

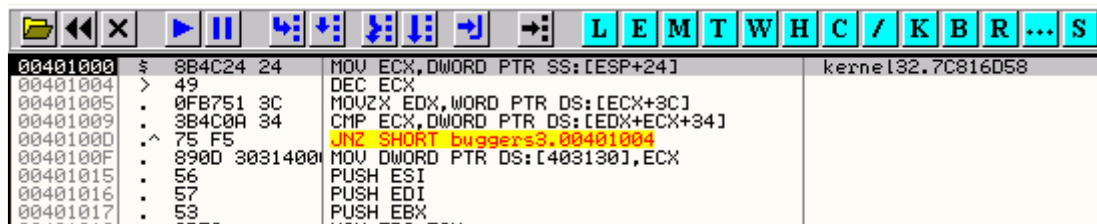
INTRODUCCION AL CRACKING CON OLLYDBG parte 21

Seguiremos ahondando sobre diferentes métodos antidebugging, hoy usaremos un crackme modificado por mí para la explicación.

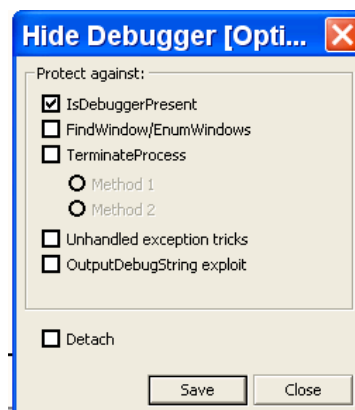
Es el crackme buggers3, al cual le hice algunos arreglos para poder explicar nuevamente la detección por nombre del proceso con otras apis, que trae este crackme, y además la detección por el nombre o clase de la ventana del OLLYDBG que también trae este crackme buggers3.

Lo abrimos con el OLLYDBG original, no el renombrado, porque estudiaremos también una variante del método que vimos en el tutorial 20, por lo tanto necesitamos que el OLLYDBG se llame OLLYDB.exe para que sea pueda ser detectado y estudiar esa protección.

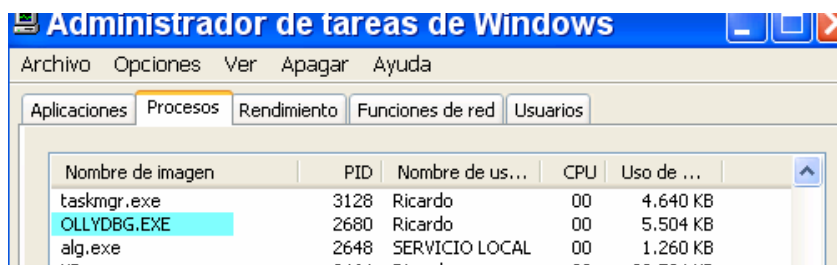
Lo abrimos entonces en el OLLYDBG.exe, solo protegido por el HideDebugger 1.23f contra la detección por la api IsDebuggerPresent.



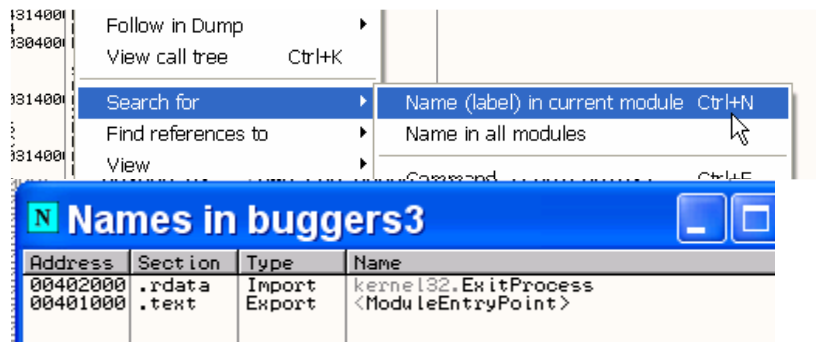
Allí esta abierto, vemos que el plugin HideDebugger esta configurado



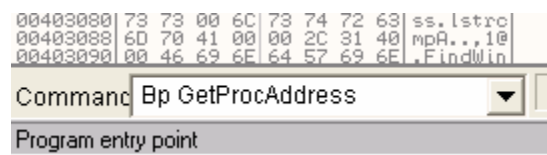
Solo para proteger al OLLYDBG contra la detección por medio de la api IsDebuggerPresent, y además vemos en la lista de procesos que usamos el OLLYDBG original, pues el proceso se llama OLLYDBG.exe



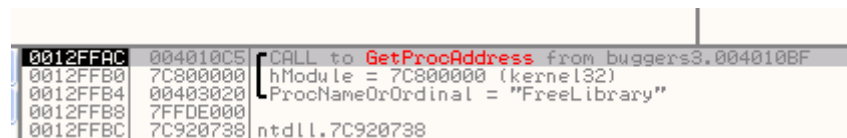
Bueno volvamos al buggers3 veamos las apis que utiliza en la lista de apis



Glup, solo tiene en la lista la api ExitProcess, el resto las debería cargar con GetProcAddress, pero GetProcAddress si ni siquiera esta en la lista si intento.

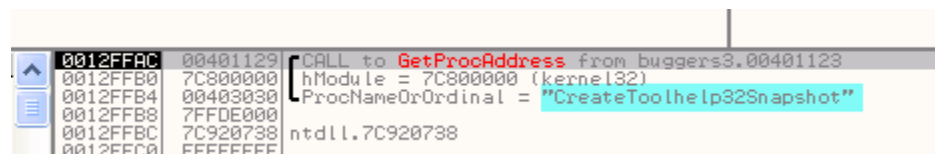


Veó que me la toma, pues nos evitamos mayores complicaciones, ahora si demos RUN.



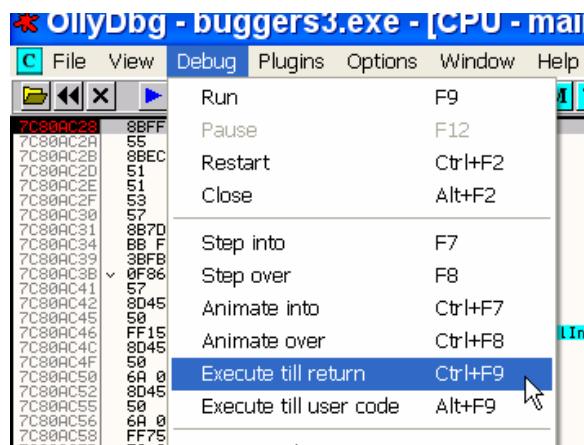
Vemos las apis que va cargando, por supuesto si vemos alguna que nos interesa, llegamos al RET para ver la dirección de la misma, y le ponemos un BP EAX, ya que en EAX estará la dirección que devuelve GetProcAddress.

