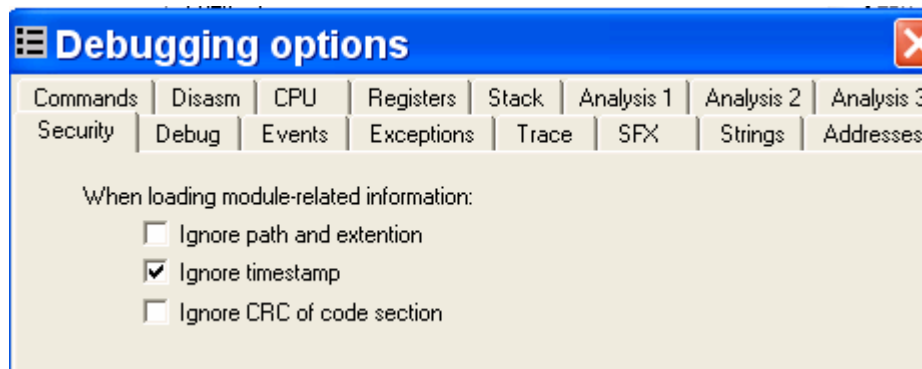


INTRODUCCION AL CRACKING CON OLLYDBG PARTE 20

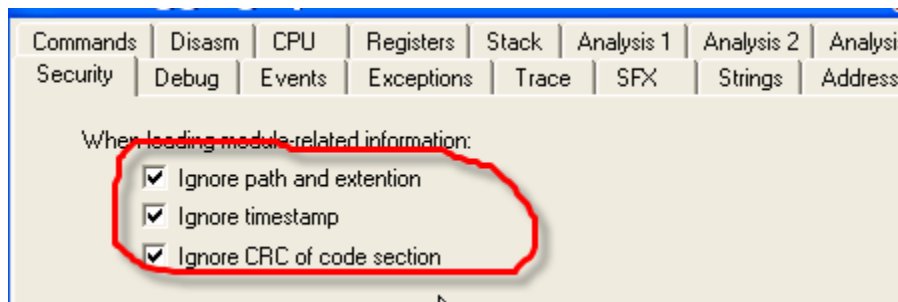
El siguiente truco ANTIOLLYDBG que veremos en esta oportunidad es la detección del mismo, por el nombre del proceso del OLLYDBG, pero antes de todo les diré una configuración del OLLYDBG que aun no hemos tocado y que es muy importante en este caso.

Si vamos a DEBUGGING OPTIONS-SECURITY



Allí lo mejor es poner las tres tildes y que cargue siempre la información, verán que con las tres tildes puestas cuando pongan BPX en una api y reinicien el OLLYDBG, el BPX continuara puesto, lo cual evita tener que repetir el proceso de ponerlos uno a uno cada vez que reiniciamos.

Estrictamente no conozco el funcionamiento interno completo del OLLYDBG pero en la practica lo que sucede es eso con las tres tildes, los BPX en las apis se mantienen luego de reiniciarlo.



Así esta mejor para mi gusto, es menos molesto, ahora si, comenzamos con el tema de antidebugging por el nombre del proceso.

Deteccion del OLLYDBG por el nombre del proceso

Cuando corremos el OLLYDBG si miramos la lista de procesos con CTRL + ALT + SUPR

Process Name	PID	Username	Session ID	Private Bytes
svchost.exe	1448	SYSTEM	00	56 KB
OLLYDBG.EXE	1412	Ricardo	00	344 KB
ipfw.exe	1324	SYSTEM	02	11.540 KB
avaemc.exe	1212	SYSTEM	00	1.128 KB

Vemos que el nombre del proceso esta allí muy claro en la lista de los mismos, que le impide a un programa revisar todos los procesos y si encuentra alguno que se llame OLLYDBG, cerrarlo, pues nada, jeje.

Usaremos un crackme que no resolveremos por ahora porque es un nivel levemente superior al actual, así que lo veremos mas adelante, pero estudiaremos en el la técnica de cómo detectan los programas a OLLYDBG por el nombre del proceso, como evitarlo manualmente y como evitarlo definitivamente jeje.

El crackme que adjunto es el DAXXOR el cual si dejamos un OLLYDBG corriendo vacío y corremos fuera de OLLY el famoso DAXXOR, veremos que el crackme corre pero cierra el OLLYDBG, lo mismo que si lo corremos en OLLYDBG, también lo cierra y por supuesto se acaba todo.

Estudiemus como hace esto, abrámoslo en OLLYDBG.

Address	Disassembly	Comment
004012FC	JMP SHORT DaXXoR.0040130E	
004012FE	DB 66	CHAR 'f'
004012FF	DB 62	CHAR 'b'
00401300	DB 3A	CHAR 'i'
00401301	DB 43	CHAR 'c'
00401302	DB 2B	CHAR '+'
00401303	DB 2B	CHAR '+'
00401304	DB 48	CHAR 'H'
00401305	DB 4F	CHAR 'O'
00401306	DB 4F	CHAR 'O'
00401307	DB 4B	CHAR 'K'
00401308	DB 90	
00401309	DB E9	
0040130A	DD OFFSET DaXXoR.CPPdebugHook	
0040130E	MOV EAX, DWORD PTR DS:[46608B]	
00401313	SHL EAX, 2	
00401316	MOV DWORD PTR DS:[46608F], EAX	

Allí esta en el ENTRY POINT, y veamos las apis que utiliza.

