

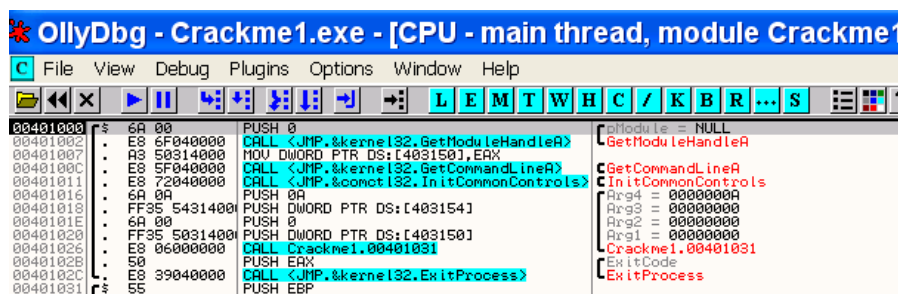
INTRODUCCION AL CRACKING EN OLLYDBG parte 19

DETECCION DEL OLLYDBG – IsDebuggerPresent

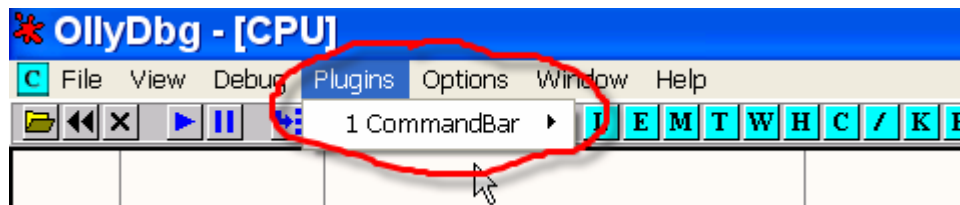
Como dijimos la parte anterior, nos concentraremos en la forma de evitar manualmente y por medio de plugins la detección de OLLYDBG por medio del programa que esta siendo debuggeado, de forma tal de poder trabajarlo tranquilamente ya que la mayoría de los programas, cuando detectan que hay un debugger trabajando sobre ellos, se cierran o comienzan a funcionar en forma diferente por lo cual hay que lograr evitar por todos los medios la detección de OLLYDBG por parte del programa victima.

Esta primera parte tratara sobre la detección por medio de la api IsDebuggerPresent, la cual es la mas comun de todas.

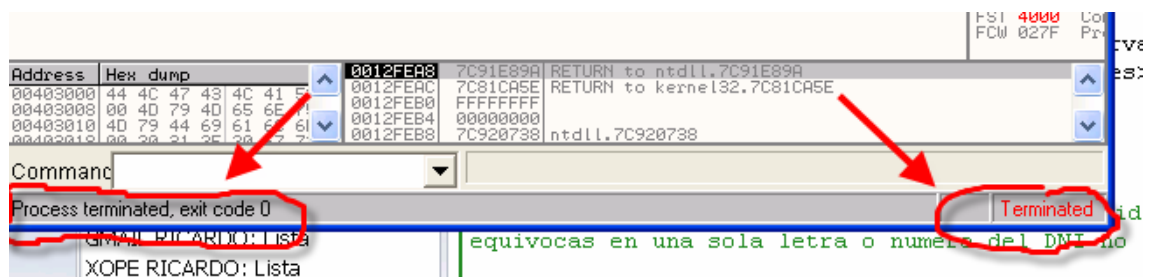
Para ello utilizaremos el Crackme1.exe que esta adjunto a esta leccion, lo arrancamos en OLLYDBG.



Recordemos que mi OLLYDBG por ahora solo tiene el plugin COMMAND BAR, por lo cual, no tiene cargado ningún plugin que pueda evitar la detección por medio de la api IsDebuggerPresent, pero como funciona dicha detección?

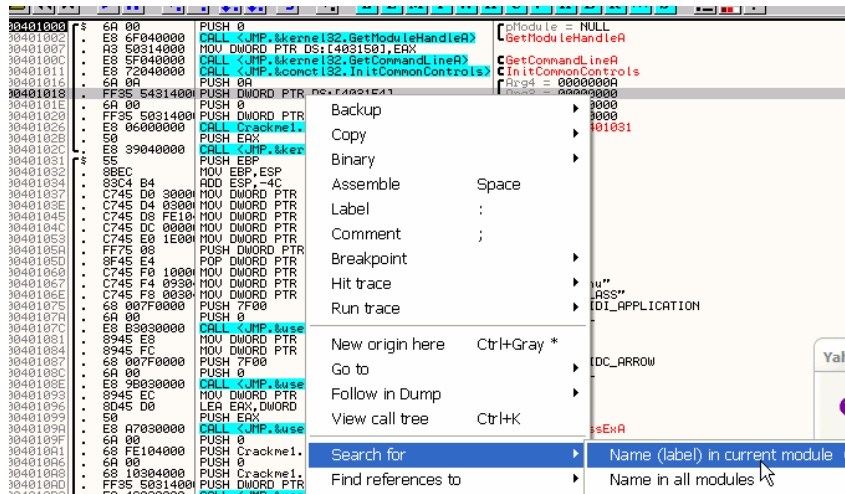


Si corremos el crackme con F9 vemos que no solo no abre la ventana del crackme si no que se cierra el programa.



OLLYDBG nos indica en la parte inferior que el programa termino, y ni siquiera vimos la ventana del crackme, jeje, que ocurre aquí, muy sencillo el crackme utiliza el método mas conocido para detectar que esta siendo debuggeado que es llamar a la api IsDebuggerPresent.

Reiniciemos el crackme y veamos si la susodicha esta en la lista de las apis que utiliza la victima, haciendo click derecho en el listado y eligiendo ..

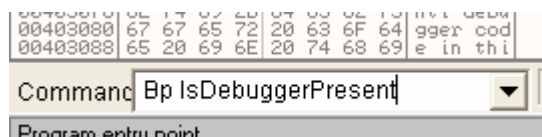


Vemos la lista de apis usadas

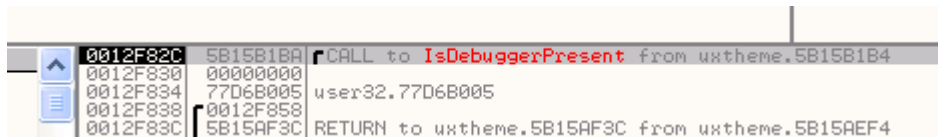
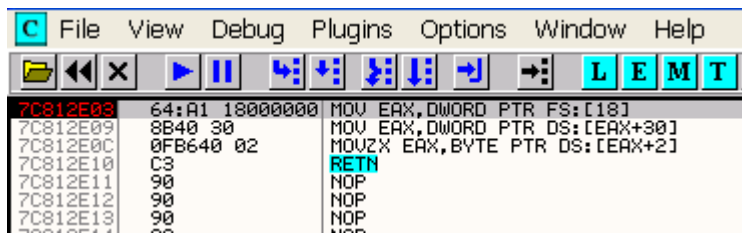
Address	Section	Type	Name
00402050	.rdata	Import	user32.CreateDialogParamA
00402058	.rdata	Import	user32.DefWindowProcA
00402054	.rdata	Import	user32.DestroyWindow
00402034	.rdata	Import	user32.DispatchMessageA
00402014	.rdata	Import	kernel32.ExitProcess
00402018	.rdata	Import	kernel32.GetCommandLineA
0040204C	.rdata	Import	user32.GetDlgItem
00402048	.rdata	Import	user32.GetMessageA
0040200C	.rdata	Import	kernel32.GetModuleHandleA
00402008	.rdata	Import	kernel32.GetVolumeInformationA
00402030	.rdata	Import	user32.GetWindowTextA
00402000	.rdata	Import	comctl32.InitCommonControls
00402010	.rdata	Import	kernel32.IsDebuggerPresent
0040202C	.rdata	Import	user32.LoadCursorA
00402020	.rdata	Import	user32.LoadIconA
00402024	.rdata	Import	user32.MessageBoxA
00401000	.text	Export	<ModuleEntryPoint>
00402028	.rdata	Import	user32.PostQuitMessage
0040205C	.rdata	Import	user32.RegisterClassExA
00402064	.rdata	Import	user32.SendMessageA
00402038	.rdata	Import	user32.SetWindowTextA
0040203C	.rdata	Import	user32.ShowWindow
00402040	.rdata	Import	user32.TranslateMessage
00402044	.rdata	Import	user32.UpdateWindow
00402060	.rdata	Import	user32.wsprintfA

Y si parece que la usa jeje

Pongamos un BP en dicha api a ver si para el programa cuando la utilice.



Damos Run y para en la api



Vemos en el stack que es una api sin parámetros, lo único que consulta es si el programa esta siendo debuggeado o no, para los que tienen dudas con alguna api y su funcionamiento, lo mejor es consultar el archivo WINAPIS32 que esta en mi http en herramientas.

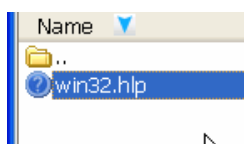
<http://www.ricnar456.dyndns.org>

user:hola
pass:hola

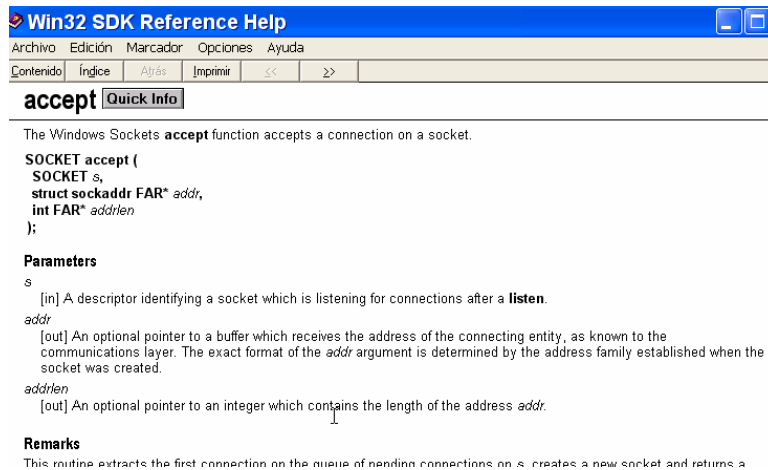
y allí en la carpeta HERRAMIENTAS/V-W-X-Y-Z esta el archivo que contiene la especificación de las apis mas importante publicada por Microsoft, no están todas pero si la gran mayoría, el único problema es que esta en ingles, y es un poquito complicado de entender para el que no esta acostumbrado, pero a partir de ahora, para irnos acostumbrando, veremos las definiciones de las apis allí.

