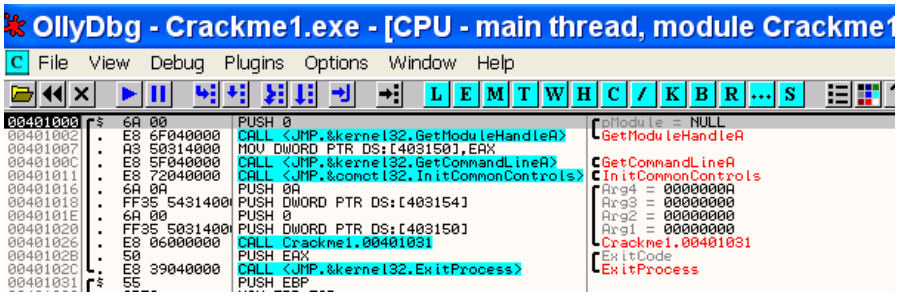
INTRODUCCION AL CRACKING EN OLLYDBG parte 19

DETECCION DEL OLLYDBG – IsDebuggerPresent

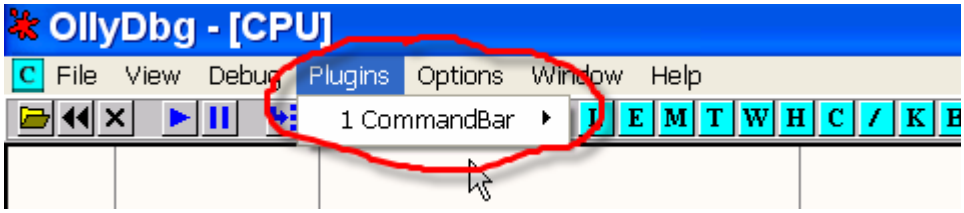
Como dijimos la parte anterior, nos concentraremos en la forma de evitar manualmente y por medio de plugins la detección de OLLYDBG por medio del programa que esta siendo debuggeado, de forma tal de poder trabajarlo tranquilamente ya que la mayoría de los programas, cuando detectan que hay un debugger 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.

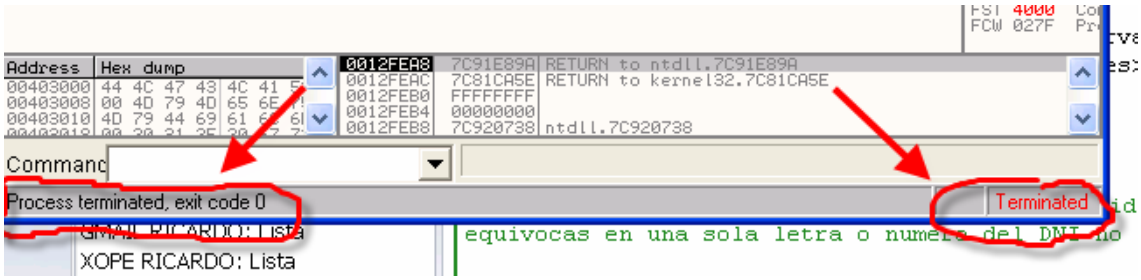
Para ello utilizaremos el Crackme1.exe que esta adjunto a esta leccion, lo arrancamos en OLLYDBG.



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?

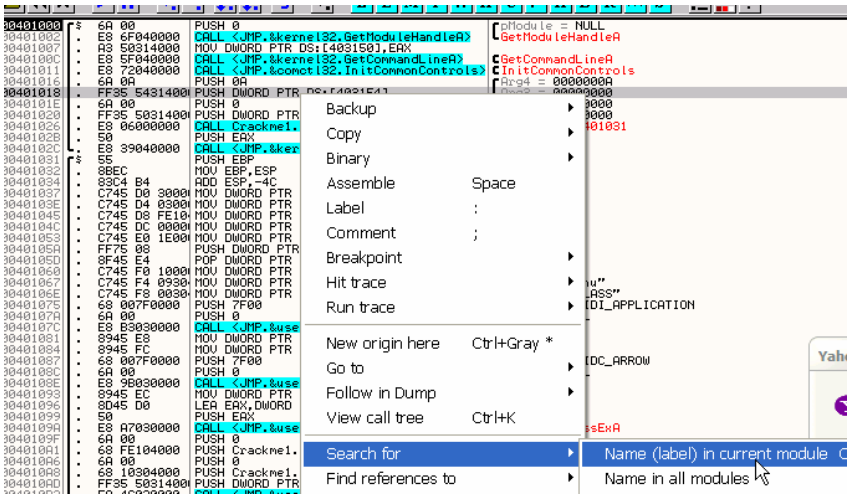


Si corremos el crackme con F9 vemos que no solo no abre la ventana del crackme si no que se cierra el programa.