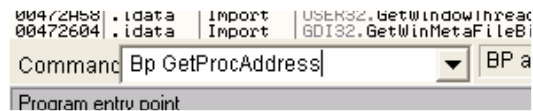
Address	Segment	Type	Name
00472C5C	.idata	Import	OLEAUT32.#8
00472C48	.idata	Import	OLEAUT32.#84
00472C50	.idata	Import	OLEAUT32.#9
00472C38	.idata	Import	OLEAUT32.#94
00402930	.text	Export	@Unit2@Finalize
00402920	.text	Export	@Unit2@Initialize
00472908	.idata	Import	USER32.ActivateKeyboardLayout
0047290C	.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
0047259C	.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
0047294C	.idata	Import	USER32.DefFrameProcA
00472950	.idata	Import	USER32.DefMDIChildProcA
00472244	.idata	Import	USER32.DefWindowProcA
00472248	.idata	Import	KERNEL32.DeleteCriticalSection
004725A8	.idata	Import	GDI32.DeleteDC

Bueno hay unas cuantas, pero aquí hay otra protección agregada y es que el crackme no tiene cargadas en el inicio las apis que usara para detectar al OLLYDBG, y las cargara a medida que corra, de paso explicaremos también este método de protección que en si, no te permite ver todas las apis que usara, en la lista de NAMES, pero por otra lado tiene como contrapartida que el que se da cuenta del truco y descubre las apis que va cargando el programa a medida que corre, sabrá a ciencia cierta que esas apis son las IMPORTANTES, pues por algo el programa las oculta para que no aparezcan en la lista.

Esto casi siempre es evidente cuando un programa que no esta empacado como en este caso, hace uso de la api GetProcAddress

004725E8	.idata	Import	GDI32.GetPaletteEntries
00472A1C	.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
00472A28	.idata	Import	USER32.GetScrollInfo

GetProcAddress se utiliza, para que el programa cargue nuevas apis, que no están en la lista para poder usarlas, ya veremos un uso intensivo y mas detallado de esta api en el capítulo de desempacado, pero por ahora pongamos un BP en dicha api.



Y demos RUN

0012FF60	0045C920	CALL to GetProcAddress from DaXXoR.0045C925
0012FF64	00400000	hModule = 00400000 (DaXXoR)
0012FF68	00467BA8	ProcNameOrOrdinal = "__CPPdebugHook"
0012FF6C	004607C1	RETURN to DaXXoR.004607C1
0012FF70	00000000	
0012FF74	00466034	DaXXoR.00466034
0012FF78	7FFDF000	
0012FF7C	0046C040	DaXXoR.0046C040

Cada vez que para vemos que esta pidiendo por medio de la api GetProcAddress, la dirección en nuestra maquina de una determinada api la primera que pide es en este caso, __CPPdebugHook que no pertenece al truco que estamos estudiando por lo cual damos RUN nuevamente.

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

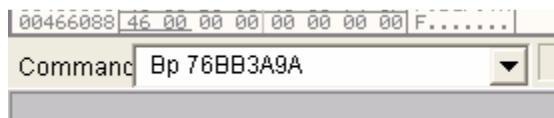
Así vamos pasando con F9 hasta que encontremos apis relacionadas con el truco, aquí vemos la api EnumProcesses que es usada, así que lo que hacemos es llegar hasta el RET de GetProcAddress la cual nos devuelve en EAX la dirección de la api solicitada en nuestra maquina y allí le ponemos un BPX, veamos, lleguemos al RET con EXECUTE TILL RETURN.

Registers (FPU)			
EAX	76BB3A9A		
ECX	7C929AEB	ntdll.7C929AEB	
EDX	7C98C0D8	ntdll.7C98C0D8	
EBX	00972460		
ESP	0012E7F0		
EBP	0012FDE8		
ESI	00971024		
EDI	004664A0	DaXXoR.004664A0	
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 7FDE000(FFF)	
T 0	GS 0000	NULL	
D 0			

Allí en EAX devuelve la dirección en mi maquina de la api solicitada, en mi caso es 76BB3A9A en sus maquinas puede variar.



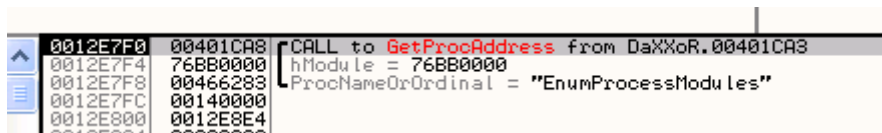
Por otro lado veo que OLLYDBG no identifica la api que no esta en la lista de NAMES, por lo cual no se puede poner BP directos al nombre de la api, si no que hay que poner el BP en la dirección.



Ahora si lo toma

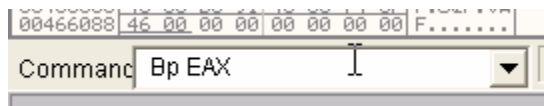


Ya tenemos puesto el BP en la api sospechosa, sigamos corriendo OLLYDBG, para ver si carga mas apis.