<http://www.ricnar456.dyndns.org/HERRAMIENTAS/V-W-X-Y-Z/winapi32%20NEW.rar>

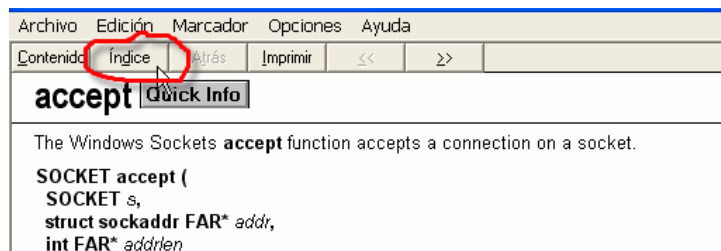
user:hola
pass:hola



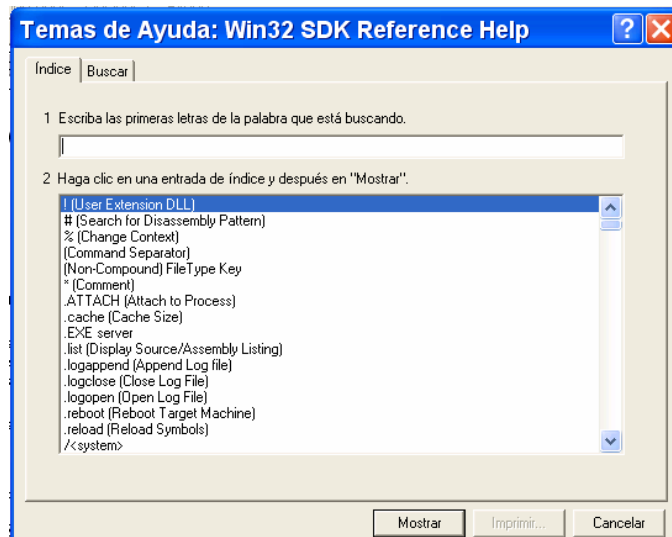
El archivo zip, contiene un archivo de extensión HLP que es el que nos interesa lo ejecutamos.



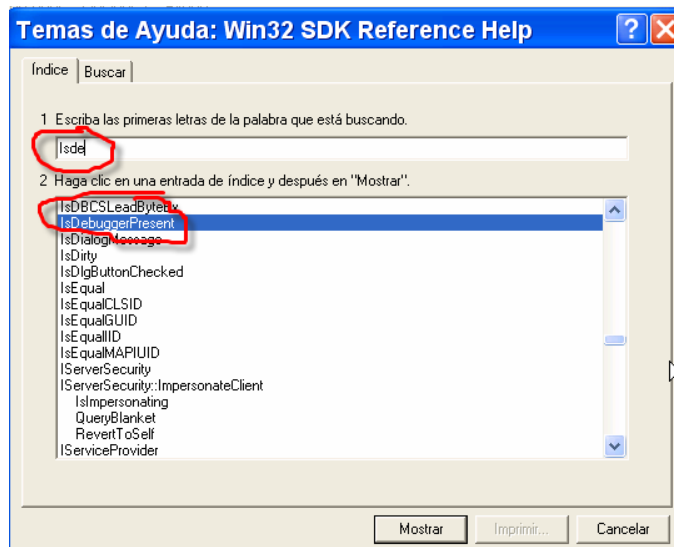
Allí se abrió, ahora como buscamos la api que nos interesa?



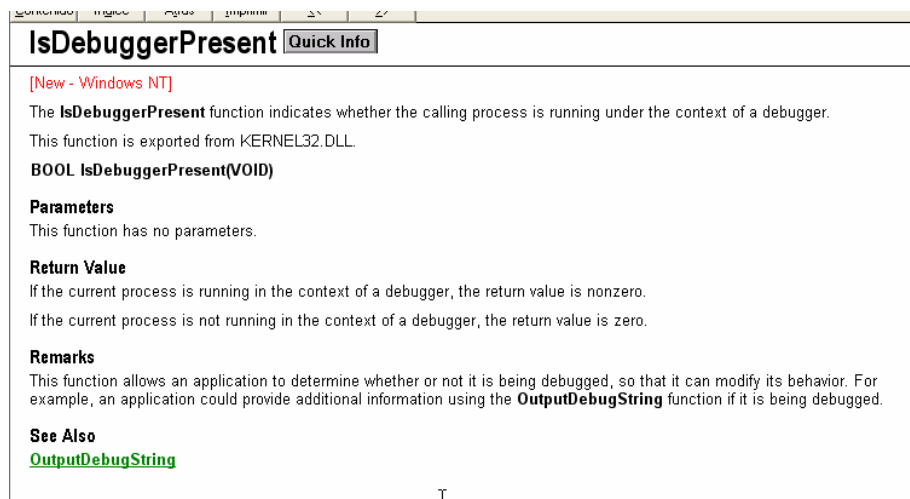
Vamos a INDICE



Allí en la barra a medida que vamos tipeando el nombre de la api, va buscando las que coinciden con la letra que tipeamos



Bueno allí la hallamos hacemos doble click en IsDebuggerPresent

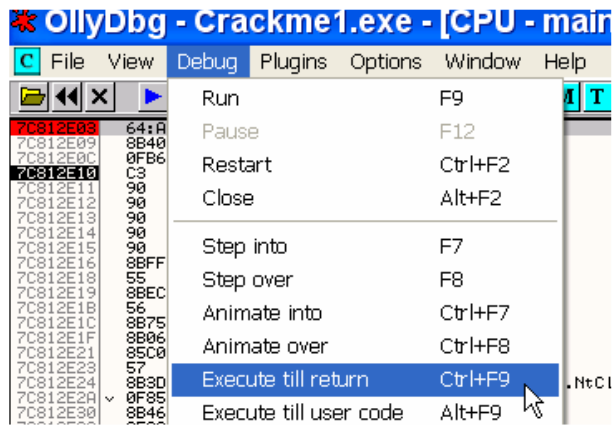


Bueno allí esta la aclaración y traducimos

La api IsDebuggerPresent indica cuando el proceso que la llamo, esta bajo el contexto de un debugger o sea siendo debuggeado.

Luego dice que esta api o función es exportada por la KERNEL32.dll, que no tiene parámetros como vimos, y en RETURN VALUES o sea en valores de retorno, dice que si el programa esta siendo debuggeado el retorno es un valor distinto de cero, y si no lo esta es cero.

Bueno esa información es muy importante, ejecutemos la api hasta el RET a ver donde devuelve la información.



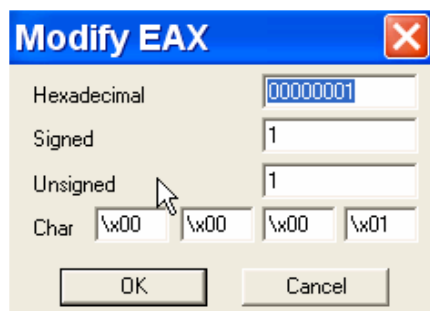
Allí llegue al RET si veo los registros

7C812E01	90	NOP
7C812E02	90	NOP
7C812E03	64:A1 18000000	MOV EAX,DWORD PTR FS:[18]
7C812E09	8B40 30	MOV EAX,DWORD PTR DS:[EAX+30]
7C812E0C	0FB640 02	MOVZX EAX,BYTE PTR DS:[EAX+2]
7C812E10	C3	RET
7C812E11	90	NOP
7C812E12	90	NOP
7C812E13	90	NOP

Registers (FPU)	
EAX	00000001
ECX	77D6B005 use
EDX	00000000
EBX	0012F874
ESP	0012F82C
EBP	0012F838
ESI	00000001
EDI	00000000
EIP	7C812E10 key

El que esta en rojo o sea que cambio es EAX, como casi todas las apis, devuelven en EAX el valor de retorno, así que EAX es igual a 1 lo cual le dice al programa que esta siendo debuggeado.

Problemos que ocurre si lo cambiamos a mano y ponemos EAX=0 o sea que no esta siendo debuggeado.



Registers (FPU)	
EAX	00000000
ECX	77D6B005 user32
EDX	00000000
EBX	0012F874
ESP	0012F82C
EBP	0012F838
ESI	00000001
EDI	00000000
EIP	7C812E10 kernel
C 0	ES 0023 32bit
P 0	CS 001B 32bit
A 0	SS 0023 32bit

Demos RUN ahora

7C812E07	90	NOP
7C812E08	64:A1 18000000	MOV EAX,DWORD PTR FS:[18]
7C812E09	8B40 30	MOV EAX,DWORD PTR DS:[EAX+30]
7C812E0C	0FB640 02	MOVZX EAX,BYTE PTR DS:[EAX+2]
7C812E10	C3	RET
7C812E11	90	NOP
7C812E12	90	NOP
7C812E13	90	NOP

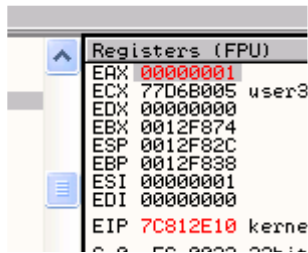
Vemos que vuelve a parar en la api, así que llegamos al ret nuevamente y cambiamos EAX a cero.



