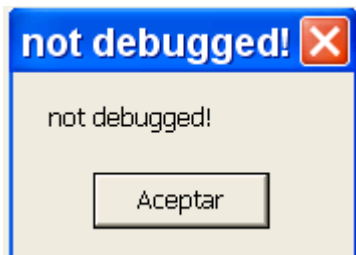
恩,接下来该 CrackMe 会逐一比较每个进程看是否为 OLLYDBG.EXE。



现在我们处于 4011B1 这个条件分支处,当前找到一个进程名为 OLLYDBG.EXE,条件跳转将不会发生并且会执行下面的关闭 OD 的代码,因此,我们需要将该 JNZ 指令修改为 JMP 指令,让关闭 OD 的代码永远得不到执行。

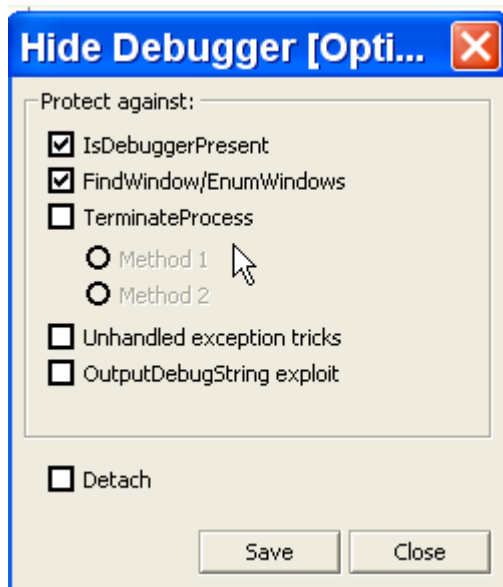
00401198	. 6A 00	PUSH 0	ExitCode = 0
0040119A	. E8 57000000	CALL <JMP.&kernel32.ExitProcess>	ExitProcess
0040119F	> 68 B6304000	PUSH buggers3.004030B6	ASCII "OLLYDBG.EXE"
004011A4	. 68 58314000	PUSH buggers3.00403158	ASCII "smss.exe"
004011A9	. FF15 24314000	CALL DWORD PTR DS:[403124]	kernel32.lstrcpA
004011AF	. 0BC0	OR EAX,EAX	
004011B1	. EB 2F	JMP SHORT buggers3.004011E2	
004011B3	. FF35 3C314000	PUSH DWORD PTR DS:[40313C]	
004011B9	. 6A 01	PUSH 1	
004011BB	. 68 FF0F1F00	PUSH 1F0FFF	
004011C0	. FF15 14314000	CALL DWORD PTR DS:[403114]	kernel32.OpenProcess
004011C6	. A3 64324000	MOV DWORD PTR DS:[403264],EAX	
004011CB	. 6A 00	PUSH 0	
004011CD	. FF35 64324000	PUSH DWORD PTR DS:[403264]	
004011D3	. FF15 20314000	CALL DWORD PTR DS:[403120]	kernel32.TerminateProcess
004011D9	. 6A 00	PUSH 0	ExitCode = 0
004011DB	. E8 16000000	CALL <JMP.&kernel32.ExitProcess>	ExitProcess
004011E0	. EB 11	JMP SHORT buggers3.004011F3	
004011E2	> 68 34314000	PUSH buggers3.00403134	
004011E7	. FF35 5C324000	PUSH DWORD PTR DS:[40325C]	
004011ED	. FF15 1C314000	CALL DWORD PTR DS:[40311C]	kernel32.Process32Next
004011F0	. EB 2D	JMP SHORT buggers3.004011F3	

现在删除所有断点运行起来。



好了,这样该反调试就被绕过了。我们知道 HideDebugger 插件也可以绕过 FindWindowA 对于 OD 窗口的检测,并且我们也可以将原版的 OLLYDBG.EXE 重命名为 PIRULO.EXE 让其找到 OLLYDBG 这个进程名。