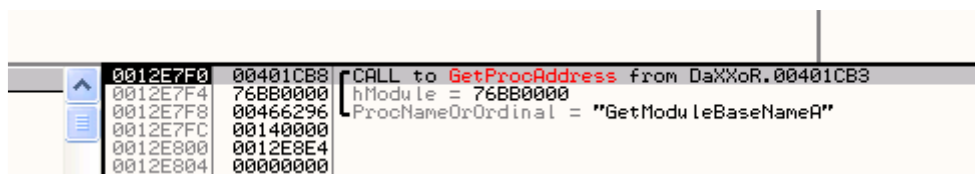


Hmm enumera los módulos de un proceso, hmm repito el procedimiento anterior llego al RET y le pongo un BPX a la dirección que me muestra EAX.

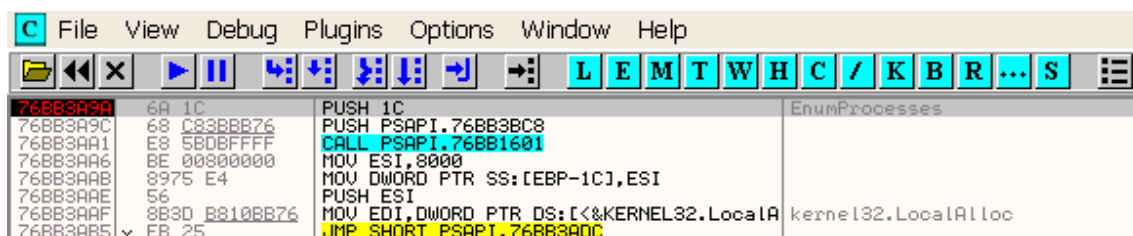
Para no tipear mucho si estoy en el RET directamente pongo



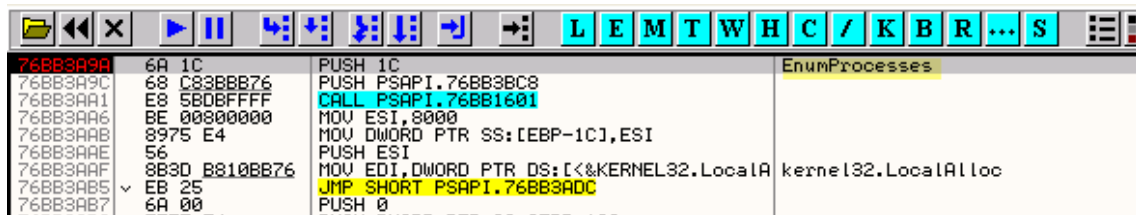
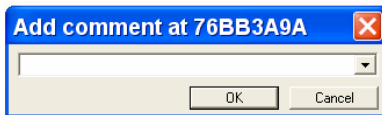
Que me servirá para colocarle BPX a todas las apis cuando este en el RET .



Otra sospechosa le pongo BP de la misma forma que a las anteriores, y doy RUN y para en EnumProcesses

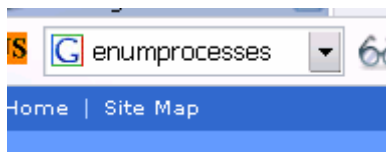


El comentario a la derecha con el nombre de la api se lo agregue yo, haciendo doble click en esa zona nos permite agregar un comentario.



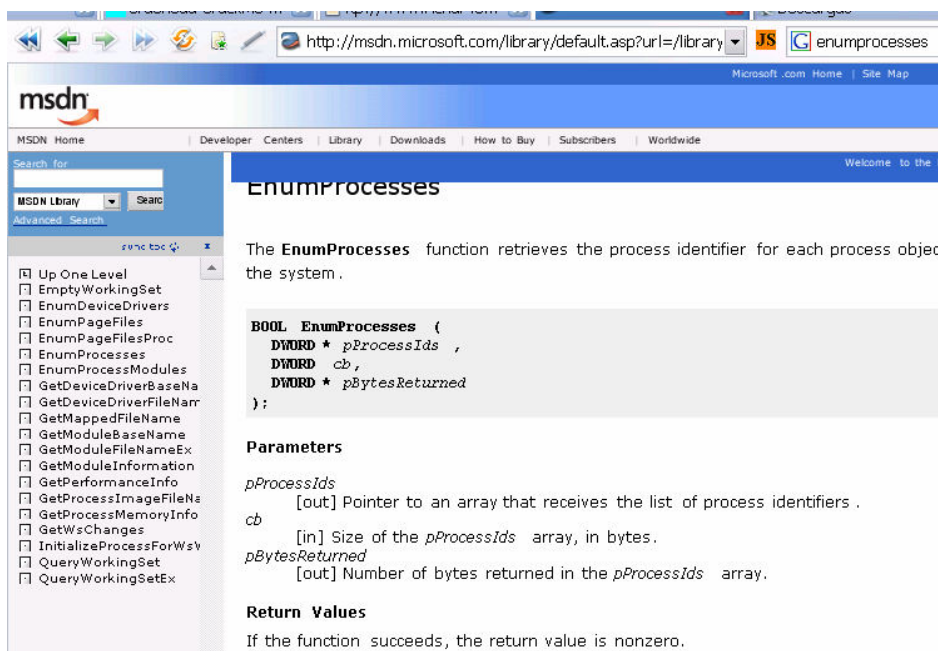
Esto lo realizo en todas las apis que el programa va cargando y veo sospechosas cuando le pongo un BPX cosa de que cuando pare, sepa que api era, pues el OLLYDBG no me aclarara nada al no ser una api de las de la lista.