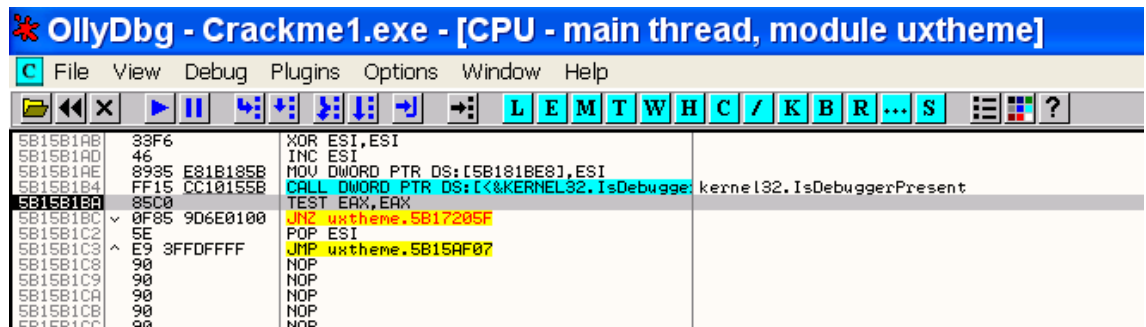
Vemos que el programa arranca, o sea que toda su detección se basa en dicha api, reiniciémoslo y cuando pare en IsDebuggerPresent, veamos que hace con el valor que le devuelve la api.

File View Debug Plugins Options Window Help		
[Icons]		
7C812E07	90	NOP
7C812E08	64:A1 18000000	MOV EAX,DWORD PTR FS:[18]
7C812E09	8B40 30	MOV EAX,DWORD PTR DS:[EAX+30]
7C812E0C	0FB640 02	MOVZX EAX,BYTE PTR DS:[EAX+2]
7C812E10	C3	RET
7C812E11	90	NOP
7C812E12	90	NOP
7C812E13	90	NOP

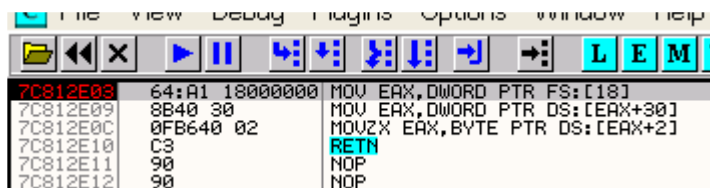
Allí paro nuevamente lleguemos hasta el ret, podemos tracear con f8 pues la api es cortita.



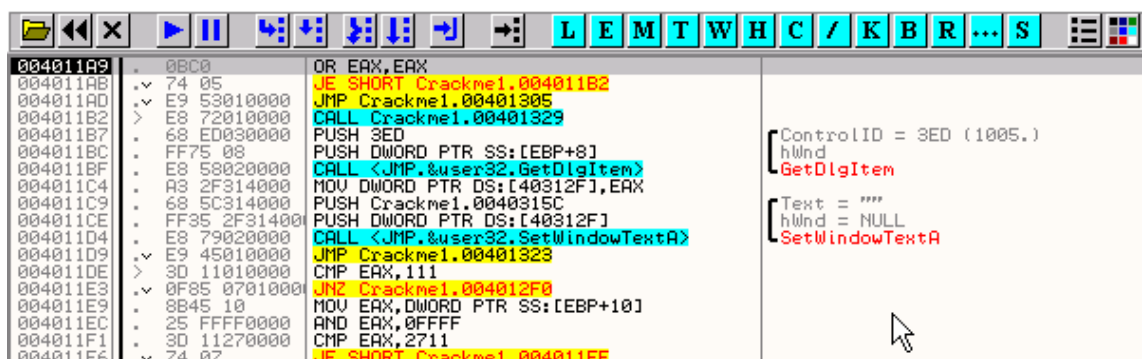
Allí esta, al llegar al RET, vale EAX=1, volvamos al programa con f8 nuevamente.



Vemos que la primera vez no es llamado desde el mismo ejecutable si no de una dll que es uxtheme, demos RUN a ver la segunda vez que para en la api.



Ahí paro la segunda vez, llego hasta el RET, y vuelvo al programa traceando con f8.



Llega aquí, y allí hay un JE que testea si EAX se cero o no, veamos.

Address	Disassembly	Comment
004011A9	0BC0	OR EAX,EAX
004011AB	74 05	JE SHORT Crackme1.004011B2
004011AD	E9 53010000	JMP Crackme1.00401305
004011B2	E8 72010000	CALL Crackme1.00401329
004011B7	68 ED030000	PUSH 3ED
004011BC	FF75 08	PUSH DWORD PTR SS:[EBP+8]
004011BF	E8 58020000	CALL <JMP.&user32.GetDlgItem>
004011C4	A3 2F314000	MOV DWORD PTR DS:[40312F],EAX
004011C9	68 5C314000	PUSH Crackme1.0040315C
004011CE	FF35 2F314000	PUSH DWORD PTR DS:[40312F]
004011D4	E8 79020000	CALL <JMP.&user32.SetWindowTextA>
004011D9	E9 45010000	JMP Crackme1.00401323
004011DE	3D 11010000	CMP EAX,111
004011E3	0F85 07010000	JNZ Crackme1.004012F0
004011E9	8B45 10	MOV EAX,DWORD PTR SS:[EBP+10]
004011EC	25 FFFF0000	AND EAX,0FFFF
004011F1	3D 11270000	CMP EAX,2711
004011F6	74 07	JE SHORT Crackme1.004011FF

ControlID = 3ED (1005.)
hWnd
GetDlgItem

Text = ""
hWnd = NULL
SetWindowTextA

Vemos que el JE al ser EAX diferente de cero no salta, observamos también que si saltara continua la ejecución del programa con GetDlgItem leyendo de la ventana del crackme.

Como no salta continua al JMP, el cual no es evitado, y aquí si se puede aplicar bien la palabra el crackme me tiro fuera, vemos que el JMP nos lleva bien lejos, posiblemente a cerrar el programa.

Address	Disassembly	Comment
004011A9	0BC0	OR EAX,EAX
004011AB	74 05	JE SHORT Crackme1.004011B2
004011AD	E9 53010000	JMP Crackme1.00401305
004011B2	E8 72010000	CALL Crackme1.00401329
004011B7	68 ED030000	PUSH 3ED
004011BC	FF75 08	PUSH DWORD PTR SS:[EBP+8]
004011BF	E8 58020000	CALL <JMP.&user32.GetDlgItem>
004011C4	A3 2F314000	MOV DWORD PTR DS:[40312F],EAX
004011C9	68 5C314000	PUSH Crackme1.0040315C
004011CE	FF35 2F314000	PUSH DWORD PTR DS:[40312F]
004011D4	E8 79020000	CALL <JMP.&user32.SetWindowTextA>
004011D9	E9 45010000	JMP Crackme1.00401323
004011DE	3D 11010000	CMP EAX,111
004011E3	0F85 07010000	JNZ Crackme1.004012F0
004011E9	8B45 10	MOV EAX,DWORD PTR SS:[EBP+10]
004011EC	25 FFFF0000	AND EAX,0FFFF
004011F1	3D 11270000	CMP EAX,2711
004011F6	74 07	JE SHORT Crackme1.004011FF
004011F8	3D EC030000	CMP EAX,3EC
004011FD	75 13	JNZ SHORT Crackme1.00401212
004011FF	6A 00	PUSH 0
00401201	6A 00	PUSH 0
00401203	6A 10	PUSH 10
00401205	FF75 08	PUSH DWORD PTR SS:[EBP+8]
00401208	E8 3F020000	CALL <JMP.&user32.SendMessageA>
0040120D	E9 11010000	JMP Crackme1.00401323
00401212	3D 75270000	CMP EAX,2775
00401217	75 19	JNZ SHORT Crackme1.00401232
00401219	6A 00	PUSH 0
0040121B	68 25314000	PUSH Crackme1.00403125
00401220	68 19304000	PUSH Crackme1.00403019
00401225	FF75 08	PUSH DWORD PTR SS:[EBP+8]
00401228	E8 0D020000	CALL <JMP.&user32.MessageBoxA>
0040122D	E9 F1000000	JMP Crackme1.00401323
00401232	3D EB030000	CMP EAX,3EB
00401237	0F85 E6000000	JNZ Crackme1.00401323
0040123D	68 EE030000	PUSH 3EE
00401242	FF75 08	PUSH DWORD PTR SS:[EBP+8]
00401245	E8 D2010000	CALL <JMP.&user32.GetDlgItem>
0040124B	A3 33314000	MOV DWORD PTR DS:[403133],EAX
0040124F	68 F1030000	PUSH 3F1
00401254	FF75 08	PUSH DWORD PTR SS:[EBP+8]
00401257	E8 C0010000	CALL <JMP.&user32.GetDlgItem>
0040125C	A3 37314000	MOV DWORD PTR DS:[403137],EAX

ControlID = 3ED (1005.)
hWnd
GetDlgItem

Text = ""
hWnd = NULL
SetWindowTextA

lParam = 0
wParam = 0
Message = WM_CLOSE
hWnd
SendMessageA

Style = MB_OK|MB_APPLMODAL
Title = " Note"
Text = " 1. Written by RadASM, thanks
hOwner
MessageBoxA

ControlID = 3EE (1006.)
hWnd
GetDlgItem

ControlID = 3F1 (1009.)
hWnd
GetDlgItem

Continuemos traceando a ver donde va

Address	Disassembly	Comment
004012FF	> 837D 0C 02	CMP DWORD PTR SS:[EBP+C],2
00401303	> 75 09	JNZ SHORT Crackme1.0040130E
00401305	> 6A 00	PUSH 0
00401307	> E8 34010000	CALL <JMP.&user32.PostQuitMessage>
0040130C	> EB 15	JMP SHORT Crackme1.00401323
0040130E	> FF75 14	PUSH DWORD PTR SS:[EBP+14]
00401311	> FF75 10	PUSH DWORD PTR SS:[EBP+10]

ExitCode = 0
PostQuitMessage

lParam
wParam

Vemos que va a la api PostQuitMessage miremos en el winapis32

PostQuitMessage Quick Info

The **PostQuitMessage** function indicates to Windows that a thread has made a request to terminate (quit). It is typically used in response to a [WM_DESTROY](#) message.

```
VOID PostQuitMessage(  
    int nExitCode    // exit code  
);
```

Parameters

nExitCode
Specifies an application exit code. This value is used as the *wParam* parameter of the WM_QUIT message.