En este caso no nos interesa, seguimos con F9



Para varias veces hasta que hallamos la primera sospechosa de homicidio, ustedes dirán como sabe, pues porque conozco esta protección, y por eso se las enseño, para que conozcan cuales son las apis que se pueden utilizar tanto en la versión del tute 20, como en esta diferente versión de la protección.

Bueno hago EXECUTE TILL RETURN para llegar al RET



7C80AC70	FF75 08	PUSH DWORD PTR SS:[EBP+8]
7C80AC73	E8 AAECEFFF	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	0F85 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]

Allí llegamos al RET y por supuesto en EAX esta la dirección de la api en nuestra maquina, que busca el programa, en este caso la de CreateToolhelp32Snapshot ya veremos cuando la utilice para que sirve esta api, por ahora pongámosle un BP, con BP EAX

Registers (FPU)	
EAX	7C8647B7 kernel32.CreateToolhelp32Snapshot
ECX	7C929AEB ntdll.7C929AEB
EDX	7C98C008 ntdll.7C98C008
EBX	7C800000 kernel32.7C800000
ESP	0012FFAC
EBP	0012FFF0
ESI	00403110 buggers3.00403110
EDI	00403030 ASCII "CreateToolhelp32Snapshot"
EIP	7C80AC87 kernel32.7C80AC87
C 0	ES 0023 32bit 0(FFFFFFFF)
P 1	CS 001B 32bit 0(FFFFFFFF)
A 0	SS 0023 32bit 0(FFFFFFFF)
Z 0	DS 0023 32bit 0(FFFFFFFF)
S 0	FS 003B 32bit 7FFDD000(FFF)
T 0	GS 0000 NULL
D 0	
O 0	LastErr ERROR_DLL_INIT_FAILED (0000045A)
EFL	00000206 (NO,NB,NE,A,NS,PE,GE,G)
ST0	empty -UNORM BCE0 01050104 00000000

Allí quedo puesto el BP en la api.

7C8647B7	8BFF	MOV EDI,EDI
7C8647B9	55	PUSH EBP
7C8647BA	8BEC	MOV EBP,ESP
7C8647BC	83EC 0C	SUB ESP,0C
7C8647BF	56	PUSH ESI
7C8647C0	8B75 0C	MOV ESI,DWORD PTR SS:[EBP+C]
7C8647C3	85F6	TEST ESI,ESI
7C8647C5	75 07	JNZ SHORT kernel32.7C8647CE
7C8647C7	E8 8251FAFF	CALL kernel32.GetCurrentProcessId
7C8647CC	8BF0	MOV ESI,EAX
7C8647CE	8D45 FC	LEA EAX,DWORD PTR SS:[EBP-4]
7C8647D1	50	PUSH EAX

Demos RUN nuevamente a ver si carga más apis sospechosas.

0012FFAC	00401129	CALL to GetProcAddress from buggers3.00401123
0012FFB0	7C800000	hModule = 7C800000 (kernel32)
0012FFB4	00403049	ProcNameOrOrdinal = "OpenProcess"
0012FFB8	7FFDE000	
0012FFBC	7C920738	ntdll.7C920738

Bueno esta ya sabemos que es peligrosa, pues para terminar el proceso, debe obtener el manejador o handle como vimos en la parte 20, y eso lo hace con OpenProcess, así que lleguemos al ret y pongámosle un BP EAX a esta api también.

0012FFAC	00401129	CALL to GetProcAddress from buggers3.00401123
0012FFB0	7C800000	hModule = 7C800000 (kernel32)
0012FFB4	00403055	ProcNameOrOrdinal = "Process32First"
0012FFB8	7FFDE000	
0012FFBC	7C920738	ntdll.7C920738

Otra culpable de asesinato el primer grado, jeje, ya verán porque por ahora pongámosle un BP también con el mismo método y a su hermanita Process32Next.

0012FFAC	00401129	CALL to GetProcAddress from buggers3.00401123	
0012FFB0	7C800000	hModule = 7C800000 (kernel32)	
0012FFB4	00403064	ProcNameOrOrdinal = "Process32Next"	
0012FFB8	7FFDE000		
0012FFBC	7C920738	ntdll.7C920738	

Luego viene TerminateProcess

0012FFAC	00401129	CALL to GetProcAddress from buggers3.00401123	
0012FFB0	7C800000	hModule = 7C800000 (kernel32)	
0012FFB4	00403072	ProcNameOrOrdinal = "TerminateProcess"	
0012FFB8	7FFDE000		
0012FFBC	7C920738	ntdll.7C920738	

Esta ya sabemos que nos cerrara el OLLY, sin chistar no le ponemos BP porque antes debe pasar por las otras, pero es candidata al BP siempre y culpable seguro jeje.

0012FFAC	00401129	CALL to GetProcAddress from buggers3.00401123	
0012FFB0	77D10000	hModule = 77D10000 (user32)	
0012FFB4	00403091	ProcNameOrOrdinal = "FindWindowA"	
0012FFB8	7FFDE000		
0012FFBC	7C920738	ntdll.7C920738	

Otra reculpable jeje le ponemos un BP también con el método acostumbrado.

Bueno y eso es todo, la próxima vez ya para en la api CreateToolhelp32Snapshot

7C8647B7	8BFF	MOV EDI,EDI	ntdll.7C920738
7C8647B9	55	PUSH EBP	
7C8647BA	8BEC	MOV EBP,ESP	
7C8647BC	83EC 0C	SUB ESP,0C	
7C8647BF	56	PUSH ESI	
7C8647C0	8B75 0C	MOV ESI,DWORD PTR SS:[EBP+C]	
7C8647C3	85F6	TEST ESI,ESI	
7C8647C5	75 07	JNZ SHORT kernel32.7C8647DE	

El stack

0012FFB8	00401151	CALL to CreateToolhelp32Snapshot from buggers3.0040114B	
0012FFBC	00000002	Flags = TH32CS_SNAPPROCESS	
0012FFC0	00000000	ProcessID = 0	
0012FFC4	7C816D4F	RETURN to kernel32.7C816D4F	
0012FFC8	7C920738	ntdll.7C920738	

Pues busquemos en el winapis32 que hace la dichosa api

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.