Si busco en el WINAPIS32 esta api no figura, por lo tanto busco en GOOGLE



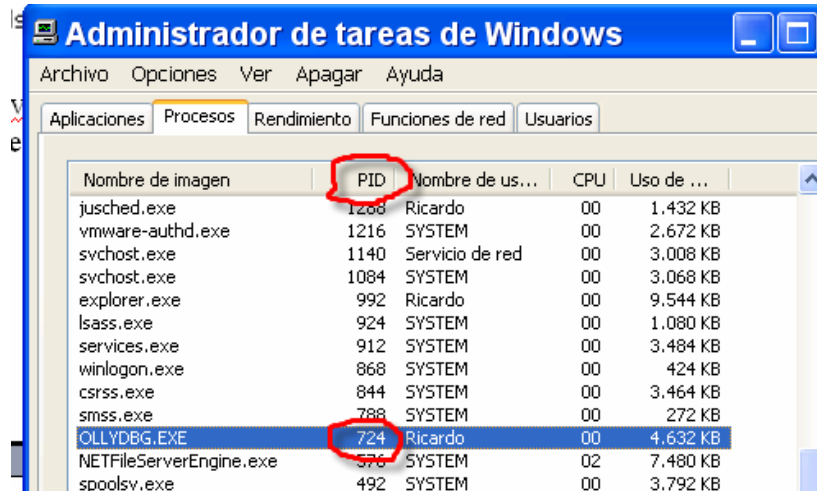
Y en la pagina de Microsoft normalmente se encuentran, es este caso la pagina es

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/perfmon/base/enumprocesses.asp>

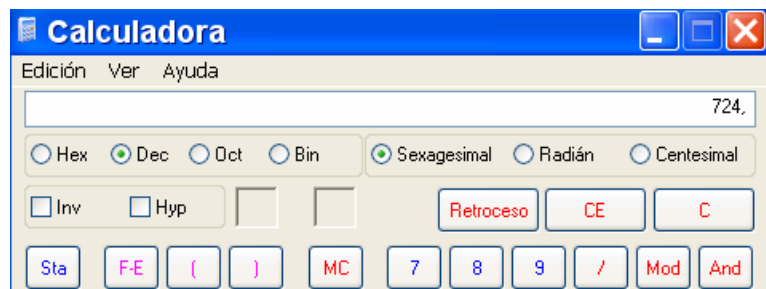


Bueno aquí dice que la susodicha api, nos devolverá el PROCESS IDENTIFIER o PID de cada proceso que esta corriendo, pues bien veamos antes que es el PID ese, jeje.

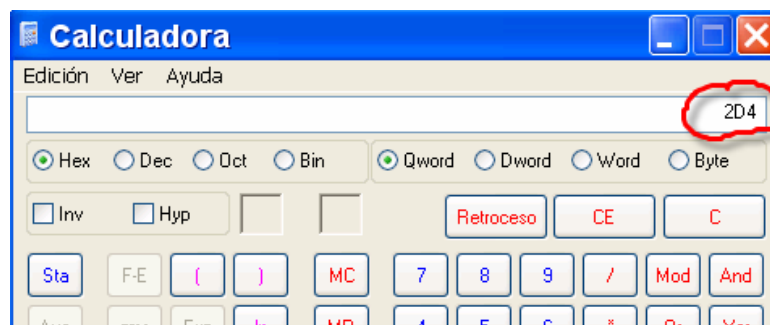
Cada proceso que corre esta identificado con un número que varia cada vez que se arranca un proceso, si vemos en la lista de procesos



Vemos que el OLLYDBG en este caso tiene un PID de 724 decimal ya que esta utilidad trabaja con números decimales, pero bueno si queremos saber el PID del OLLYDBG en hexa con la calculadora de Windows.

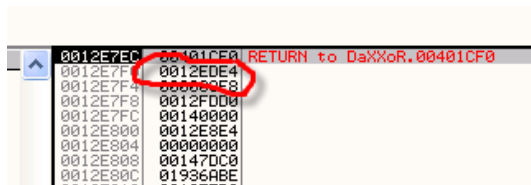


Apreto el boton hex para pasar a HEXA



2D4 será el PID del OLLYDBG, pueden verificar que si lo cierran al OLLYDBG y lo vuelven a abrir el PID variara, pues cada proceso al reiniciarse recibirá otro PID.

Tampoco tendremos la suerte en este caso de que OLLYDBG nos muestre los parámetros de la api pues para OLLYDBG no existe la misma.



Sabemos por la pagina de Microsoft que los tres parámetros son estos

pProcessIds

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

cb

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

pBytesReturned

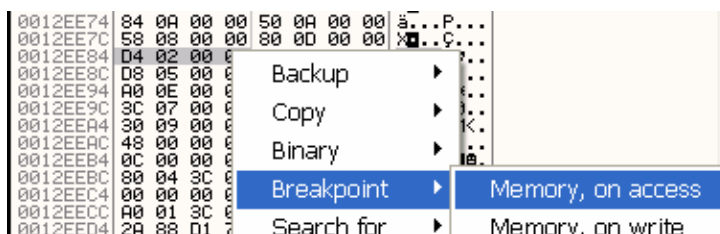
[out] Number of bytes returned in the *pProcessIds* array.