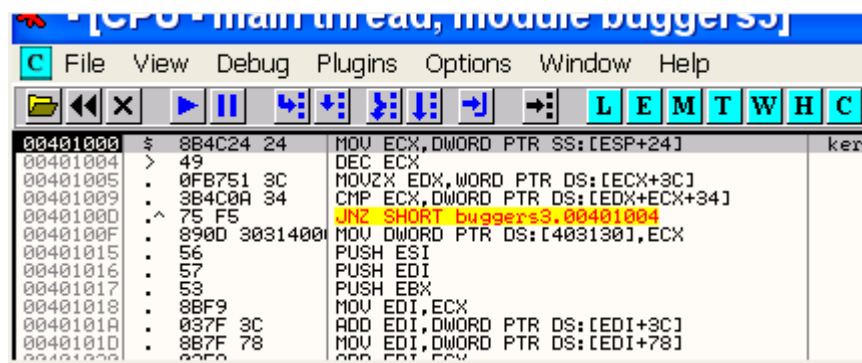
我们打开 PIRULO.EXE。



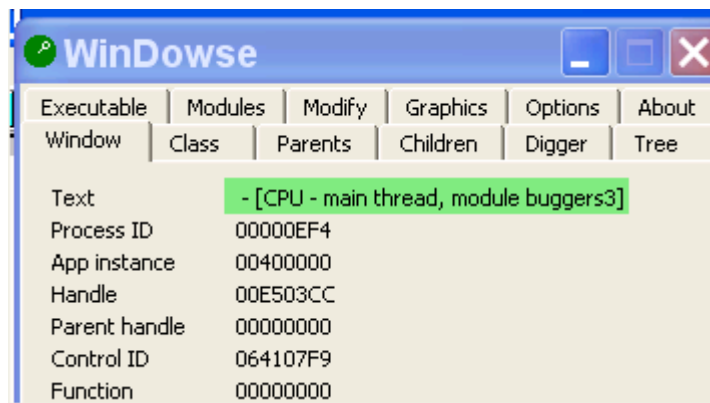
我们勾选上绕过 FindWindow 的选项,然后单击保存。



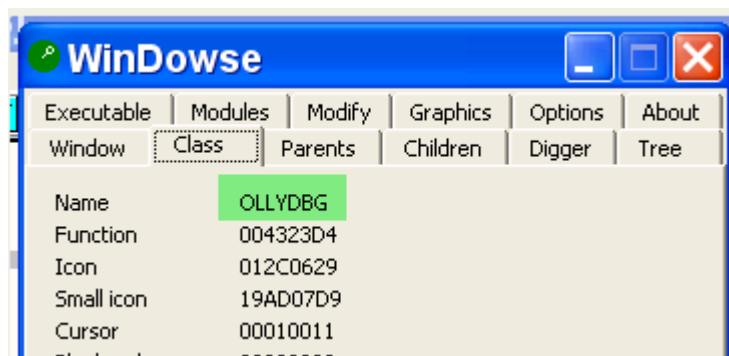
接着我们重新启动 OllyDbg。



我们加载 buggers3 之前,先来解决一个小问题,我们来看一下 WinDowse,看看 WinDowse 还是否能够检测 OD 的窗口名。



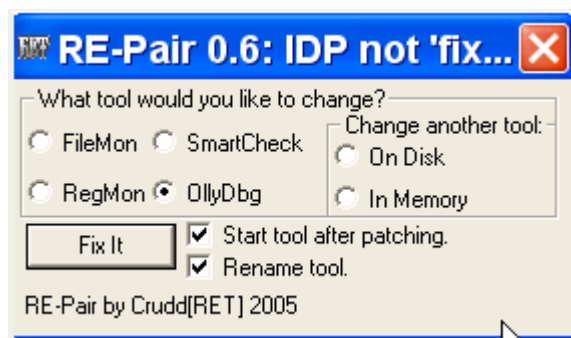
我们可以看到 OLLYDBG 并没有出现在标题栏中,那 OD 的窗口类名呢?