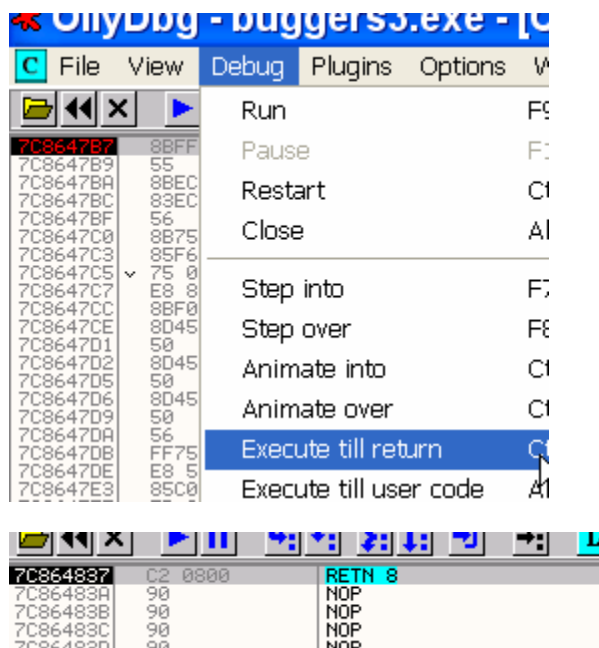
th32ProcessID

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.

Pues lo que hace esta api es tomar como una fotografía o instantánea (SNAPSHOT) de los procesos que están corriendo en la maquina, pero de esta foto, solo nos devuelve el handle o manejador de la misma ya que en los parámetros no hay ningún buffer donde guardara la lista de procesos, veamos lleguemos hasta el RET.



Y en EAX devuelve el handle

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.

Allí dice que devuelve la información del primer proceso que encuentra en el snapshot o foto que se tomo anteriormente y que los parámetros son el handle de la snapshot que en mi caso es 2C y la dirección donde guardara la info o buffer que en mi maquina es 403134.

```
0012FFB8 00401162 CALL to Process32First from buggers3.0040115C
0012FFBC 0000002C hSnapshot = 0000002C
0012FFC0 00403134 pProcessentry = buggers3.00403134
0012FFC4 7C816D4F RETURN to kernel32.7C816D4F
0012FFC8 7C920738 ntdll.7C920738
0012FFCC FFFFFFFF
0012FFD0 7FFDE000
```

Esta api solo devuelve la info sobre el primer proceso de la lista, para los siguientes se utiliza Process32Next que es la compañera de esta, en la tarea de leer los datos sobre los procesos.

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

Allí esta en el dump, el buffer donde guardara la info sobre el primer proceso, así que ejecutemos hasta el RET para que guarde allí la info.

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

Allí vemos el nombre del primer proceso de la lista que siempre es el proceso SYSTEM PROCESS continuemos con RUN.

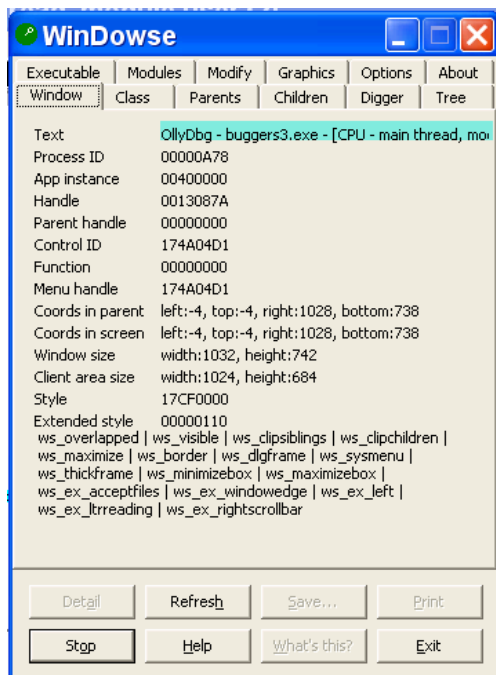
```
0012FFB8 0040116F CALL to FindWindowA from buggers3.00401169
0012FFBC 004030AE Class = "OllYdbg"
0012FFC0 00000000 Title = NULL
0012FFC4 7C816D4F RETURN to kernel32.7C816D4F
0012FFC8 7C920738 ntdll.7C920738
0012FFCC FFFFFFFF
0012FFD0 7FFDE000
0012FFD4 8054A938
```

Ajaja aquí hay otro truco, en este caso usa la api FindWindowA y le esta preguntando si la ventana superior que esta a la vista, tiene OLLYDBG como clase de la misma, aquí podría preguntar por el nombre de la ventana también, ambos datos se encuentran en los parámetros, pero en este caso, pregunta por la clase de la ventana principal que esta en uso, que obviamente es la del OLLYDBG y cuya clase es OLLYDBG.

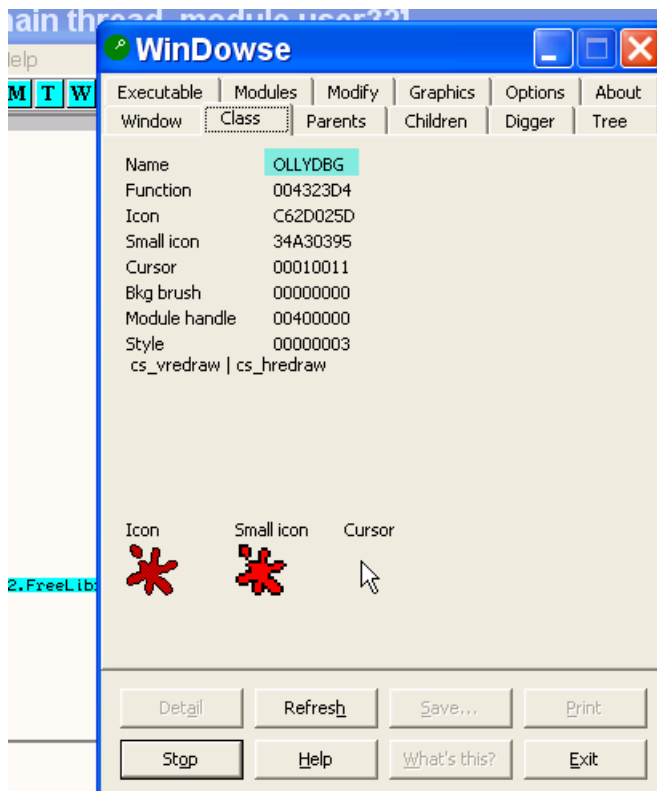
Como podemos saber eso, pues usemos una utilidad que se encuentra en mi http

<http://www.ricnar456.dyndns.org/HERRAMIENTAS/V-W-X-Y-Z/WindowseGREATIS5setup.exe>

Yo se que hay plugins de OLLYDBG que permiten hallar la class y otros datos de una ventana, pero la verdad lo mas completo que vi es este programa veamos, lo instalo y lo arranco.