O sea que en 12eDe4 guardara la lista de PIDs de todos los procesos que corren en mi maquina, hagamos execute till return para llegar al ret de la api y ver en el dump si los guarda alli.

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	...'. ..
0012EE14	EC 01 00 00 E0 03 00 00	y...'. ..
0012EE1C	08 05 00 00 10 05 00 00	...'. ..
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...T... ..
0012EE3C	5C 05 00 00 6C 05 00 00	\...l... ..
0012EE44	B0 01 00 00 D4 07 00 00	...E... ..
0012EE4C	EC 07 00 00 C8 00 00 00	y...'. ..
0012EE54	40 02 00 00 D0 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 D8 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	D4 02 00 00 7C 0B 00 00	...i... ..
0012EE8C	D8 05 00 00 FC 0D 00 00	i...'. ..
0012EE94	A0 0E 00 00 E0 0F 00 00	...'. ..
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	...'. ..
0012EEBC	80 04 3C 00 00 4D 3C 00	...M<...
0012EEC4	00 00 00 00 67 04 D4 77	...g... ..
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	...\$. ..
0012EFF4	3A 0A 0A 0A 7F 0A 0A 0A	...\$. ..

Ahí esta la lista de PIDs y esta el de mi OLLYDBG sniff, sniff

Pongo un BPM ON ACCESS allí para ver cuando lo usa, veamos



Ahora si doy RUN

00401D36	89C4 08	ADD ESP,8	
00401D39	8B4D E4	MOV ECX,DWORD PTR SS:[EBP-1C]	
00401D3C	FFB48D FCEFF	PUSH DWORD PTR SS:[EBP+ECX*4-1004]	
00401D43	6A 00	PUSH 0	
00401D45	68 10040000	PUSH 410	
00401D4A	E8 A9300600	CALL <JMP.&KERNEL32.OpenProcess>	
00401D4F	8945 DC	MOV DWORD PTR SS:[EBP-24],EAX	
00401D52	837D DC 00	CMP DWORD PTR SS:[EBP-24],0	

ProcessId = 2d4
 Inheritable = FALSE
 Access = VM_READ|QUERY_INFORMATION
 OpenProcess

Vemos que para allí y va a usar la api OpenProcess que verifica si un proceso esta corriendo, y si esta corriendo te devuelve su handle o manejador.

Que diferencia hay entre el PID y el handle, muy sencillo, el PID es un identificador genérico, en toda tu maquina, en cualquier proceso el PID del OLLYDBG será el mismo mientras no se reinicie, en mi caso será 2d4, ahora el handle, como su nombre lo indica es un manejador, o sea que es un numero que te devuelve el sistema para que tu programa pueda manejar ese proceso, y el numero puede variar para cada programa, es como una solicitud para controlarlo, si no la pedís no tendrás el numerito y no lo podrás controlar, si lo pedís el sistema te devolverá el manejador y lo podrás manejar y hacerle guarradas jeje.

Veamos mas detalladamente la definición de OpenProcess en el WinApis32, la misma tiene muchos parámetros pero lo que nos interesa es esto.

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](#).

~ .

O sea devolverá el manejador del proceso que esta corriendo, en resumidas cuentas es lo que el programa quiere saber en este caso.

Traceemos con f8 hasta pasar la api

Registers (FPU)	
EAX	00000058
ECX	0012E7AC
EDX	7C91EB94 ntdll.K
EBX	00972460
ESP	0012E7FC
EBP	0012FDE8
ESI	00971D24
EDI	004664A0 DaXXoR.
EIP	00401D4F DaXXoR.
C 0	ES 0023 32bit 0
P 1	CS 001B 32bit 0
A 0	SS 0023 32bit 0

Y en EAX devuelve el handle o manejador del OLLYDBG que en mi caso es 58.

El mismo OLLYDBG nos muestra los HANDLES con los cuales esta trabajando el programa en la ventana H.

Handle	Type	Refs	Process	T	Info	Name
00000028	Desktop	3753.	000F01FF			\Default
00000008	Directory	82.	00000003			\KnownDlls
00000014	Directory	50.	000F000F			\Windows
00000034	Directory	478.	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			cs\Documents and Settings\Ricardo\Escritorio\DaXXoR_
00000010	Key	2.	000F003F			HKEY_LOCAL_MACHINE
00000038	Key	2.	000F003F			HKEY_CURRENT_USER
00000004	KeyedEvent	48.	000F0003			\KernelObjects\CritSecOutOfMemoryEvent
00000018	Port	3.	001F0001			
0000004C	Port	2.	001F0001			
00000058	Process	65.	00004110			
0000001C	Section	47.	000F001F			
00000050	Section	27.	00000004			
00000024	WindowStation	88.	000F037F			\Windows\WindowStations\WinSta0
0000002C	WindowStation	88.	000F037F			\Windows\WindowStations\WinSta0

Vemos que allí aparece el 58 y el TYPE o tipo es PROCESS o PROCESO, así que el programa maneja el handle 58 que pertenece a un proceso en este caso al OLLYDBG.