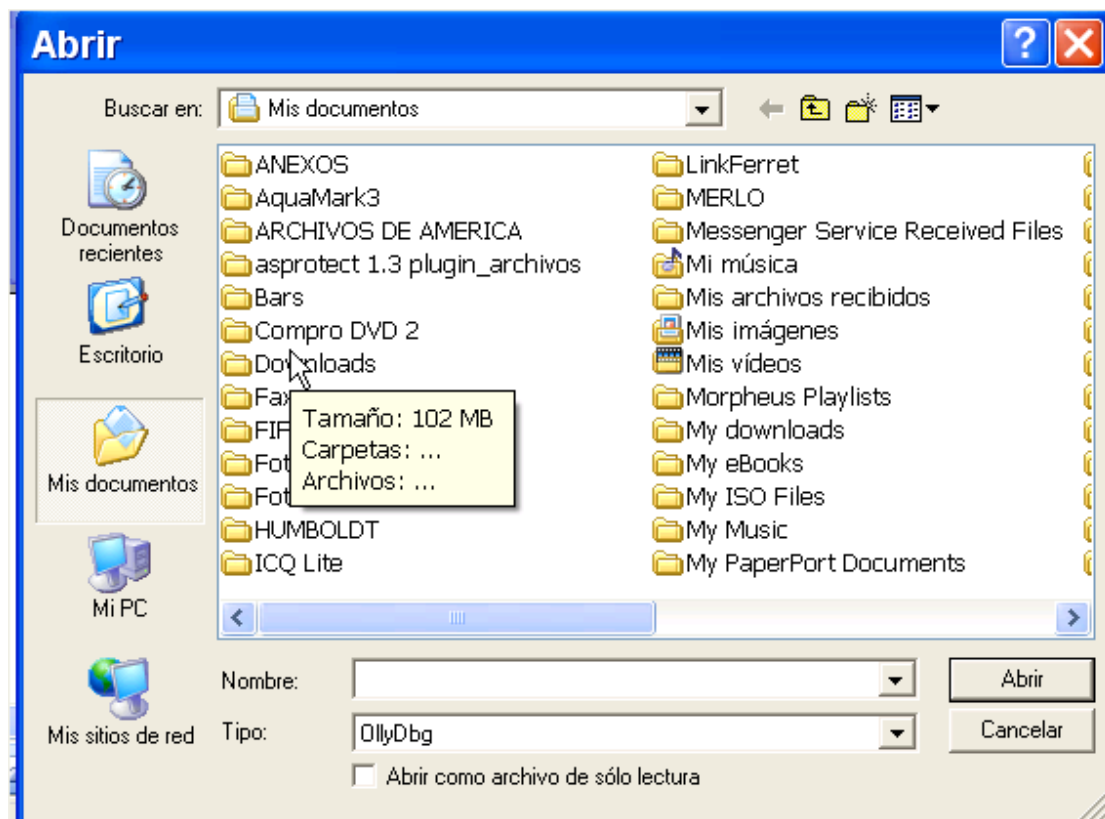
我们可以 OLLYDBG 的窗口类名被检测出来了,这里我们还需要借助另一个小工具。