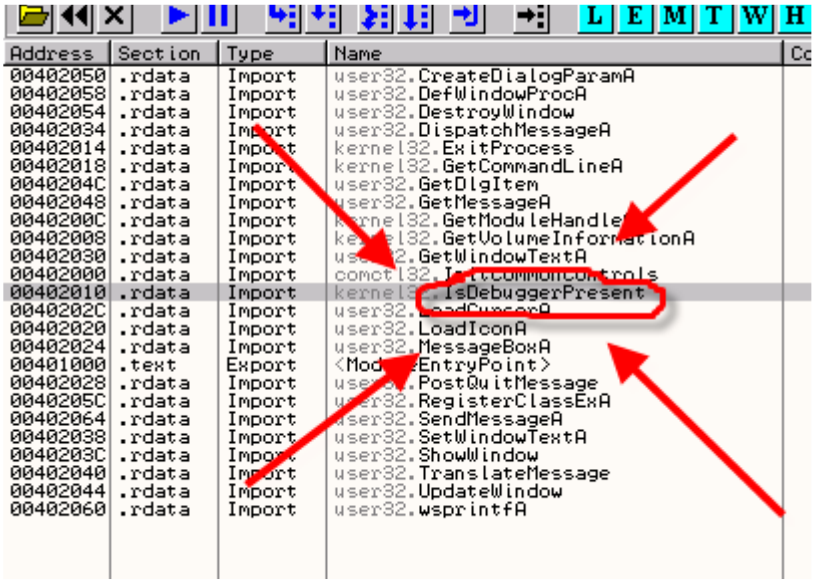


OLLYDBG nos indica en la parte inferior que el programa termino, y ni siquiera vimos la ventana del crackme, jeje, que ocurre aquí, muy sencillo el crackme utiliza el método mas conocido para detectar que esta siendo debuggeado que es llamar a la api IsDebuggerPresent.

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 ..

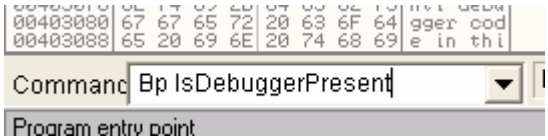


Vemos la lista de apis usadas

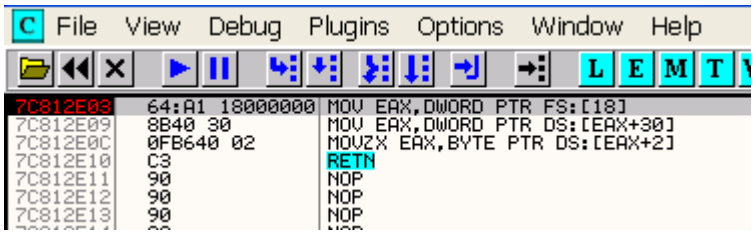


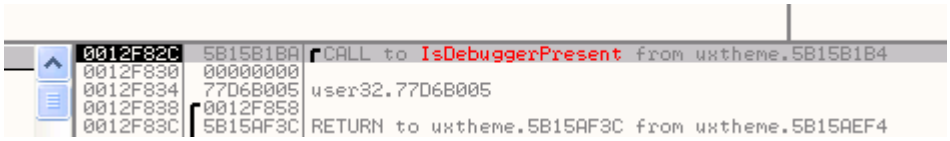
Y si parece que la usa jeje

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.