Return Values

This function does not return a value.

Remarks

The **PostQuitMessage** function posts a [WM_QUIT](#) message to the thread's message queue and returns immediately; the function simply indicates to the system that the thread is requesting to quit at some time in the future.

When the thread retrieves the WM_QUIT message from its message queue, it should exit its message loop and return control to Windows. The exit value returned to Windows must be the *wParam* parameter of the WM_QUIT message.

See Also

[GetMessage](#), [PeekMessage](#), [PostMessage](#), [WM_DESTROY](#), [WM_QUIT](#)

Traducción esta api pasa un mensaje para cerrar la ventana, jeje o sea esta api postea un mensaje WM_QUIT o sea es una indicación de que esta cerrando cosas, haciendo las valijas y se ira.

Al volver del mismo

00401305	>	6A 00	PUSH 0	ExitCode = 0
00401307	.	E8 34010000	CALL <JMP.&user32.PostQuitMessage>	PostQuitMessage
0040130C	>	EB 15	JMP SHORT Crackme1.00401323	
0040130E	.	FF75 14	PUSH DWORD PTR SS:[EBP+14]	[Param
00401311	.	FF75 10	PUSH DWORD PTR SS:[EBP+10]	wParam
00401314	.	FF75 0C	PUSH DWORD PTR SS:[EBP+C]	Message
00401317	.	FF75 08	PUSH DWORD PTR SS:[EBP+8]	hWnd
0040131A	.	E8 EB000000	CALL <JMP.&user32.DefWindowProcA>	DefWindowProcA
0040131F	.	C9	LEAVE	
00401320	.	C2 1000	RETN 10	
00401323	>	33C0	XOR EAX,EAX	
00401325	.	C9	LEAVE	
00401326	.	C2 1000	RETN 10	
00401329	.	55	PUSH EBP	
0040132A	.	8BEC	MOV EBP,ESP	

Y si tenemos paciencia de tracear bastante mientras va cerrando todo, observaremos que llegara a la api ExitProcess que es la que cierra el proceso.(si no tienen paciencia le ponen un BP ExitProcess y parara en ella)

0040101E	.	5A 00	PUSH 0	Hrsc = 00000000
00401020	.	FF35 50314000	PUSH DWORD PTR DS:[403150]	Arg1 = 00400000
00401026	.	E8 06000000	CALL Crackme1.00401031	Crackme1.00401031
0040102B	.	50	PUSH EAX	ExitCode
0040102C	.	E8 39040000	CALL <JMP.&kernel32.ExitProcess>	ExitProcess
00401031	.	55	PUSH EBP	
00401032	.	8BEC	MOV EBP,ESP	
00401034	.	83C4 B4	ADD ESP,-4C	
00401037	.	C745 D0 3000	MOV DWORD PTR SS:[EBP-30],30	

Así que ya vimos que el salto que esta al volver de la api IsDebuggerPresent es el que toma la decisión de si corre o se cierra, una solución podría ser parchearlo, reiniciemos el programa y lleguemos al salto decisivo.

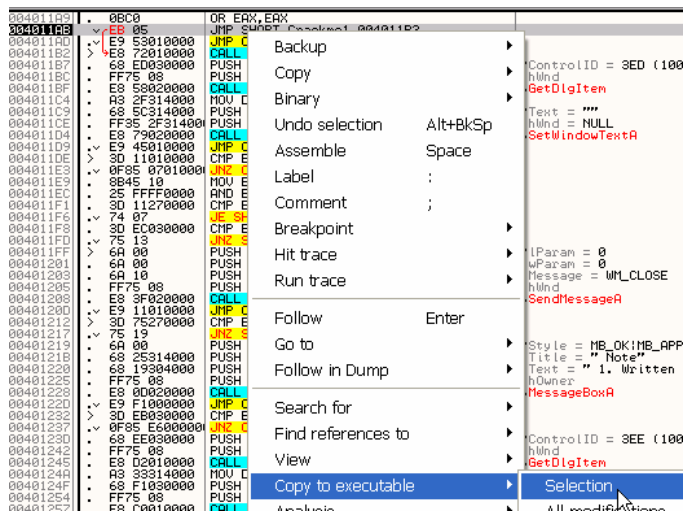
004011A9	.	0BC0	OR EAX,EAX	
004011AB	>	74 05	JE SHORT Crackme1.004011B2	
004011AD	.	E9 53010000	JMP Crackme1.00401305	
004011B2	>	E8 72010000	CALL Crackme1.00401329	
004011B7	.	68 ED030000	PUSH 3ED	ControlID = 3ED (1005)
004011BC	.	FF75 08	PUSH DWORD PTR SS:[EBP+8]	hWnd
004011BF	.	E8 58020000	CALL <JMP.&user32.GetDlgItem>	GetDlgItem

Allí esta y como habíamos visto tenia que saltar siempre, por lo cual lo cambiamos a JMP, tecleando la barra espaciadora y escribiendo JMP 4011b2.

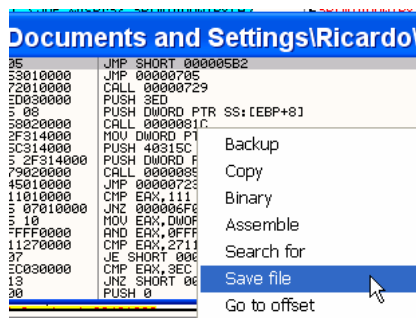


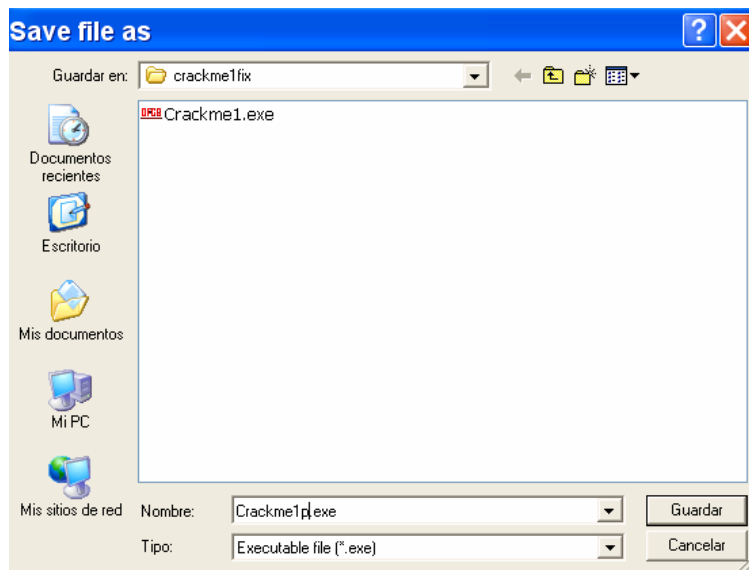
Allí esta, vemos que ahora evita el JMP que va a cerrar el programa y el mismo continua funcionando.

Guardemos los cambios en un archivo con otro nombre, hagamos click derecho



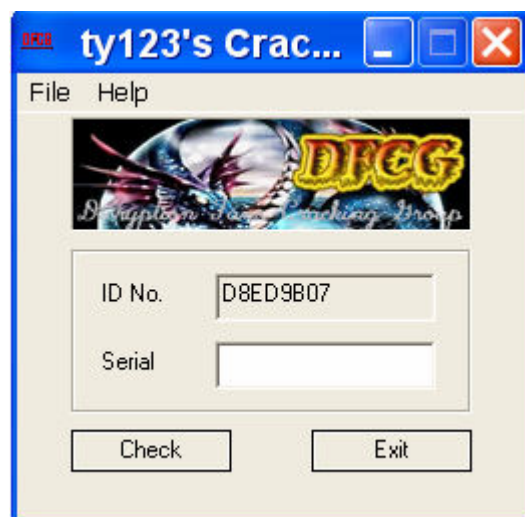
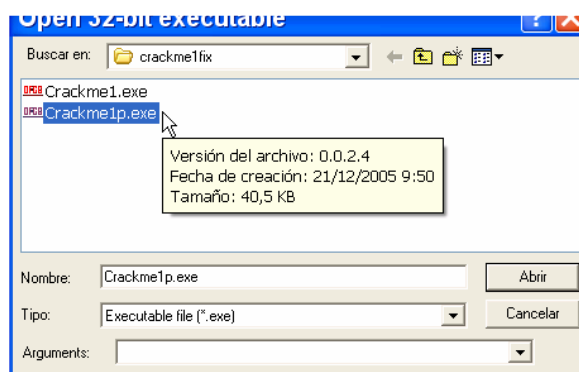
Y en la nueva ventana que se abre hagamos nuevamente click derecho





Lo guardo con el nombre crackme1p, así tengo los dos el original y el parcheado.