它的名字叫做 repair.0.6,它是 OLLYDBG 的一个补丁程序。

好了,我们现在关闭 OllyDbg 然后运行该补丁程序。

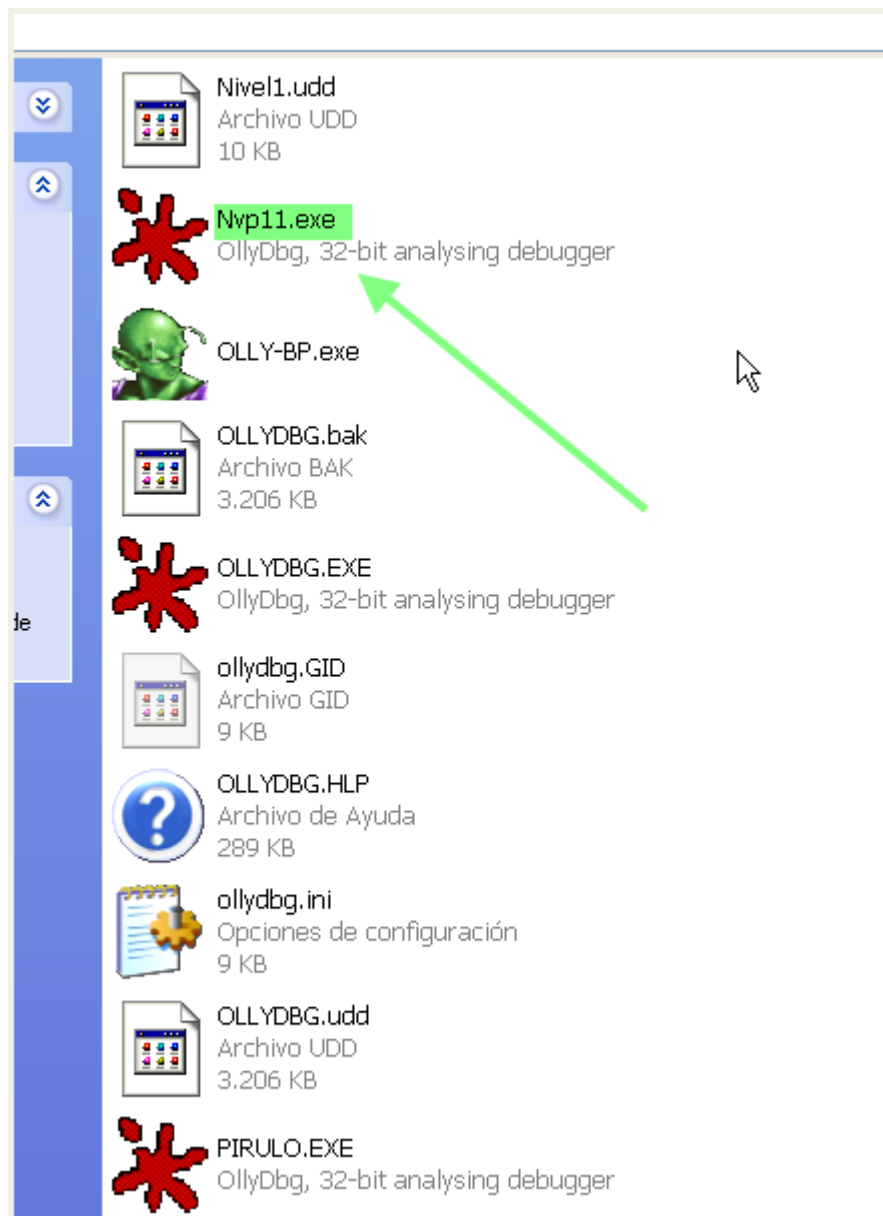




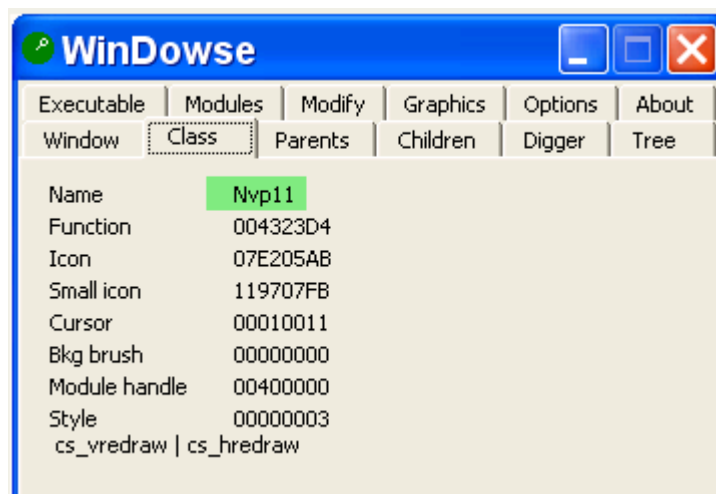


好了打完补丁以后我们现在有了第 3 个 OllyDbg 了,就是 Nvp11.exe。我们来看看 OD 所在的文件夹。

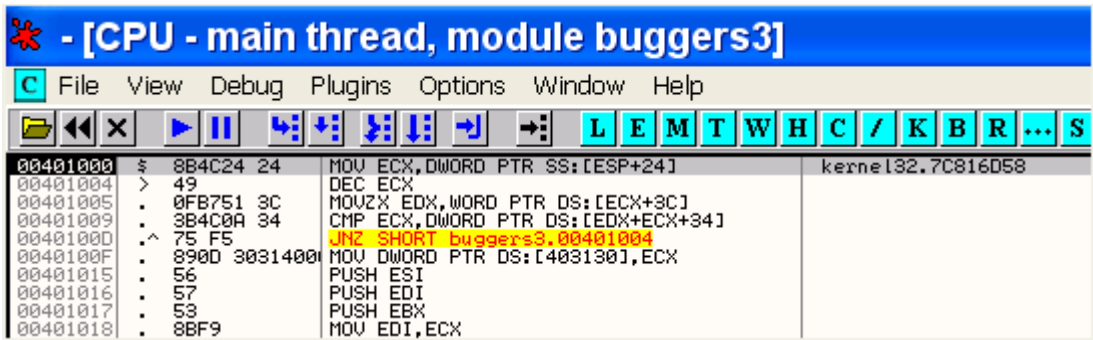




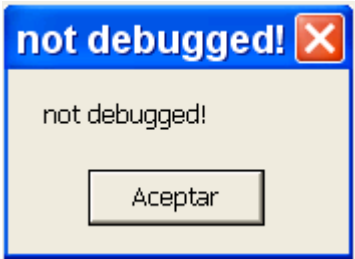
好了,我们现在运行它,看看其窗口类名。



我们可以看到现在的窗口类名为 Nvp11,进程名称也变成了 Nvp11,好了,现在我们可以完美运行 buggers3 了,我们来验证一下。



运行起来。



好了,我们给OD打了补丁以后,OllyDbg就不那么容易被检测到了,现在就不会被通过进程名,窗口名或者窗口类名的方法检测到了,嘿嘿,下一章我们继续讨论其他的反调试方法。我们首先弄明白如何手工绕过对应的反调试,然后使用插件来绕过就很简单了,嘿嘿。