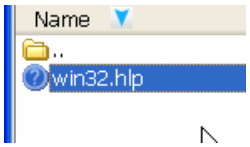
http://www.ricnar456.dyndns.org

user:hola
pass:hola

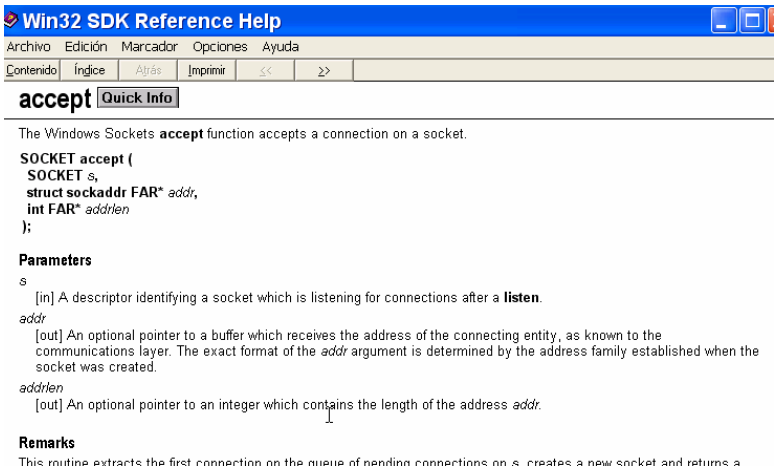
y allí en la carpeta HERRAMIENTAS/V-W-X-Y-Z esta el archivo que contiene la especificación de las apis mas importante publicada por Microsoft, no están todas pero si la gran mayoría, el único problema es que esta en ingles, y es un poquito complicado de entender para el que no esta acostumbrado, pero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

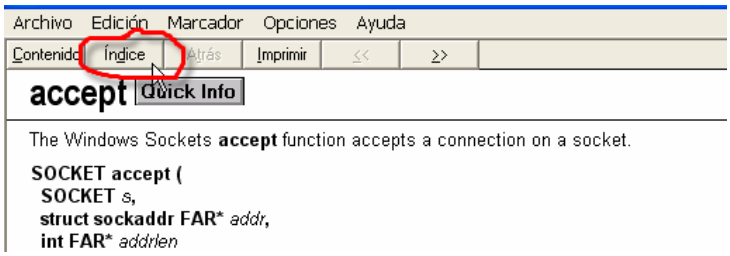
user:hola
pass:hola



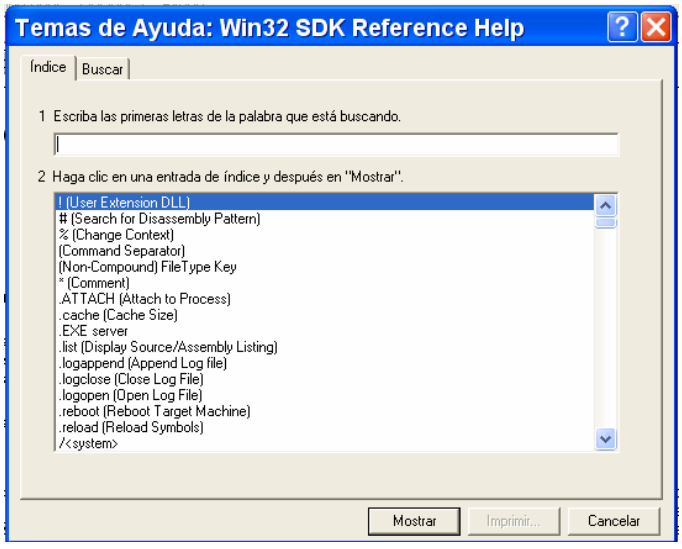
El archivo zip, contiene un archivo de extensión HLP que es el que nos interesa lo ejecutamos.



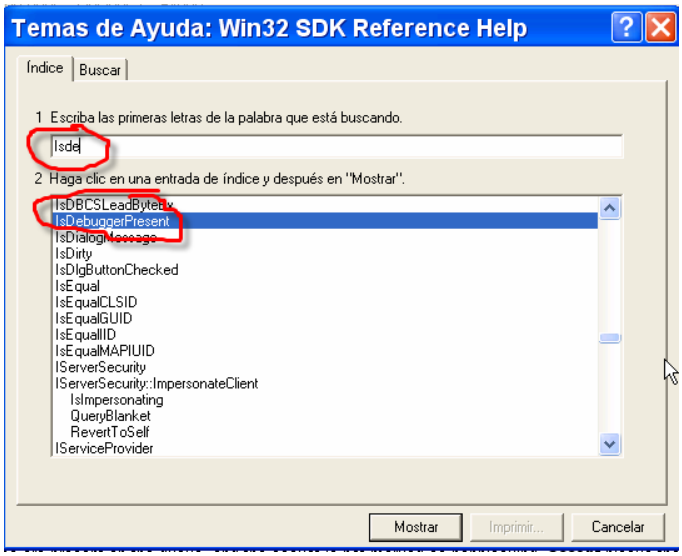
Allí se abrió, ahora como buscamos la api que nos interesa?