Vemos que en la ventana WINDOW nos da el nombre de la ventana del OLLYDBG y en la ventana CLASS nos da la clase de la misma.



Que como vemos es OLLYDBG

Vemos que la api FindWindowA nos devuelve el handle de la ventana, con lo cual ya sabemos puede cerrarla, o hacer lo que quiera con ella.

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.

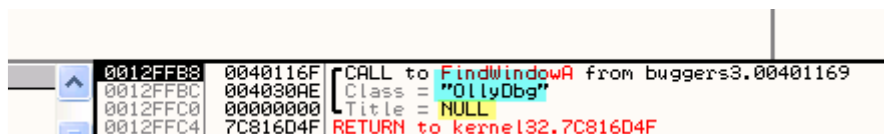
Return Values

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

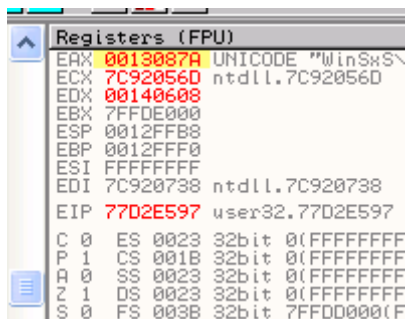
If the function fails, the return value is NULL. To get extended error information, call [GetLastError](#).

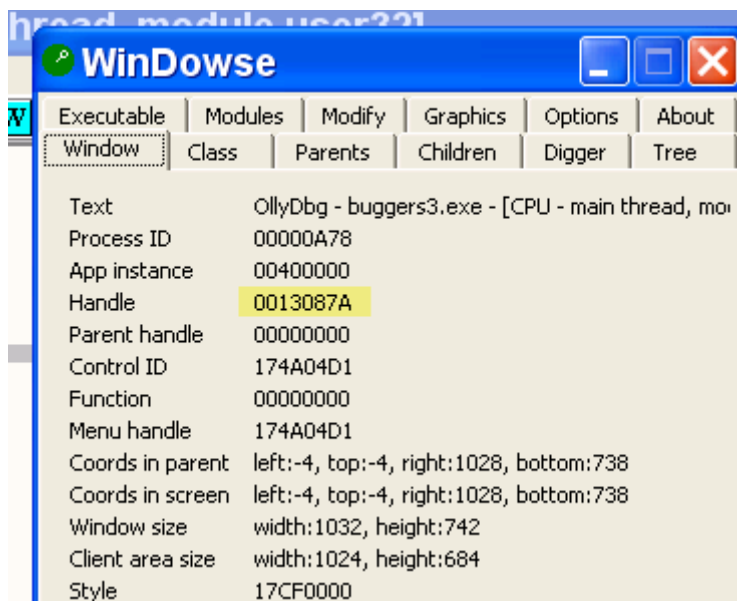
...

O sea no se necesita poner ambos datos el nombre y la clase, podemos buscar uno solo de los dos, y el otro se pone a cero como en nuestro caso.



Bueno lleguemos al RET de la api a ver si nos devuelve el handle de la ventana





Que por supuesto coincide con el que nos averigua el Windowse.

Bueno traceemos a ver que hace el programa con el handle de la ventana

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
0040119F	. 68 B6304000	PUSH buggers3.004030B6	ExitProcess
004011A4	. 68 58314000	PUSH buggers3.00403158	ASCII "OLLYDBG.EXE"
004011A9	. FF15 24314000	CALL DWORD PTR DS:[403124]	ASCII "[System Process]"
004011AF	. 0BC0	OR EAX,EAX	kernel32.lstrcpA
004011B1	. 75 2F	JNZ SHORT buggers3.004011E2	

O sea compara si es cero, ese seria el caso de que no hubiera una ventana con la clase OLLYDBG o sea en ese caso correría pues no hay OLLYDBG, ahora, al hallar un handle que es diferente de cero, eso significa que hay una ventana con el Class OLLYDBG y salta a ExitProcess.

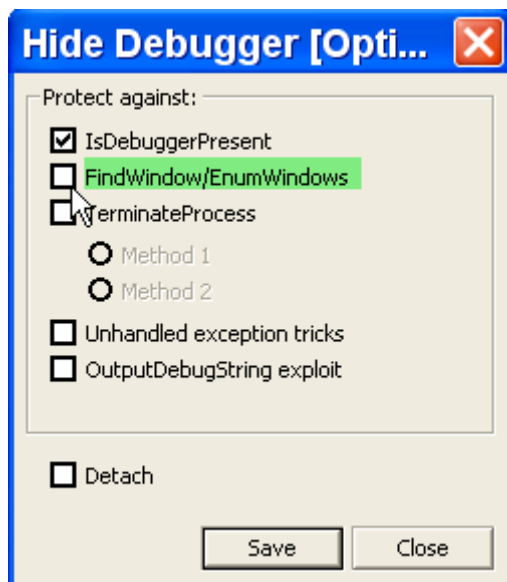
00401164	. 68 AE304000	PUSH buggers3.004030AE	ASCII "OllyDbg"
00401169	. FF15 28314000	CALL DWORD PTR DS:[403128]	user32.FindWindowA
0040116F	. 83F8 00	CMP EAX,0	
00401172	. 0BC0	OR EAX,EAX	
00401174	. 75 04	JNZ SHORT buggers3.0040117A	
00401176	. 7C 27	JL SHORT buggers3.0040119F	
00401178	. EB 25	JMP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

Al saltar allí va directo a la salida del programa sin haber mostrado aun nada.

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]	

O sea que el tema es que al volver de FindWindowA, EAX debe ser cero y no saltar.