Si otro proceso usara EnumProcess para hallar el PID en este momento seria el mismo 2d4 mientras no se termine el proceso OLLYDBG, ahora si pide al sistema un handle o manejador para dicho proceso, será cualquier otro numero pues, los handles son particulares de cada proceso.

En este caso el peligro para nuestro OLLYDBG es que el programa ya tiene un manejador, con ello puede hacer lo que quiere, lo que si aun no sabe que pertenece a OLLYDBG solo que es un proceso y que esta corriendo, ahora debe verificar el nombre del mismo para determinar si este proceso es OLLYDBG, obviamente esto lo realiza con cada PID que obtiene de la lista de procesos, nosotros saltamos todos y llegamos hasta cuando trabaja con el OLLYDBG al haber puesto un BPM ON ACCESS en su PID.

Continuemos traceando con f8

00401D5E	8B5B CB	LEA EDI,DWORD PTR SS:[EBP-74]	
00401D61	52	PUSH EDI	
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 DaXxor.00401D84	
00401D6C	68 04010000	PUSH 104	
00401D71	8D8D F8EEFF	LEA ECX,DWORD PTR SS:[EBP-1108]	

Vemos que allí llega a la otra api que nos quiso ocultar en este caso EnumProcessModules, en la misma pagina de MICROSOFT vemos el detalle de esta api.

Search for

MSDN Library

Search

Advanced Search

func to

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.

O sea que ahora va a buscar de este proceso que esta investigando, la lista de handles de los módulos que usa, lleguemos con f7 hasta la api.

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	53	PUSH EBX	

En el stack vemos los parámetros

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

Según nuestro amigo HILL los handles de cada modulo se guardaran allí, vemos justo arriba el 58 perteneciente al handle del proceso OLLYDBG que es el parámetro superior.

Aquí hay una cosita que aclarar, cuando pedimos handle de los módulos el sistema nos devuelve la dirección base o donde comienza dicho proceso en la memoria, en este caso nos devuelve 400000 ya que el proceso OLLYDBG comienza allí.

Address	Hex dump	ASCII
0012FDA8	00 00 40 00 01 00 00 00	..@.0...
0012FDB0	0B 00 00 00 0B 00 00 00	8...8...
0012FDB8	00 00 BB 76 F0 FD 12 00	..UV-2+
0012FDC0	67 04 04 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*...
0012FD08	31 00 00 00 FA 03 00 00	1...b...

Ahora continúo traceando y veo que llega a la tercera api oculta

00401D71	808D F8EEFF	LEA ECX,DWORD PTR SS:[EBP-1108]	
00401D77	51	PUSH ECX	
00401D78	FF75 C0	PUSH DWORD PTR SS:[EBP-40]	
00401D7B	FF75 DC	PUSH DWORD PTR SS:[EBP-24]	
00401D7E	FF55 B0	CALL DWORD PTR SS:[EBP-50]	PSAPI.GetModuleBaseNameA
00401D81	8945 C4	MOV DWORD PTR SS:[EBP-3C],EAX	
00401D84	EC7E DC	POP DWORD PTR SS:[EBP-34]	