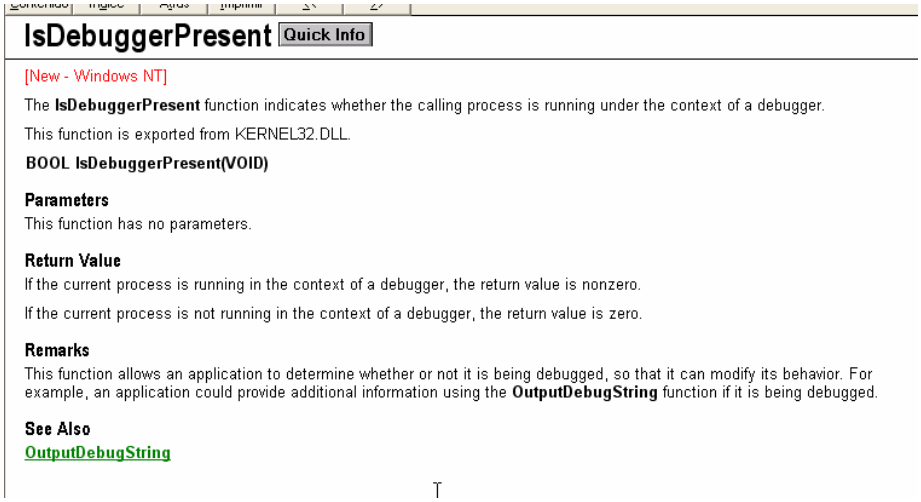
Vamos a INDICE



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

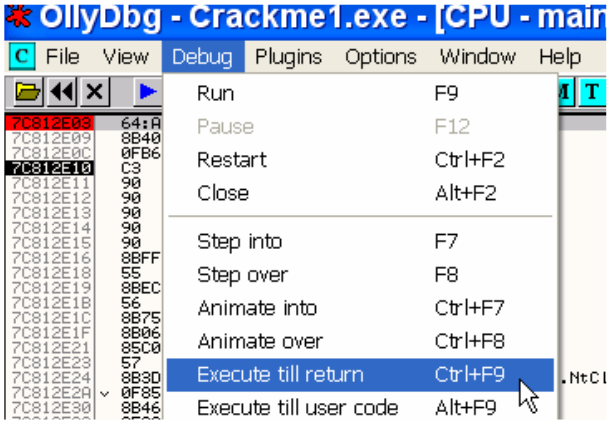


Bueno allí esta la aclaración y traducimos

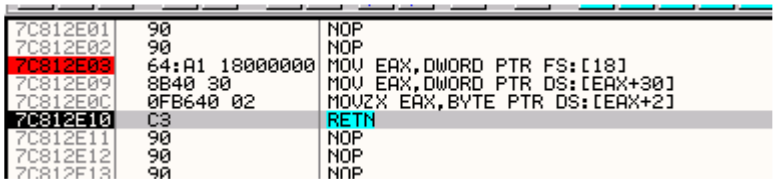
La api IsDebuggerPresent indica cuando el proceso que la llamo, esta bajo el contexto de un debugger o sea siendo debuggeado.

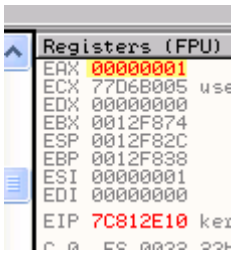
Luego dice que esta api o función es exportada por la KERNEL32.dll, que no tiene parámetros como vimos, y en RETURN VALUES o sea en valores de retorno, dice que si el programa esta siendo debuggeado el retorno es un valor distinto de cero, y si no lo esta es cero.