Bueno el plugin HideDebugger 1.23f ahora que ya sabemos evitarlo a mano, también viene preparado, para esta detección, si vemos en las opciones del PLUGIN



Si ponemos la tilde en el segundo lugar vemos que nos protege contra la detección por medio de las apis FindWindow y EnumWindows que es la otra que detecta el nombre de la ventana, así que aprendimos a verlo a mano, a evitarlo, y a entender como funciona, ahora no le ponemos aun la tilde, porque deberíamos reiniciar el OLLYDBG para que haga efecto, lo haremos al terminar, por ahora hacemos que el salto que nos lleva a ExitProcess, no salte así continua el programa.

```

EIP 00401174 b
C 0 ES 0023 3
P 0 CS 001B 3
A 0 SS 0023 3
Z 0 DS 0023 3
S 0 FS 003B 3
T 0 GS 0000 N
D 0
O 0 LastErr E
EFL 00000202 (

```

Hago doble click en el flag Z lo cual lo pone a 1 y hace que el JNZ no salte.

```

EIP 00401174
C 0 ES 0020
P 0 CS 001E
A 0 SS 0020
Z 1 DS 0020
S 0 FS 003E
T 0 GS 0000
D 0
O 0 LastErr
EFL 00000242
ST0 empty _l

```

Y ahora el JNZ no salta

```

00401169 . FF15 28314000 CALL DWORD PTR DS:[403128]
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

```

Y llegamos al JMP que saltea el call a ExitProcess

```

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]
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>
0040119F . 68 B6304000 PUSH buggers3.004030B6
004011A4 . 68 58314000 PUSH buggers3.00403158
004011A9 . FF15 24314000 CALL DWORD PTR DS:[403124]
004011AF . 0BC0 OR EAX,EAX
004011B1 . 75 2F JNZ SHORT buggers3.004011E2
004011B3 . FF35 3C314000 PUSH DWORD PTR DS:[40313C]

```

kernel32.FreeLibrary

```

ExitCode = 0
ExitProcess
ASCII "OLLYDBG.EXE"
ASCII "[System Process"
kernel32.lstrcmpA

```

Bueno continuemos ya terminado el truco de FindWindowA ahora continuara con el de los nombres de los procesos demos RUN

```

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

```

Vemos que llama a Process32Next para ver el segundo proceso de la foto y guardara la info en 403134.

Hagamos execute till return y veamos lo que guardo

```

Return to 004011F3 (buggers3.004011F3)

```

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 00A....
0040314C	00 00 00 00 08 00 00 008....
00403154	00 00 00 00 53 79 73 74	...Syst...
0040315C	65 60 00 20 50 72 6F 63	em. Proc
00403164	65 73 73 50 00 00 00 00	essl....
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

Ahora el nombre es SYSTEM y su PID es 4 veamos la lista de procesos

NETFileServerEngine.exe	148	SYSTEM	00	18,644 KB
System	4	SYSTEM	00	216 KB
Proceso inactivo del sistema	0	SYSTEM	98	16 KB

Allí esta veamos que hace con cada proceso volvamos al programa traceando

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.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 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 E5FFFFFF	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	

Vemos que aquí llama a la api lstrcmpA con la cual compara la string SYSTEM que es el nombre del proceso, con el de la crackme buggers3.exe, vemos que esa será la salida del crackme, cuando llegue a encontrar el nombre de su mismo proceso, pues ira a un MessageBoxA que vemos debajo que nos dice NOT DEBUGGED, pero aun no llegamos allí, así que sigamos traceando

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 A0314000	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 08314000	CALL DWORD PTR DS:[403108]	kernel32.GetProcAddress

Al no ser iguales las strings, el resultado de la comparación es FFFFFFFF

Registers (FPU)	
EAX	FFFFFFFF
ECX	0000A6B8
EDX	0000000E
EBX	7FFDE000
ESP	0012FFC4
EBP	0012FFF0
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

Y por lo tanto al no ser cero, o sea iguales salta a 40119f sigamos allí

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	

Aquí vemos la parte caliente, compara el nombre del primer proceso con OLLYDBG.exe y si es igual, pues el resultado es cero y no salta, con lo cual ira a OpenProcess a ver el handle del mismo y luego a TerminateProcess para cerrarlo como vimos en la parte 20.

```
00401197 . 58          POP EAX
00401198 . 6A 00       PUSH 0
0040119A . E8 57000000 CALL <JMP.&kernel32.ExitProcess>
0040119F > 68 B6304000 PUSH buggers3.004030B6
004011A4 . 68 58314000 PUSH buggers3.00403158
004011A9 . FF15 24314000 CALL DWORD PTR DS:[403124]
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]
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]
004011D9 . 6A 00       PUSH 0
004011DB . E8 16000000 CALL <JMP.&kernel32.ExitProcess>
004011DE . 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]
004011F3 > EB 7B       JMP SHORT buggers3.00401270
004011FE . 7C         INT3

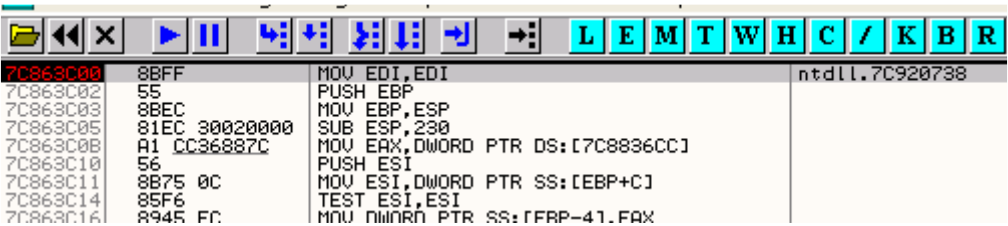
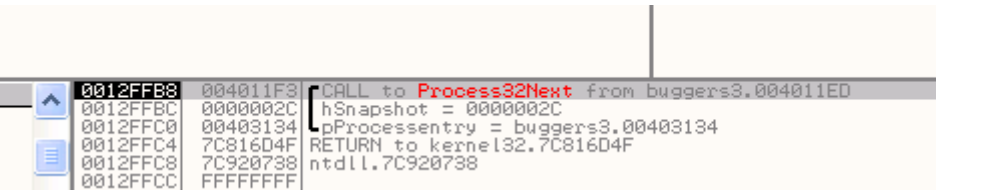
[ExitCode = 0
ExitProcess
ASCII "OLLYDBG.EXE"
ASCII "System"
kernel32.lstrcpA

kernel32.OpenProcess

kernel32.TerminateProcess
ExitCode = 0
ExitProcess

kernel32.Process32Next
```

Vemos que al no ser el primer proceso OLLYDBG.exe pues salta a Process32Next a buscar el segundo proceso lleguemos allí.