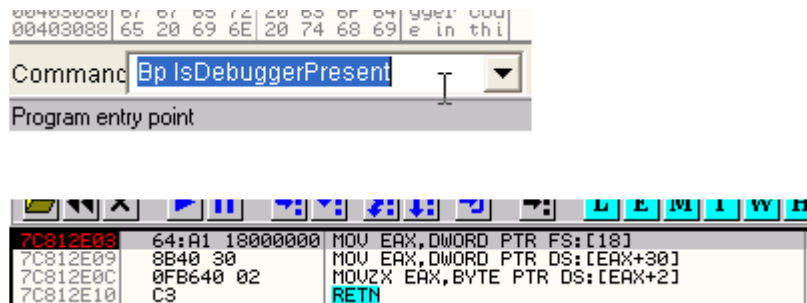
Ahora abramos el parcheado en OLLYDBG para ver si corre en el mismo, sin poner ningún BP ni nada



Corre perfectamente, así que ya sabemos hacer correr un programa cambiando EAX al volver de IsDebuggerPresent, también sabemos parchear para que corra y no tengamos que tomarnos el trabajo de hacerlo a mano.

Con este crackme no tendremos problema, para parchearlo y que corra, aunque por supuesto hay métodos mas sencillos usando plugins que nos evitan realizar todo este trabajo, pero es bueno conocer como funcionan las cosas.

Ahora la pregunta que alguien se podría hacer es la siguiente volvamos al original y miremos la api.

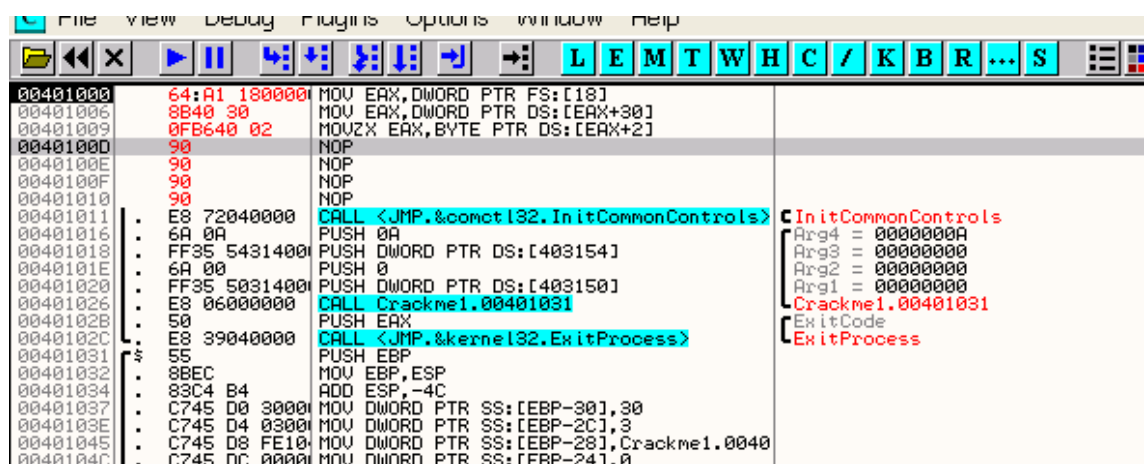
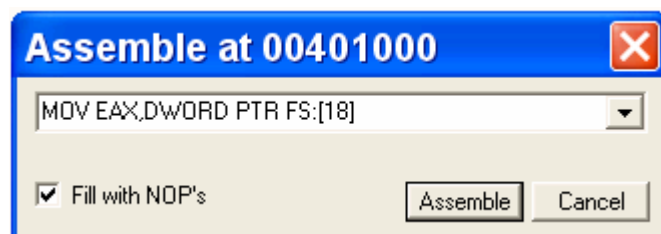


Son tres tristes MOV y con eso la api determina si esta siendo debuggeado el programa o no.

Lo primero que se me viene a la cabeza es, porque el programa no podría ejecutar estas tres líneas mezcladas entre su código, las cuales le devolverían en EAX el valor 1 si esta siendo debuggeado, probemos reiniciemos el crackme original y en el entry point escribamos.

```
MOV EAX,DWORD PTR FS:[18]
MOV EAX,DWORD PTR DS:[EAX+30]
MOVZX EAX,BYTE PTR DS:[EAX+2]
```

Podemos copiar y pegar cada línea de aquí.



Allí tenemos las tres líneas escritas.

Address	Disassembly	Comment
00401000	MOV EAX,DWORD PTR FS:[18]	
00401006	MOV EAX,DWORD PTR DS:[EAX+30]	
00401009	MOVZX EAX,BYTE PTR DS:[EAX+2]	
0040100D	NOP	
0040100E	NOP	
0040100F	NOP	
00401010	NOP	
00401011	CALL <JMP.&comet132.InitCommonControls>	CInitCommonCor
00401016	PUSH 0A	rArg4

Vemos que cuando paso la ultima instrucci3n, EAX se puso a 1 en la misma forma que lo hace en la api.

Register	Value
EAX	00000001
ECX	0012FFB0
EDX	7C91EB94 n
EBX	7FFDC000
ESP	0012FFC4
EBP	0012FFC0
ESI	FFFFFFFF
EDI	7C920738 n
EIP	0040100D C
CS	0023 3

Y obviamente nunca el programa parara en la api IsDebuggerPresent, pues nunca fue llamada, e igual detecto que esta siendo debuggeado.

Por ello es muy importante determinar que es lo que lee la api o mejor dicho esas tres l3neas famosas.

En concreto cuando un programa arranca el sistema guarda en una direcci3n de memoria determinada un uno si detecta que el programa esta siendo debuggeado y un cero si no lo esta, ese byte es el que leen las tres l3neas estas y por supuesto la api tambi3n.

Como podemos localizarlo al byte?, pues vayamos nuevamente a la primera l3nea de estas tres y miremos paso a paso lo que hace.

La primera l3nea es

Address	Disassembly	Comment
00401000	MOV EAX,DWORD PTR FS:[18]	
00401006	MOV EAX,DWORD PTR DS:[EAX+30]	
00401009	MOVZX EAX,BYTE PTR DS:[EAX+2]	
0040100D	NOP	
0040100E	NOP	
0040100F	NOP	
00401010	NOP	
00401011	CALL <JMP.&comet132.InitCommonControls>	CInitCommonCor
00401016	PUSH 0A	rArg4 = 00000001
00401018	PUSH DWORD PTR DS:[403154]	rArg3 = 00000001
0040101E	PUSH 0	rArg2 = 00000001
00401020	PUSH DWORD PTR DS:[403154]	rArg1 = 00000001

Pues aqu3 explicaremos lo m3s sencillo posible algo de teor3a, en el OLLYDBG vemos aqu3

Registers (FPU)	
EAX	7FFDC000
ECX	0012FFB0
EDX	7C91EB94 ntdll.KiFastSystemCallRet
EBX	7FFDC000
ESP	0012FFC4
EBP	0012FFF0
ESI	FFFFFFFF
EDI	7C920738 ntdll.7C920738
EIP	00401009 Crackme1.00401009
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 7FFDC000(FFF)
T 0	GS 0000 NULL
D 0	
O 0	LastErr ERROR_NO_IMPERSONATION_TC
EFL	00000246 (NO,NB,E,BE,NS,PE,GE,LE)
ST0	empty -UNORM BCE0 01050104 00000000
ST1	empty 0.0
ST2	empty 0.0
ST3	empty 0.0
ST4	empty 0.0
ST5	empty 0.0

En la ventana de los registros un valor muy importante, no quiero dar nombres difíciles pero ese valor apunta a una estructura que guarda información muy importante sobre el programa que arranco, vayamos a ver en el DUMP dicha dirección (en su maquina puede estar en otra dirección solo vayan a la dirección que les indica olly allí)

Address	Hex dump	ASCII
7FFDF000	E0 FF 12 00 00 00 13 00	0 0 0 0 0 0 0 0
7FFDF008	00 00 12 00 00 00 00 00	. 0 0 0 0 0 0 0 0
7FFDF010	00 1E 00 00 00 00 00 00	. 0 0 0 0 0 0 0 0
7FFDF018	00 F0 FD 7F 00 00 00 00	. 0 0 0 0 0 0 0 0
7FFDF020	E8 09 00 00 A8 0A 00 00	. 0 0 0 0 0 0 0 0
7FFDF028	00 00 00 00 00 00 00 00	. 0 0 0 0 0 0 0 0
7FFDF030	00 C0 FD 7F 1D 05 00 00	. 0 0 0 0 0 0 0 0
7FFDF038	00 00 00 00 00 00 00 00	. 0 0 0 0 0 0 0 0
7FFDF040	00 C7 2B E3 00 00 00 00	. 0 0 0 0 0 0 0 0
7FFDF048	00 00 00 00 00 00 00 00	. 0 0 0 0 0 0 0 0
7FFDF050	00 00 00 00 00 00 00 00	. 0 0 0 0 0 0 0 0
7FFDF058	00 00 00 00 00 00 00 00	. 0 0 0 0 0 0 0 0
7FFDF060	00 00 00 00 00 00 00 00	. 0 0 0 0 0 0 0 0

Esta estructura se llama TEB O TIB y guarda buena información que si consultamos nos puede ayudar muchísimo con el programa, en la TIB podemos ver por ejemplo donde comienza y donde termina el stack del programa.

Address	Hex dump	ASCII
7FFDF000	E0 FF 12 00 00 00 13 00	0 0 0 0 0 0 0 0
7FFDF008	00 00 12 00 00 00 00 00	. 0 0 0 0 0 0 0 0
7FFDF010	00 1E 00 00 00 00 00 00	. 0 0 0 0 0 0 0 0
7FFDF018	00 F0 FD 7F 00 00 00 00	. 0 0 0 0 0 0 0 0
7FFDF020	E8 09 00 00 A8 0A 00 00	. 0 0 0 0 0 0 0 0
7FFDF028	00 00 00 00 00 00 00 00	. 0 0 0 0 0 0 0 0
7FFDF030	00 C0 FD 7F 1D 05 00 00	. 0 0 0 0 0 0 0 0
7FFDF038	00 00 00 00 00 00 00 00	. 0 0 0 0 0 0 0 0
7FFDF040	00 C7 2B E3 00 00 00 00	. 0 0 0 0 0 0 0 0
7FFDF048	00 00 00 00 00 00 00 00	. 0 0 0 0 0 0 0 0
7FFDF050	00 00 00 00 00 00 00 00	. 0 0 0 0 0 0 0 0
7FFDF058	00 00 00 00 00 00 00 00	. 0 0 0 0 0 0 0 0
7FFDF060	00 00 00 00 00 00 00 00	. 0 0 0 0 0 0 0 0

Si vemos en M la sección del stack

0012C000	00001000								
0012D000	00003000			stack of ma	Priv	RW	Gua	RW	
00130000	00003000				Priv	RW	Gua	RW	
00140000	00004000				Map	R		R	
00240000	00006000				Priv	RW		RW	

Vemos que comienza en 12d000 y termina justo antes de donde se inicia la sección siguiente en 130000.