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



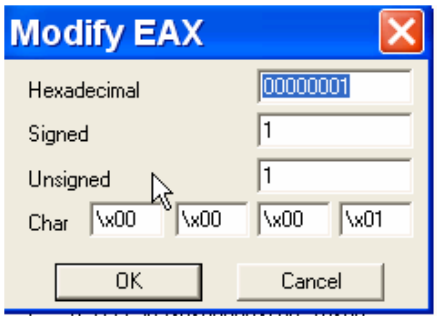
Allí llegue al RET si veo los registros

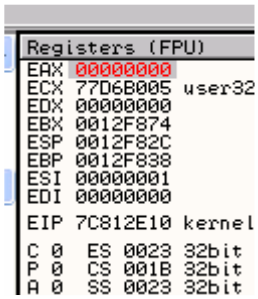




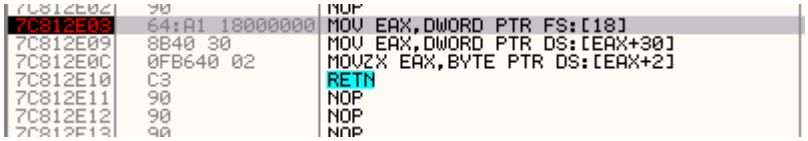
El que esta en rojo o sea que cambio es EAX, como casi todas las apis, devuelven en EAX el valor de retorno, así que EAX es igual a 1 lo cual le dice al programa que esta siendo debuggeado.

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

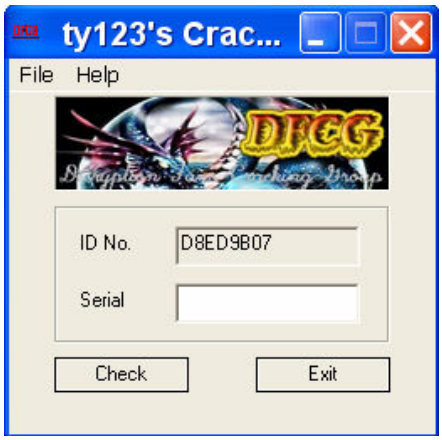




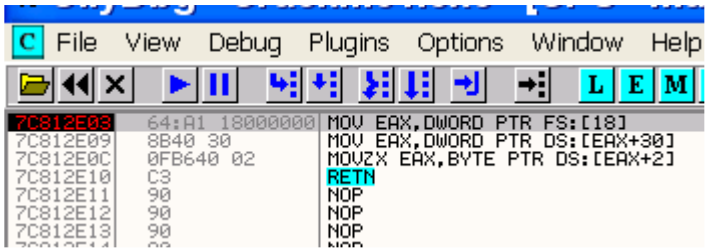
Demos RUN ahora



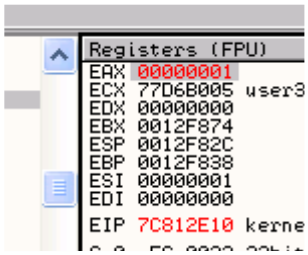
Vemos que vuelve a parar en la api, así que llegamos al ret nuevamente y cambiamos EAX a cero.



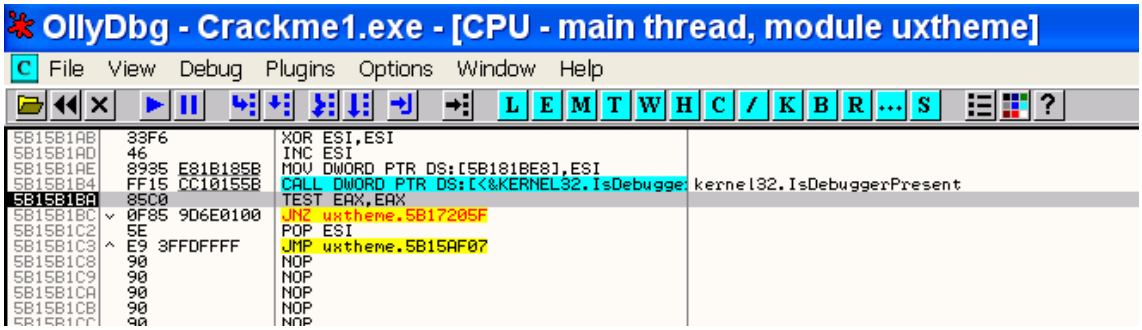
Vemos que el programa arranca, o sea que toda su detección se basa en dicha api, reiniciémoslo y cuando pare en IsDebuggerPresent, veamos que hace con el valor que le devuelve la api.