Pues guardara en el mismo lugar el nombre del segundo proceso hagamos execute till return

Address	Hex dump	ASCII
00403134	28 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	...B....
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

El segundo proceso es smss.exe y su PID es 026C si vemos en la lista de procesos

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

Allí esta el PID es 620 decimal o sea 026C en hexa

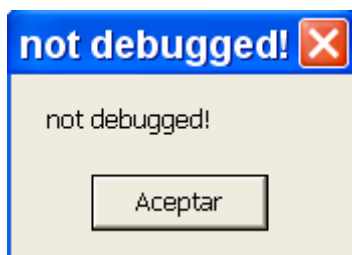
Bueno pues repetirá lo mismo con todos los procesos que hay en la maquina uno a uno los comparara con OLLYDBG.exe

00401198	. 6A 00	PUSH 0	[ExitCode = 0 ExitProcess ASCII "OLLYDBG.EXE" ASCII "smss.exe" kernel32.lstrcmpA
0040119A	. E8 57000000	CALL <JMP.&kernel32.ExitProcess>	
0040119F	> 68 B6304000	PUSH buggers3.004030B6	
004011A4	. 68 58314000	PUSH buggers3.00403158	
004011A9	. FF15 24314000	CALL DWORD PTR DS:[403124]	
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]	
004011C6	. A3 64324000	MOV DWORD PTR DS:[403264],EAX	kernel32.TerminateProcess ExitCode = 0 ExitProcess
004011CB	. 6A 00	PUSH 0	
004011CD	. FF35 64324000	PUSH DWORD PTR DS:[403264]	
004011D3	. FF15 20314000	CALL DWORD PTR DS:[403120]	
004011D9	. 6A 00	PUSH 0	
004011DB	. E8 16000000	CALL <JMP.&kernel32.ExitProcess>	
004011E0	. EB 11	JMP SHORT buggers3.004011F3	
004011E2	> 68 34314000	PUSH buggers3.00403134	
004011E7	. FF35 5C324000	PUSH DWORD PTR DS:[40325C]	

Y siempre el resultado de la comparación nos llevara al salto condicional de 4011b1 el cual cuando halle el proceso OLLYDBG.exe no saltara y nos cerrara el OLLYDBG, así que podemos cambiarlo por un JMP, lo cual evitara que se cierre.

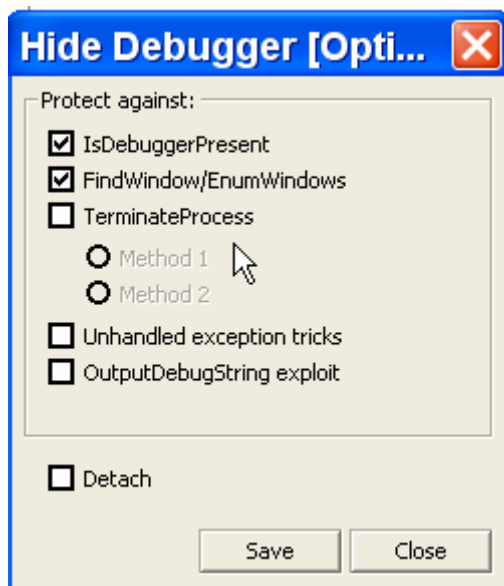
00401198	. 6A 00	PUSH 0	[ExitCode = 0 ExitProcess ASCII "OLLYDBG.EXE" ASCII "smss.exe" kernel32.lstrcmpA
0040119A	. E8 57000000	CALL <JMP.&kernel32.ExitProcess>	
0040119F	> 68 B6304000	PUSH buggers3.004030B6	
004011A4	. 68 58314000	PUSH buggers3.00403158	
004011A9	. FF15 24314000	CALL DWORD PTR DS:[403124]	
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]	
004011C6	. A3 64324000	MOV DWORD PTR DS:[403264],EAX	kernel32.TerminateProcess ExitCode = 0 ExitProcess
004011CB	. 6A 00	PUSH 0	
004011CD	. FF35 64324000	PUSH DWORD PTR DS:[403264]	
004011D3	. FF15 20314000	CALL DWORD PTR DS:[403120]	
004011D9	. 6A 00	PUSH 0	
004011DB	. E8 16000000	CALL <JMP.&kernel32.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

Ahora si desactivamos todos los BP y damos RUN



Con lo cual la protección ha sido vencida, ahora ya sabemos que con el plugin HideDebugger nos ocultara la ventana del OLLYDBG de la detección de FindWindowA y usando el OLLYDBG renombrado como PIRULO.exe no detectara ningún proceso llamado OLLYDBG, por lo cual allí debería correr sin problemas, veamos.

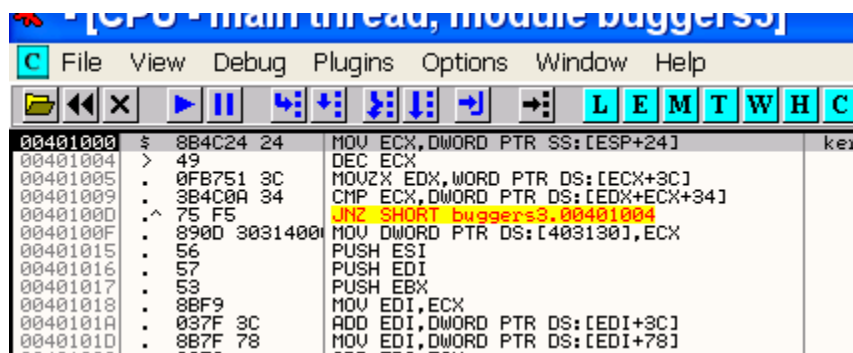
Abramos el PIRULO.exe



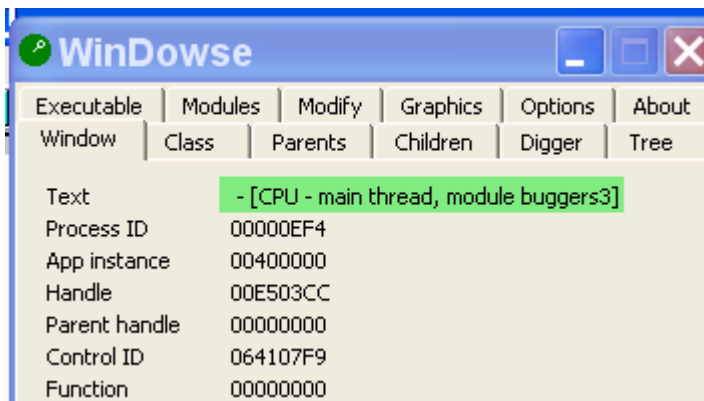
Coloquemosle la tilde en la opción para ocultar de la api FindWindowA y apretemos SAVE