Hay muchos otros valores interesantes como el primero, que veremos mas detalladamente cuando estudiemos las excepciones, pero como adelanto si miramos en el stack ese valor 12ffE0 en mi maquina.

Address	Hex dump	ASCII
7FFDF000	E0 FF 12 00 00 00 13 00	0 0...!!
7FFDF008	00 00 12 00 00 00 00 00	. \$+.....
7FFDF010	00 1E 00 00 00 00 00 00	. \$+.....
7FFDF018	00 F0 FD 7F 00 00 00 00	. \$+.....
7FFDF020	E8 09 00 00 A8 0A 00 00	. \$+.....
7FFDF028	00 00 00 00 00 00 00 00	. \$+.....
7FFDF030	00 C0 FD 7F 1D 05 00 00	. \$+.....
7FFDF038	00 00 00 00 00 00 00 00	. \$+.....
7FFDF040	00 C7 2B E3 00 00 00 00	. \$+.....
7FFDF048	00 00 00 00 00 00 00 00	. \$+.....
7FFDF050	00 00 00 00 00 00 00 00	. \$+.....
7FFDF058	00 00 00 00 00 00 00 00	. \$+.....
7FFDF060	00 00 00 00 00 00 00 00	. \$+.....

Lo busco en el stack

0012FFC4	7C816D4F	RETURN to kernel32.7C816D4F
0012FFC8	7C920738	ntdll.7C920738
0012FFCC	FFFFFFFF	
0012FFD0	7FFDC000	
0012FFD4	8054A938	
0012FFD8	0012FFC8	
0012FFDC	84543DA8	
0012FFE0	FFFFFFFF	End of SEH chain
0012FFE4	7C8399F3	SE handler
0012FFE8	7C816D58	kernel32.7C816D58
0012FFEC	00000000	
0012FFF0	00000000	
0012FFF4	00000000	
0012FFF8	00401000	Crackme1.<ModuleEntryPoint>
0012FFFC	00000000	

Vemos que esta marcado como END OF SEH CHAIN, por ahora no diremos mas de el, pero si que esta relacionado con las excepciones.

Una cosa interesante de este TIB es que existe una nomenclatura para acceder al cualquier valor del mismo, por ejemplo este valor que marque antes seria Fs: [0] .

Address	Hex dump	ASCII
7FFDF000	E0 FF 12 00 00 00 13 00	0 0...!!
7FFDF008	00 00 12 00 00 00 00 00	. \$+.....
7FFDF010	00 1E 00 00 00 00 00 00	. \$+.....
7FFDF018	00 F0 FD 7F 00 00 00 00	. \$+.....
7FFDF020	E8 09 00 00 A8 0A 00 00	. \$+.....
7FFDF028	00 00 00 00 00 00 00 00	. \$+.....
7FFDF030	00 C0 FD 7F 1D 05 00 00	. \$+.....
7FFDF038	00 00 00 00 00 00 00 00	. \$+.....
7FFDF040	00 C7 2B E3 00 00 00 00	. \$+.....
7FFDF048	00 00 00 00 00 00 00 00	. \$+.....
7FFDF050	00 00 00 00 00 00 00 00	. \$+.....
7FFDF058	00 00 00 00 00 00 00 00	. \$+.....
7FFDF060	00 00 00 00 00 00 00 00	. \$+.....
7FFDF068	00 00 00 00 00 00 00 00	. \$+.....
7FFDF070	00 00 00 00 00 00 00 00	. \$+.....
7FFDF078	00 00 00 00 00 00 00 00	. \$+.....
7FFDF080	00 00 00 00 00 00 00 00	. \$+.....
7FFDF088	00 00 00 00 00 00 00 00	. \$+.....

Command	? fs:[0]	HEX: 12FFE0 - DEC: 1245152 - ASCII: 0x00
---------	----------	--

Ese seria Fs: [0], si en la command bar buscamos el valor de fs: [1], ese seria

Address	Hex dump	ASCII
7FFDF000	E0 FF 12 00 00 00 13 00	0 0...!!
7FFDF008	00 00 12 00 00 00 00 00	. \$+.....
7FFDF010	00 1E 00 00 00 00 00 00	. \$+.....
7FFDF018	00 F0 FD 7F 00 00 00 00	. \$+.....
7FFDF020	E8 09 00 00 A8 0A 00 00	. \$+.....
7FFDF028	00 00 00 00 00 00 00 00	. \$+.....
7FFDF030	00 C0 FD 7F 1D 05 00 00	. \$+.....
7FFDF038	00 00 00 00 00 00 00 00	. \$+.....
7FFDF040	00 C7 2B E3 00 00 00 00	. \$+.....
7FFDF048	00 00 00 00 00 00 00 00	. \$+.....
7FFDF050	00 00 00 00 00 00 00 00	. \$+.....
7FFDF058	00 00 00 00 00 00 00 00	. \$+.....
7FFDF060	00 00 00 00 00 00 00 00	. \$+.....
7FFDF068	00 00 00 00 00 00 00 00	. \$+.....
7FFDF070	00 00 00 00 00 00 00 00	. \$+.....
7FFDF078	00 00 00 00 00 00 00 00	. \$+.....
7FFDF080	00 00 00 00 00 00 00 00	. \$+.....
7FFDF088	00 00 00 00 00 00 00 00	. \$+.....

Command	? fs:[1]	HEX: 12FF - DEC: 4863 - ASCII: 0x00
---------	----------	-------------------------------------

Por lo tanto:

Fs: [0] en si es el contenido en mi maquina de 7ffdf000

Fs: [1] en si es el contenido en mi maquina de 7ffdf001

Fs: [18] en si es el contenido en mi maquina de 7ffdf018

Pues ese es el valor que lee en la primera línea la api, si recordamos

```
00401000 64:A1 180000 MOV EAX,DWORD PTR FS:[18]
00401006 8B40 30 MOV EAX,DWORD PTR DS:[EAX*30]
00401009 0FB640 02 MOVZX EAX, BYTE PTR DS:[EAX+2]
0040100D 90 NOP
0040100F 9A NOP
```

O sea veamos el contenido de 7ffdf018 o sea FS:[18]

Address	Hex dump	ASCII	
7FFDF000	E0 FF 12 00 00 00 13 00	0 \$...!!	
7FFDF008	00 D0 12 00 00 00 00 00	. \$ \$... ..	
7FFDF010	00 1E 00 00 00 00 00 00	. ^... ..	
7FFDF018	00 F0 FD 7F 00 00 00 00	- 2 0... ..	
7FFDF020	E8 09 00 00 00 00 00 00	8 ... 2... ..	
7FFDF028	00 00 00 00 00 00 00 00	
7FFDF030	00 C0 FD 7F 10 35 00 00	. L 2 0 \$ \$... ..	
7FFDF038	00 00 00 00 00 00 00 00	
7FFDF040	00 C7 2B E3 00 00 00 00	\$ \$+0... ..	
7FFDF048	00 00 00 00 00 00 00 00	
7FFDF050	00 00 00 00 00 00 00 00	
7FFDF058	00 00 00 00 00 00 00 00	
7FFDF060	00 00 00 00 00 00 00 00	
7FFDF068	00 00 00 00 00 00 00 00	
7FFDF070	00 00 00 00 00 00 00 00	
7FFDF078	00 00 00 00 00 00 00 00	
7FFDF080	00 00 00 00 00 00 00 00	
7FFDF088	00 00 00 00 00 00 00 00	

0012FFC4	7C816D4F	RETURN
0012FFC8	7C920738	ntdll.7
0012FFCC	FFFFFFFF	
0012FFD0	7FFDC000	
0012FFD4	8054A938	
0012FFD8	0012FFC8	
0012FFDC	84543DA8	
0012FFE0	FFFFFFFF	End of
0012FFE4	7C8399F3	SE hanc
0012FFE8	7C816D58	kernel3
0012FFEC	00000000	
0012FFF0	00000000	
0012FFF4	00000000	
0012FFF8	00401000	Crackme
0012FFFC	00000000	

Command	? fs:[18]	HEX: 7FFDF000 - DEC: 2147348480 - ASCII: 0x00
---------	-----------	---

Es ese valor, o sea que el valor que guarda el TIB en FS : [18] es el valor que OLLYDBG nos mostraba en los registros el puntero al inicio del TIB, que lo almacena allí.

O sea que en la primera línea, solo mueva EAX el puntero al inicio del TIB, para enterarse donde esta ubicado en tu maquina, y eso esta en fs: [18] por lo tanto si ejecuto la línea con f8, moverá ese valor a EAX.