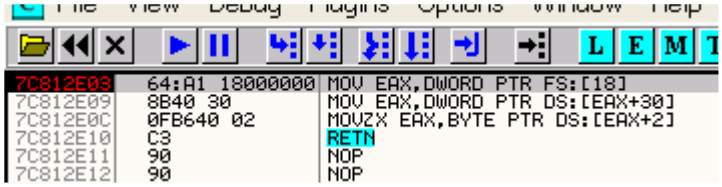
Allí paro nuevamente lleguemos hasta el ret, podemos tracear con f8 pues la api es cortita.



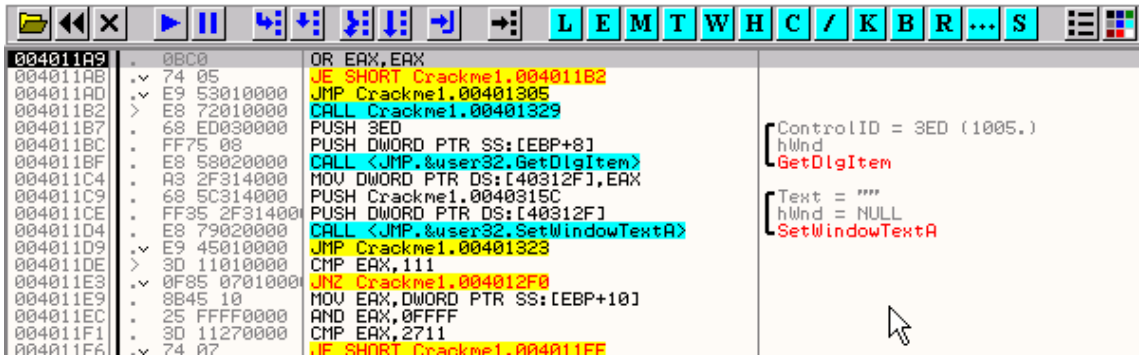
Allí esta, al llegar al RET, vale EAX=1, volvamos al programa con f8 nuevamente.



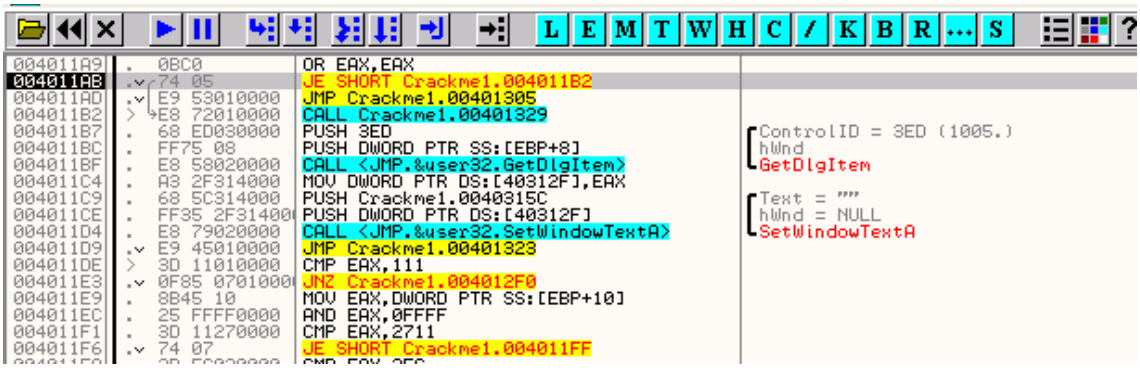
Vemos que la primera vez no es llamado desde el mismo ejecutable si no de una dll que es uxtheme, demos RUN a ver la segunda vez que para en la api.