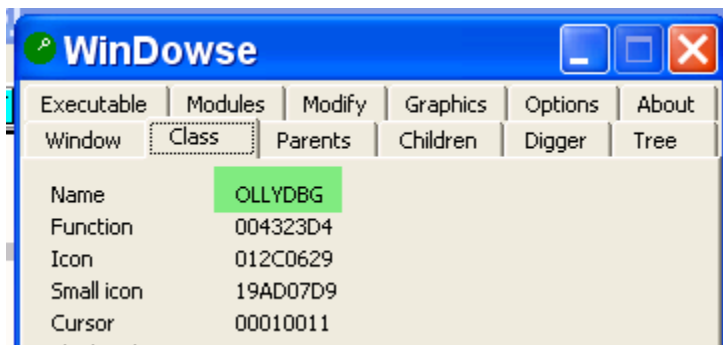
Pues reiniciémoslo cerrándolo completamente y volviéndolo a abrir



Y cargo el buggers3 pero antes me voy a sacar una duda, voy a ver con el Windowse, que cambios realizo el plugin en la ventana del OLLY para que no sea detectada



Vemos que ahora no aparece el nombre OLLYDBG en el título de la ventana y la clase ?

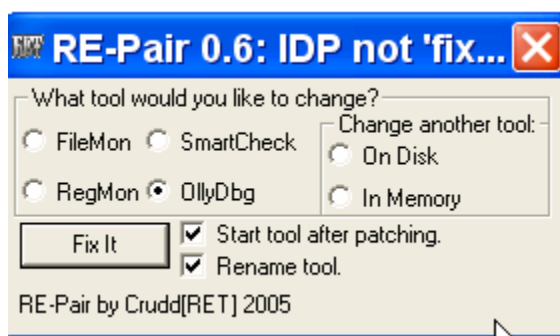


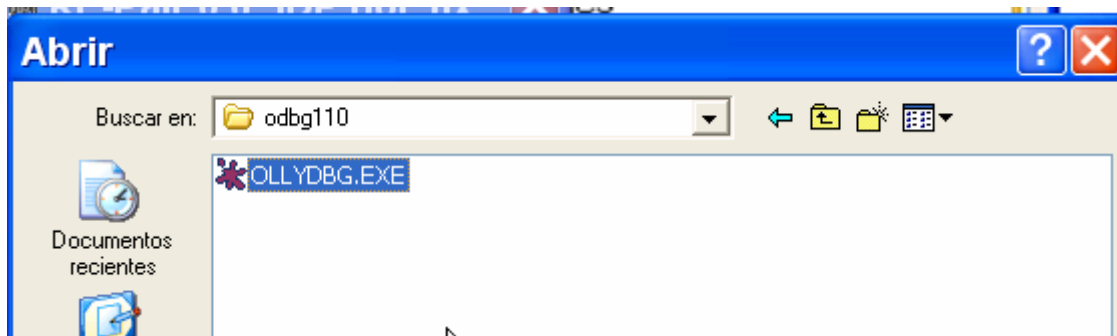
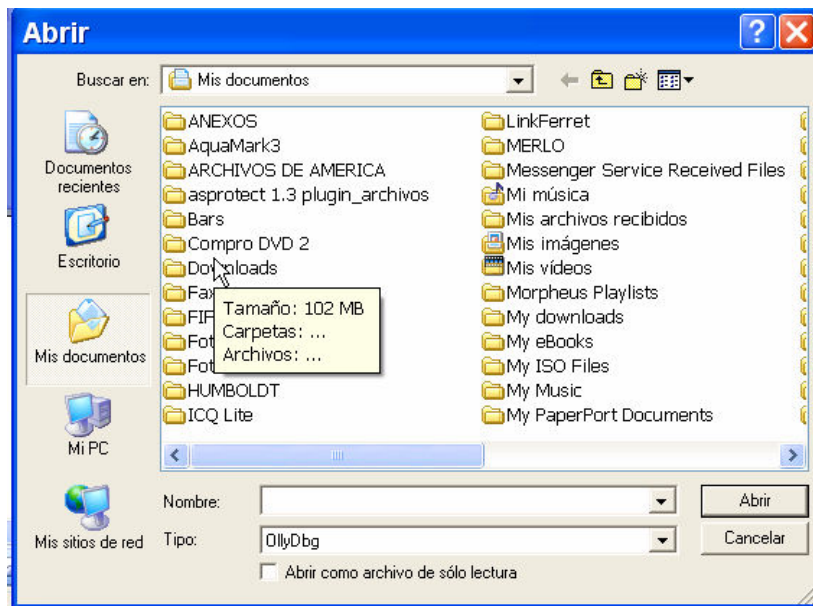
Vemos que por el lado de la clase no nos protege para nada debemos hallar otra cosa.