GetModuleBaseNameA

GetModuleBaseName

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

```
DWORD GetModuleBaseName (
    HANDLE hProcess,
    HMODULE hModule,
    LPTSTR 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](#).

O sea que con esto quiere hallar el nombre del modulo, ya que esta api, en el parámetro **lpBaseName**, abre un buffer para guardar el nombre del modulo que corresponde a esa base que averiguo antes, veamos lleguemos hasta la api.

En el stack los parámetros son:

El 58 que es el handle del OLLYDBG, el 400000 que es la base del modulo principal, y el buffer estará guardado en 12ECE0, así que veamos esa zona en el dump.

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	...ÀàÇ!
0012ED08	00 00 40 00 28 65 47 00	..@.(eG.
0012ED10	28 ED 12 00 A5 A0 80 7C	(Y+.NàÇ!

Jeje ya tiene ahora que llega hasta el RET el nombre del proceso y realiza esto con cada uno de los procesos que corren en tu máquina, pues ahora comparará el nombre a ver si es OLLYDBG.exe si en este caso hubiera sido OPERA.EXE por ejemplo, pues lo dejará tranquilo seguramente jeje.

Ahora llega a CloseHandle donde cierra el manejador o sea que el 58 desaparecerá de la lista de handles.

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_
00000010	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

Pues si, por ahora no podrá hacerle trastadas pues no tiene el handle abierto, pero no nos confiemos sigamos adelante.

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	

Vemos que llega a un CALL entremos a el con F7

0045DA00	. 33C0	XOR EAX,EAX	
0045DA0F	. 8A03	MOV AL,BYTE PTR DS:[EBX]	
0045DA01	. 50	PUSH EAX	
0045DA02	. E8 0D000000	CALL DaXXoR.0045DAE4	[Arg1 DaXXoR.0045DAE4
0045DA07	. 59	POP ECX	
0045DA08	. 8B03	MOV BYTE PTR DS:[EBX],AL	
0045DA0A	. 84C0	TEST AL,AL	
0045DA0C	. 75 EE	JNZ SHORT DaXXoR.0045DAC0	
0045DB17	. C3	RETN	
Stack DS:[0012ECE0]=4F ('O')			
AL=00			

Allí lee la primera letra de OLLYDBG (4F) y la pone en el stack con PUSH y entra a otro CALL entremos también.

Vemos que en este call no realiza nada demasiado importante salimos y llegamos al segundo.

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]	

Entramos en el

0045BA98	. 8B4C24 04	MOV ECX,DWORD PTR SS:[ESP+4]	
0045BA9C	. 8B5424 08	MOV EDX,DWORD PTR SS:[ESP+8]	
0045BA90	. 53	PUSH EBX	
0045BA91	. 33C0	XOR EAX,EAX	
0045BA93	. 33DB	XOR EBX,EBX	
0045BA95	. 8A01	MOV AL,BYTE PTR DS:[ECX]	
0045BA97	. 8A1A	MOV BL,BYTE PTR DS:[EDX]	
0045BA99	. 2BC3	SUB EAX,EBX	
0045BA9B	. 75 34	JNZ SHORT DaXXoR.0045BAE1	
0045BA9D	. 84DB	TEST BL,BL	
0045BA9F	. 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	
0045BABB	. 84DB	TEST BL,BL	
0045BABD	. 74 22	JE SHORT DaXXoR.0045BAE1	
0045BABF	. 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	
0045BAD9	. 83C2 04	ADD EDX,4	
0045BADD	. 84DB	TEST BL,BL	
0045BADF	. 75 C4	JNZ SHORT DaXXoR.0045BA95	
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	0045BA95 DaXXoR.0045BA95
C 0	ES 0023 32bit 0(FFFFFFFF)

Ahh acá si esta comparando el nombre del proceso que obtuvo, con el nombre OLLYDBG.exe y si son iguales ir a a kaput, jeje en este caso eran iguales veremos que hace, lleguemos al RET

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	
6A 00	PUSH 0	ExitCode = 0
FF75 DC	PUSH DWORD PTR SS:[EBP-24]	hProcess
E8 78300600	CALL <JMP.&KERNEL32.TerminateProcess>	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]	
T taken		
aXXoR.00401E21		

Allí si no son iguales, EAX será diferente de cero y saltara y no pasara nada pero si EAX es igual a cero como en nuestro caso.

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

No salta y va a OpenProcess a hallar un handle nuevamente para matarlo.

00401DB4	8B4D E4	MOV ECX,DWORD PTR SS:[EBP-1C]	
00401DB7	FFB48D FCEFF	PUSH DWORD PTR SS:[EBP+ECX*4-1004]	