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

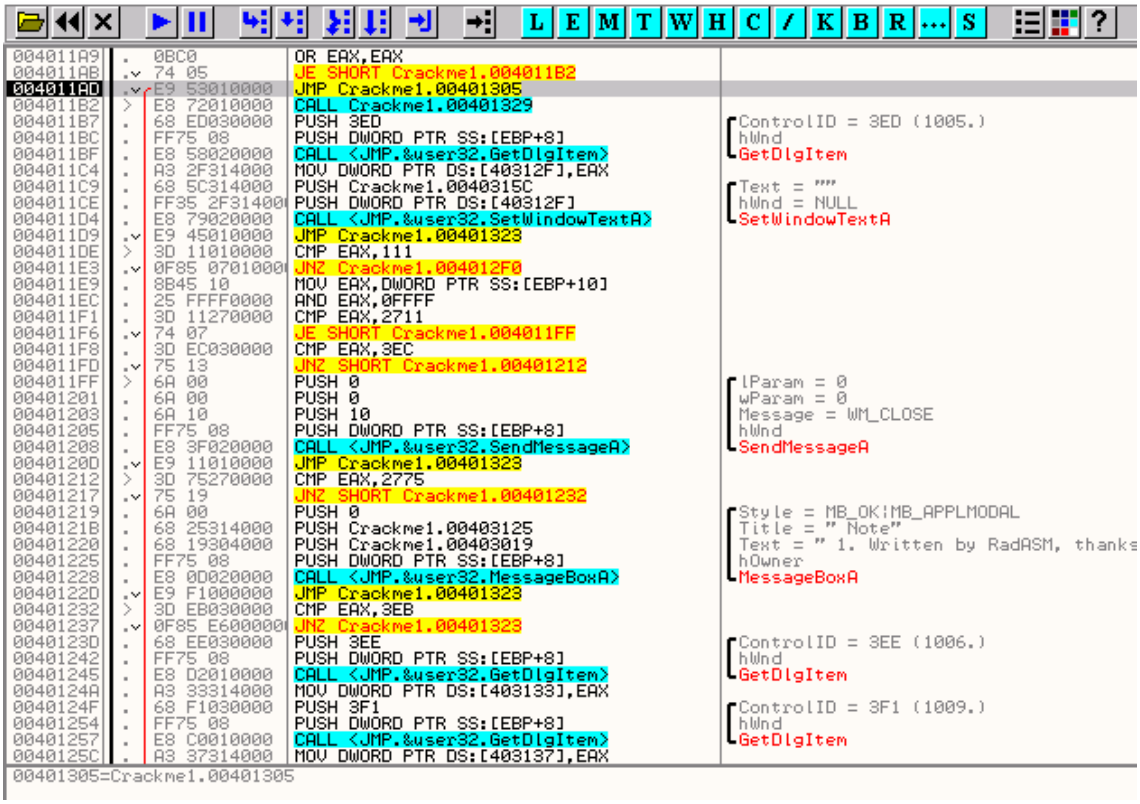


Llega aquí, y allí hay un JE que testea si EAX se cero o no, veamos.

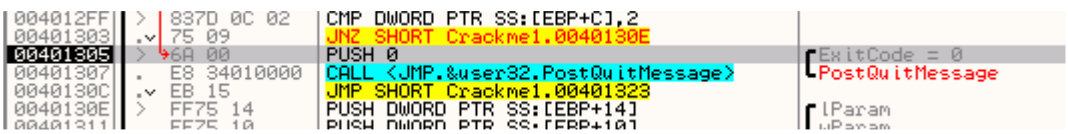


Vemos que el JE al ser EAX diferente de cero no salta, observamos también que si saltara continua la ejecución del programa con GetDlgItem leyendo de la ventana del crackme.

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.



Continuemos traceando a ver donde va



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.

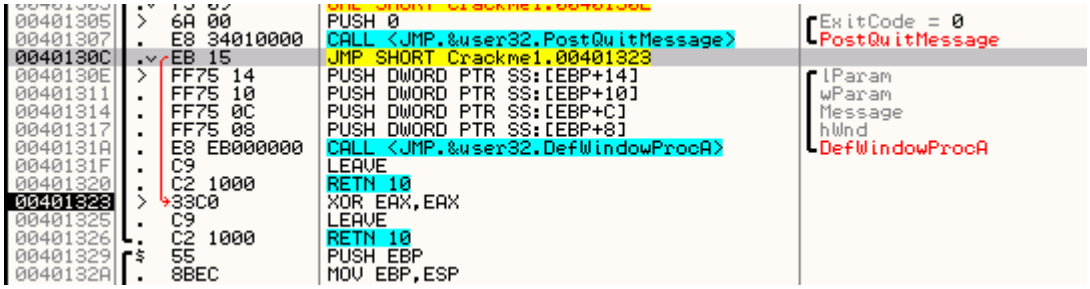
When the thread retrieves the WM_QUIT message from its message queue, it should exit its message loop and return control to Windows. The exit value returned to Windows must be the *wParam* parameter of the WM_QUIT message.