La herramienta que no es un plugin que nos ayuda con esto es el Repair 0.6, es un parcheador de OLLYDBG que se encuentra en mi http aquí

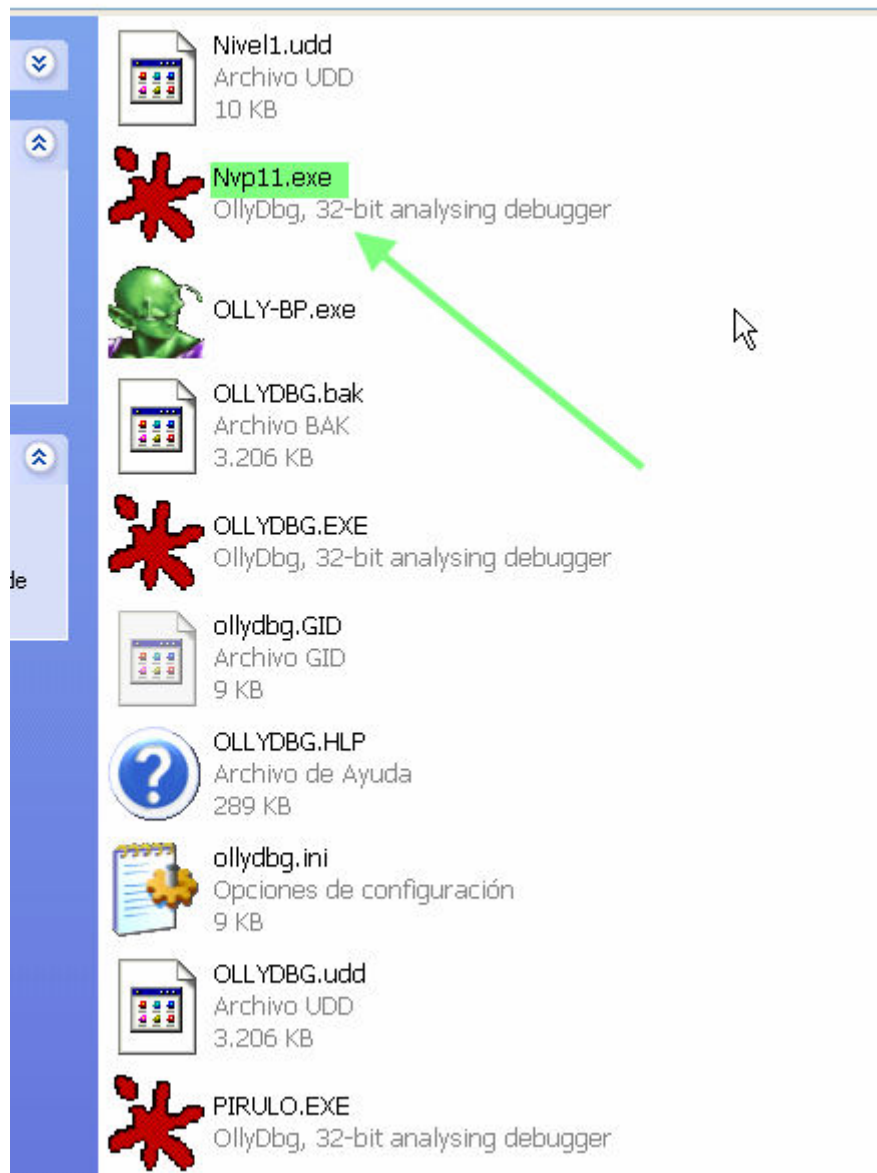
<http://www.ricnar456.dyndns.org/HERRAMIENTAS/Q-R-S-T-U/repair0.6.zip>

Si lo bajamos y cerramos el OLLYDBG y arrancamos el parcheador

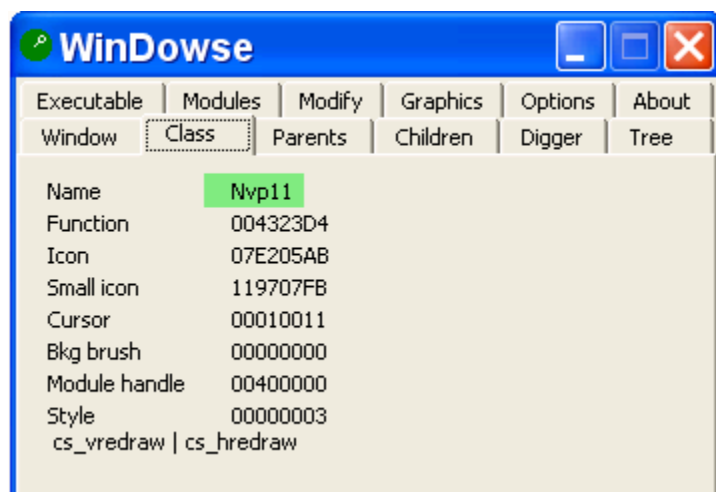




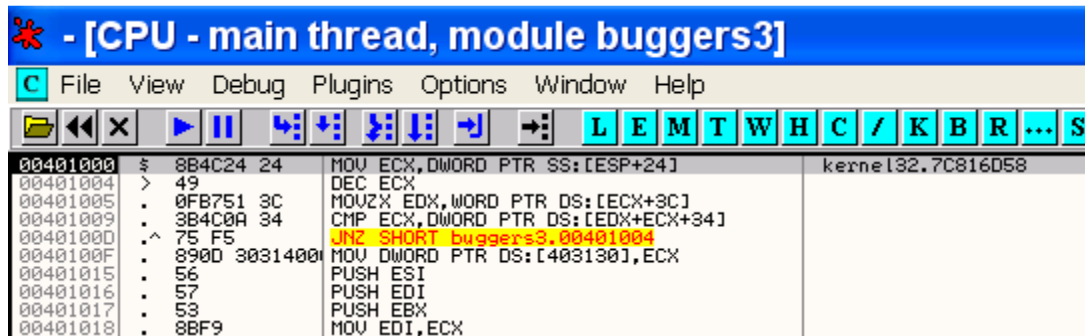
Bueno así que tenemos un tercer OLLYDBG parcheado que se llama NVP11.exe veamos en la carpeta del OLLYDBG donde esta



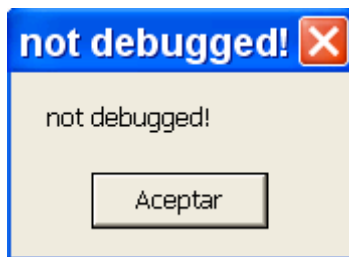
Allí esta, abrámoslo y veamos que CLASS tiene el mismo



Vemos que tiene como CLASS Nvp11 y por supuesto el nombre del proceso también es ese, por lo cual el buggers3 debería correr aquí perfectamente sin cambiar nada, probemos



Doy Run y



Quiere decir que nuestro OLLYDBG parcheado, es cada día menos detectado, al menos ya no se lo puede detectar ni por el nombre del proceso OLLYDBG ni por el título ni la clase de la ventana, jeje en la parte 22 seguiremos fortificando nuestro OLLYDBG y aprendiendo como funcionan mas detecciones antidebugger, como entenderlas, arreglarlas a mano y al final con algún plugin , para evitar trabajar de mas jeje.

Hasta la parte 22

Ricardo Narvaja

27 de diciembre de 2005