Registers (FPU)	
EAX	7FFDF000
ECX	0012FFB0
EDX	7C91EB94 ntdll.
EBX	7FFDC000
ESP	0012FFC4
EBP	0012FFF0
ESI	FFFFFFFF
EDI	7C920738 ntdll.
EIP	00401006 Crackme
C 0	ES 0023 32bit
P 1	CS 001B 32bit

Bueno ahora en EAX ya tenemos el inicio del TIB

Address	Hex dump	ASCII
7FFDF000	E0 FF 12 00 00 00 13 00	0 0...!!
7FFDF008	00 00 12 00 00 00 00 00	.S#.....
7FFDF010	00 1E 00 00 00 00 00 00	.A.....
7FFDF018	00 F0 FD 7F 00 00 00 00	..-2A.....
7FFDF020	E8 09 00 00 A8 0A 00 00	b...c.....
7FFDF028	00 00 00 00 00 00 00 00
7FFDF030	00 C0 FD 7F 1D 05 00 00	..L2A#.....
7FFDF038	00 00 00 00 00 00 00 00
7FFDF040	00 C7 2B E3 00 00 00 00	.S\$+0.....
7FFDF048	00 00 00 00 00 00 00 00
7FFDF050	00 00 00 00 00 00 00 00
7FFDF058	00 00 00 00 00 00 00 00
7FFDF060	00 00 00 00 00 00 00 00

Ahora en la siguiente linea

00401000	64:A1 180000	MOV EAX,DWORD PTR FS:[18]
00401006	8B40 30	MOV EAX,DWORD PTR DS:[EAX+30]
00401009	0FB640 02	MOVZX EAX,BYTE PTR DS:[EAX+2]
0040100D	90	NOP
0040100E	90	NOP

A EAX le suma 30 o sea seria en mi maquina 7ffdf000 + 30 =7ffdf030

El contenido de dicho valor es fs:[30]

Address	Hex dump	ASCII
7FFDF000	E0 FF 12 00 00 00 13 00	0 0...!!
7FFDF008	00 00 12 00 00 00 00 00	.S#.....
7FFDF010	00 1E 00 00 00 00 00 00	.A.....
7FFDF018	00 F0 FD 7F 00 00 00 00	..-2A.....
7FFDF020	E8 09 00 00 A8 0A 00 00	b...c.....
7FFDF028	00 00 00 00 00 00 00 00
7FFDF030	00 C0 FD 7F 1D 05 00 00	..L2A#.....
7FFDF038	00 00 00 00 00 00 00 00
7FFDF040	00 C7 2B E3 00 00 00 00	.S\$+0.....
7FFDF048	00 00 00 00 00 00 00 00
7FFDF050	00 00 00 00 00 00 00 00
7FFDF058	00 00 00 00 00 00 00 00
7FFDF060	00 00 00 00 00 00 00 00
7FFDF068	00 00 00 00 00 00 00 00
7FFDF070	00 00 00 00 00 00 00 00
7FFDF078	00 00 00 00 00 00 00 00
7FFDF080	00 00 00 00 00 00 00 00
7FFDF088	00 00 00 00 00 00 00 00

Command: ? fs:[30] HEX: 7FFDC000 - DEC: 2147336

O sea mueve a EAX el contenido de 7ffdf030 o fs:[30], dicho contenido en mi maquina es 7ffdc000 no confundir con el inicio del TIB no es el mismo valor.

Registers (FPU)	
EAX	7FFDC000
ECX	0012FFB0
EDX	7C91EB94 ntdll
EBX	7FFDC000
ESP	0012FFC4
EBP	0012FFF0
ESI	FFFFFFFF
EDI	7C920738 ntdll
EIP	00401009 Crac
C 0	ES 0023 32bi
P 1	CS 001B 32bi
A 0	SS 0023 32bi

Pues este es un puntero a otra cosa, vayamos a esa dirección en el dump

Address	Hex dump	ASCII
7FFDC000	00 00 01 00 FF FF FF FF	..0.....
7FFDC008	00 00 40 00 A0 1E 24 00	..@. \$ \$..
7FFDC010	00 00 02 00 00 00 00 00	..0.....
7FFDC018	00 00 14 00 C0 E4 98 7C	..@.L\$@!
7FFDC020	05 10 91 7C ED 10 91 7C	\$ \$! \$ \$!
7FFDC028	01 00 00 00 80 29 D1 77	@...C)0w
7FFDC030	00 00 00 00 00 00 00 00
7FFDC038	00 00 00 00 00 00 00 00
7FFDC040	80 E4 98 7C 03 00 00 00	C\$@! \$..
7FFDC048	AA AA AA AA AA AA 6F 7Fn

Allí la vemos la siguiente y ultima línea

00401009	0FB640 02	MOUZ' EAX, BYTE PTR DS:[EAX+2]
0040100D	90	NOP
0040100F	9A	NOP

Le suma 2 a EAX o sea

$$7FFDC000 + 2 = 7FFDC002$$

Y mueve el byte que contiene esa dirección a EAX y dicho byte es el byte buscado, y ya lo hallamos manualmente es en mi maquina el que esta en la dirección 7FFDC002 (en su maquina puede cambiar la dirección realizar el mismo proceso para hallarlo)

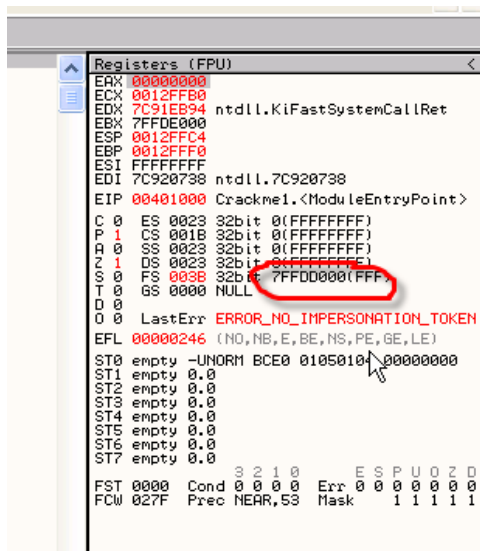
Address	Hex dump	ASCII
7FFDC000	00 00 01 00 FF FF FF FF	...0.
7FFDC008	00 00 40 00 A0 1E 24 00	...0.â\$.
7FFDC010	00 00 02 00 00 00 00 00	...0....
7FFDC018	00 00 14 00 C0 E4 98 7C	...0.ÿ!
7FFDC020	05 10 91 7E ED 10 91 7C	...0.ÿ!
7FFDC028	01 00 00 00 80 29 D1 77	0...0)0w
7FFDC030	00 00 00 00 00 00 00 00
7FFDC038	00 00 00 00 00 00 00 00
7FFDC040	80 E4 98 7C 03 00 00 00	0ÿ!0...A

Allí el sistema guardo el byte 1 que lee la api, inclusive en su misma maquina puede variar de posición al reiniciar, pero siempre lo pueden hallar rápidamente con este método, reiniciemos el crackme.

00401000	6A 00	PUSH 0	
00401002	E8 6F040000	CALL <JMP.&kernel32.GetModuleHandleA>	
00401007	A3 50314000	MOV DWORD PTR DS:[403150],EAX	
0040100C	E8 5F040000	CALL <JMP.&kernel32.GetCommandLineA>	
00401011	E8 72040000	CALL <JMP.&comctl32.InitCommonControls>	InitCommonCo
00401016	6A 0A	PUSH 0A	Arg4 = 00000
00401018	FF35 54314000	PUSH DWORD PTR DS:[403154]	Arg3 = 00000
0040101E	6A 00	PUSH 0	Arg2 = 00000
00401020	FF35 50314000	PUSH DWORD PTR DS:[403150]	Arg1 = 00000
00401026	E8 06000000	CALL Crackme1.00401031	Crackme1.004
0040102B	50	PUSH EAX	ExitCode
0040102C	E8 39040000	CALL <JMP.&kernel32.ExitProcess>	ExitProcess

Esta vez no escribimos nada a ver si localizamos el byte a mano y si lo ponemos a cero a mano, y veremos si el programa corre, pues la api leerá dicho byte y si esta a cero, pues devolverá cero, jeje.

1)BUSCAR EL INICIO DEL TIB en la ventana de los registros del OLLYDBG



Ver el TIB en el dump (ya no esta en la misma posición que arranco la vez anterior)

Address	Hex dump	ASCII
7FFD0000	E0 FF 12 00 00 00 13 00	0 \$...!!
7FFD0008	00 D0 12 00 00 00 00 00	. \$ \$.....
7FFD0010	00 1E 00 00 00 00 00 00	. \$.....
7FFD0018	00 D0 FD 7F 00 00 00 00	. \$ \$.....
7FFD0020	8C 0F 00 00 34 0B 00 00	i \$..4 \$..
7FFD0028	00 00 00 00 00 00 00 00
7FFD0030	00 E0 FD 7F 1D 05 00 00	. \$ \$ \$ \$..
7FFD0038	00 00 00 00 00 00 00 00
7FFD0040	D0 C7 2B E3 00 00 00 00	\$ \$+0.....
7FFD0048	00 00 00 00 00 00 00 00
7FFD0050	00 00 00 00 00 00 00 00
7FFD0058	00 00 00 00 00 00 00 00

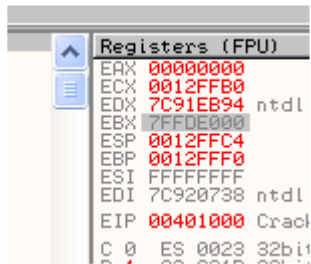
Busco fs: [30] o sea le sumo 30 al inicio del TIB y hallo su contenido

Address	Hex dump	ASCII
7FFD0000	E0 FF 12 00 00 00 13 00	0 \$...!!
7FFD0008	00 D0 12 00 00 00 00 00	. \$ \$.....
7FFD0010	00 1E 00 00 00 00 00 00	. \$.....
7FFD0018	00 D0 FD 7F 00 00 00 00	. \$ \$.....
7FFD0020	8C 0F 00 00 34 0B 00 00	i \$..4 \$..
7FFD0028	00 00 00 00 00 00 00 00
7FFD0030	00 E0 FD 7F 1D 05 00 00	. \$ \$ \$ \$..
7FFD0038	00 00 00 00 00 00 00 00
7FFD0040	D0 C7 2B E3 00 00 00 00	\$ \$+0.....
7FFD0048	00 00 00 00 00 00 00 00
7FFD0050	00 00 00 00 00 00 00 00
7FFD0058	00 00 00 00 00 00 00 00
7FFD0060	00 00 00 00 00 00 00 00