00401DBE	6A 00	PUSH 0	ProcessId
00401DC0	6A 01	PUSH 1	Inheritable = FALSE
00401DC2	E8 31300600	CALL <JMP.&KERNEL32.OpenProcess>	Access = TERMINATE
00401DC7	8945 DC	MOV DWORD PTR SS:[EBP-24],EAX	OpenProcess
00401DC9	837D DC 00	CMP DWORD PTR SS:[EBP-24],0	
0012E7F0	00000001	Access = TERMINATE	
0012E7F4	00000000	Inheritable = FALSE	
0012E7F8	000002D4	ProcessId = 2D4	
0012E7FC	00140000		

Si ejecuto con f8

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

Le volvió a asignar el sistema el numero 58, así que sigamos

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

Como vemos llega a la api TerminateProcess auxilio esta por morir mi OLLYDBG a la cual le pasa el handle 58.

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

Y al apretar f8, adiós OLLYDBG se cerro todo, con eso investigamos como funciona la detección por nombre.

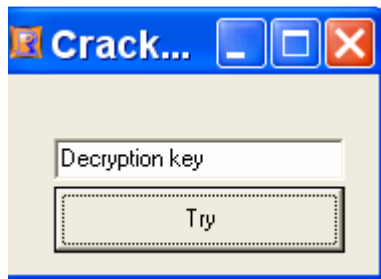
Bueno ya me canse lo haremos un poco a lo maton jeje, reinicio el OLLYDBG y pongo un BP en OpenProcess

File View Debug Plugins Options Window Help			
L E M T W H C / K B R ... S			
7C81E079	8BFF	MOV EDI,EDI	DaXXoR.004664A0
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	8045 F8	LEA EAX,DWORD PTR SS:[EBP-8]	
7C81E09A	50	PUSH EAX	
7C81E09B	8045 E0	LEA EAX,DWORD PTR SS:[EBP-20]	
7C81E09E	50	PUSH EAX	
7C81E09F	FF75 08	PUSH DWORD PTR SS:[EBP+8]	
7C81E0A2	8045 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	ntdll.ZwOpenProcess
7C81E0C2	3BC6	CMP EAX,ESI	
7C81E0C4	5E	POP ESI	
7C81E0C5	0F8C FBAA0100	JL kernel32.7C838BC6	
7C81E0CB	8B45 10	MOV EAX,DWORD PTR SS:[EBP+10]	
7C81E0CE	C9	LEAVE	
7C81E0CF	C2 0C00	RETN 0C	
7C81E0D2	50	PUSH EAX	

Allí para, si modifico la api para que siempre devuelva cero, el programa pensara que no hay procesos corriendo y no tendrá el handle de ninguno para cerrar, podemos cambiar las ultimas líneas de la api.

7C81E0BC	FF15 0C11807C	CALL DWORD PTR DS:[<&ntdll.NtOpenProces	ntdll.ZwOpenProcess
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	
7C81E0D0	EA	PIUSH EAX	

Con eso, la api siempre devolverá cero quitemos todos los Bps y demos RUN



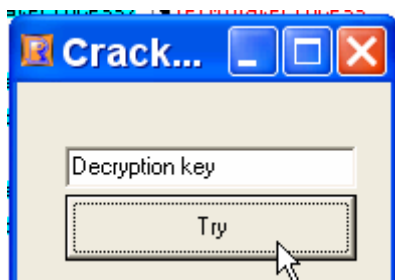
Y si así corre perfectamente, pero también podemos hacer lo siguiente:

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]	
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	894C 0C	MOV DWORD PTR SS:[EBP-24],EAX	

ProcessId
 Inheritable = FALSE
 Access = TERMINATE
 OpenProcess

Cambiamos el JNZ por JMP, así evitamos la protección

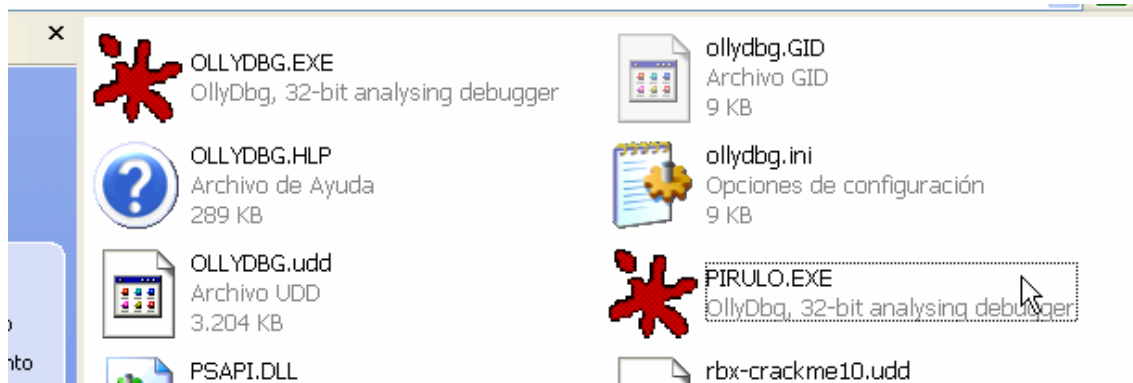
El programa arranca y muestra la ventana veamos que pasa cuando apreto TRY



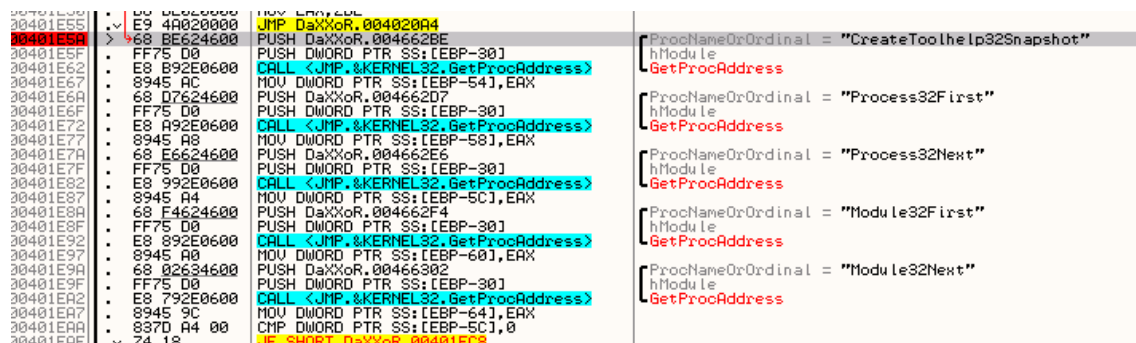
Sale el cartel de error perfectamente eso quiere decir que la protección ANTIDEBUGGER ha sido vencida.

De cualquier manera esto no es lo que se hace habitualmente para vencer esta protección, simplemente copiando el ejecutable OLLYDBG.exe a otra carpeta y cambiándole el nombre por ejemplo a PIRULO.exe y volviéndolo a su carpeta original, de forma que queden ambos el OLLYDBG.exe original y el PIRULO.exe y usando este ultimo, el programa al comparar nunca encontrara el nombre de ningún proceso llamado OLLYDBG, ya que ahora se llamara proceso PIRULO y con eso es vencida esta

protección completamente, de cualquier forma creo que es bueno que sepan como funciona por eso la explicación.



Es importante recordar que aunque usemos un OLLYDBG renombrado, debemos dejar el original en la misma carpeta si no habrá problemas con los plugins.



Hay una parte que el programa posiblemente acceda cuando colocamos un serial bueno, que llama a nuevas apis para una detección diferente, dicha detección la veremos en próximas partes detalladamente.

Hasta la próxima parte 21
Ricardo Narvaja
24 de diciembre de 2005
FELIZ NAVIDAD