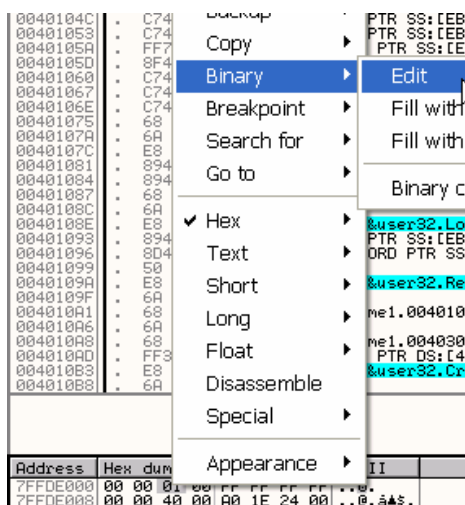
El contenido es 7ffde000 ese lo busco en el dump

Address	Hex dump	ASCII
7FFDE000	00 00 01 00 FF FF FF FF	.. \$.
7FFDE008	00 00 40 00 A0 1E 24 00	.. \$.. \$ \$ \$.
7FFDE010	00 00 02 00 00 00 00 00	.. \$.....
7FFDE018	00 00 14 00 C0 E4 98 7C	.. \$.. \$ \$ \$.
7FFDE020	05 10 91 7C ED 10 91 7C	\$ \$ \$ \$ \$ \$ \$ \$.
7FFDE028	01 00 00 00 80 29 01 77	\$ \$.. \$ \$ \$ \$.
7FFDE030	00 00 00 00 00 00 00 00
7FFDE038	00 00 00 00 00 00 00 00
7FFDE040	80 E4 98 7C 03 00 00 00	\$ \$ \$ \$ \$ \$ \$ \$.
7FFDE048	00 00 00 00 00 00 6F 7F \$ \$.
7FFDE050	00 00 6F 7F 88 06 6F 7F \$ \$ \$ \$.
7FFDE058	00 00 FB 7F 00 10 FC 7F	.. \$ \$ \$ \$.

Y le sumo 2 y allí esta el byte (muchos dirán para que tamaño explicación si los programas cuando arrancan en EBX, esta siempre el puntero a esta zona, jeje o sea EBX= fs:[30], jeje)

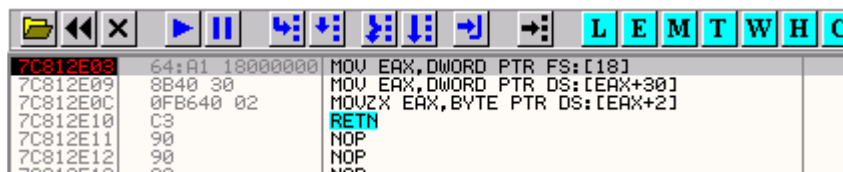


Bueno pero es bueno saber el método entero, porque por ahí uno no esta al inicio del programa y quiere hallar el byte a mano y no sabe que valor tenia EBX en el arranque, pero bueno este es el método completo, ahora pongamos a cero el byte.

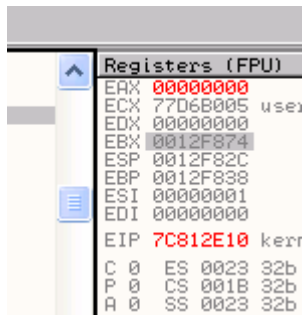


Address	Hex dump	ASCII
7FFDE000	00 00 00 00 FF FF FF FF	...
7FFDE008	00 00 40 00 A0 1E 24 00	...@.34\$.
7FFDE010	00 00 02 00 00 00 00 00	...0....
7FFDE018	00 00 14 00 C0 E4 98 7C	...0.48!
7FFDE020	05 10 91 7C ED 10 91 7C	510917CED10917C
7FFDE028	01 00 00 00 80 29 D1 77	0...0)0w
7FFDE030	00 00 00 00 00 00 00 00
7FFDE038	00 00 00 00 00 00 00 00

Bueno pongamos un BP IsDebuggerPresent a ver si funciona



Traceemos hasta el RET



Y vemos que EAX que es el valor de retorno de la api vale cero o sea que al cambiar el bytea mano, el programa cree que no esta siendo debuggeado tanto por el uso de la api o por la lectura del byte sin la api.

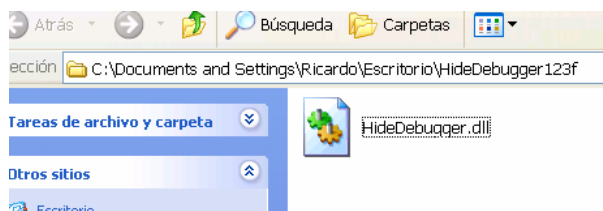


Y por supuesto corre perfectamente.

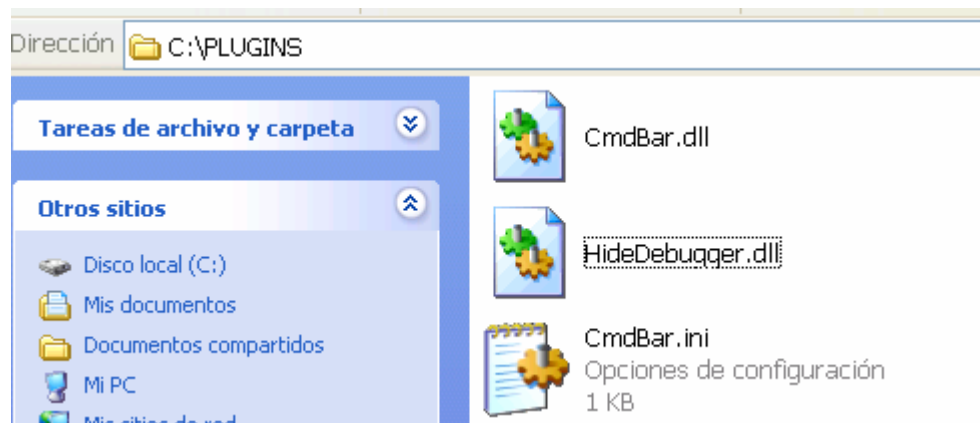
Obviamente hay muchos plugins que hacen este trabajo, uno de ellos es el plugin HideDebbuger 1.23f que pueden descargar desde mi http aquí

http://www.ricnar456.dyndns.org/HERRAMIENTAS/L-M-N-%D1-O-P/Plugins_Olly/HideDebugger123f.zip

Una vez descargado copian la dll en la carpeta de plugins

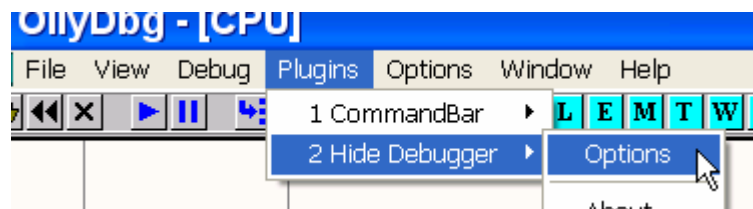


COPIANDO jeje

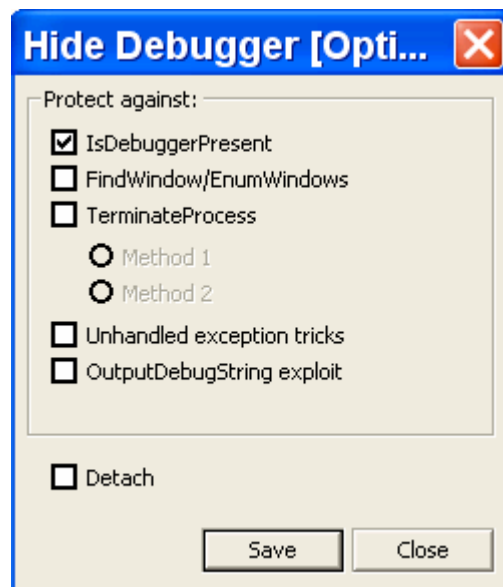


Allí esta debajo del otro

Ahora reinicio el OLLYDBG

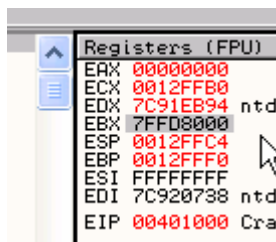


Veo que en las opciones del plugin



Viene ya marcado para evitar que detecte por el byte de IsDebuggerPresent o sea hace el trabajo que nosotros hicimos a mano lo pone a cero siempre, apretamos SAVE y las otras opciones por ahora las dejamos deshabilitadas, a medida que las vayamos estudiando, las veremos.

Arranco de nuevo el crackme y me fijo ya que EBX apunta a la zona del byte



Hago EBX-FOLLOW IN DUMP

Address	Hex dump	ASCII
7FFD8000	00 00 00 00 FF FF FF FF
7FFD8008	00 00 40 00 A0 1E 24 00	..@.â\$.
7FFD8010	00 00 02 00 00 00 00 00	..@.....
7FFD8018	00 00 14 00 C0 E4 98 7C	..@.L&ÿ!
7FFD8020	05 10 91 7C ED 10 91 7C	â!æ!ÿ!æ!
7FFD8028	01 00 00 00 80 29 01 77	@...ÿ!ðw
7FFD8030	00 00 00 00 00 00 00 00
7FFD8038	00 00 00 00 00 00 00 00

Y vemos que el plugin hizo todo, lo puso a cero acabando con este tipo de protección, quizás hubiera sido mas sencillo hacer esto solo, pero creo que comprender el porque de las cosas es muy importante por eso la explicación completa creo que viene bien.

Hasta la parte 20.

Ricardo Narvaja

21 de diciembre de 2005