See Also

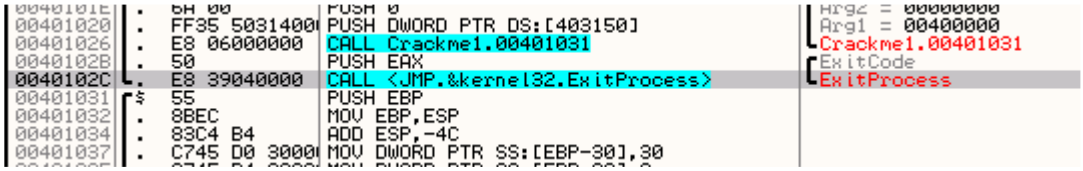
GetMessage, **PeekMessage**, **PostMessage**, WM_DESTROY, WM_QUIT

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.

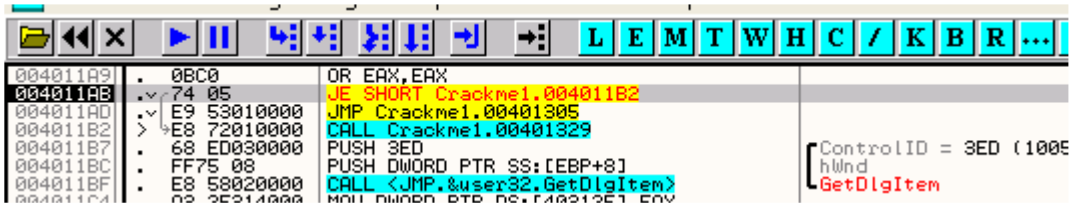
Al volver del mismo



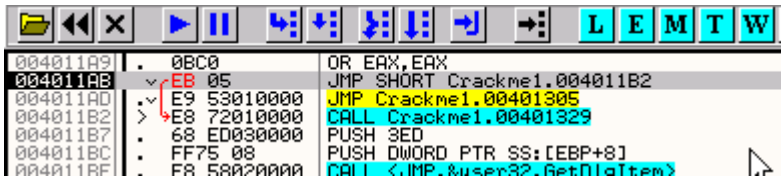
Y si tenemos paciencia de tracear bastante mientras va cerrando todo, observaremos que llegara a la api ExitProcess que es la que cierra el proceso.(si no tienen paciencia le ponen un BP ExitProcess y parara en ella)



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.

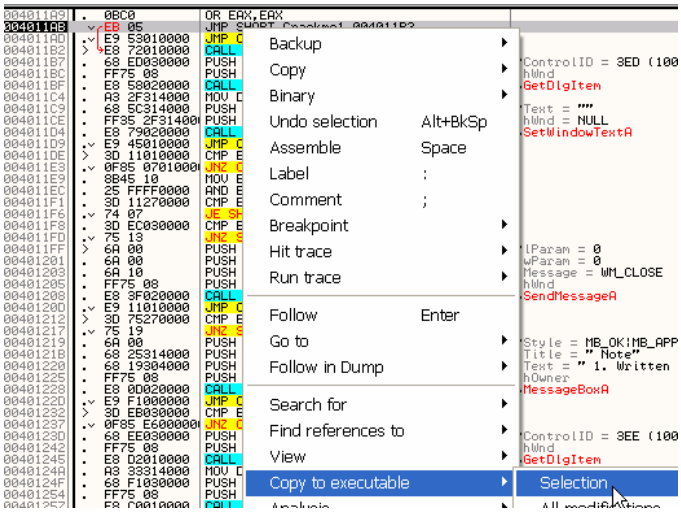


Allí esta y como habíamos visto tenia que saltar siempre, por lo cual lo cambiamos a JMP, tecleando la barra espaciadora y escribiendo JMP 4011b2.

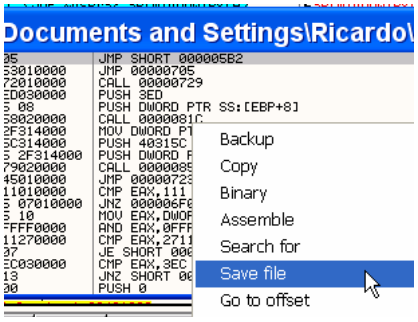


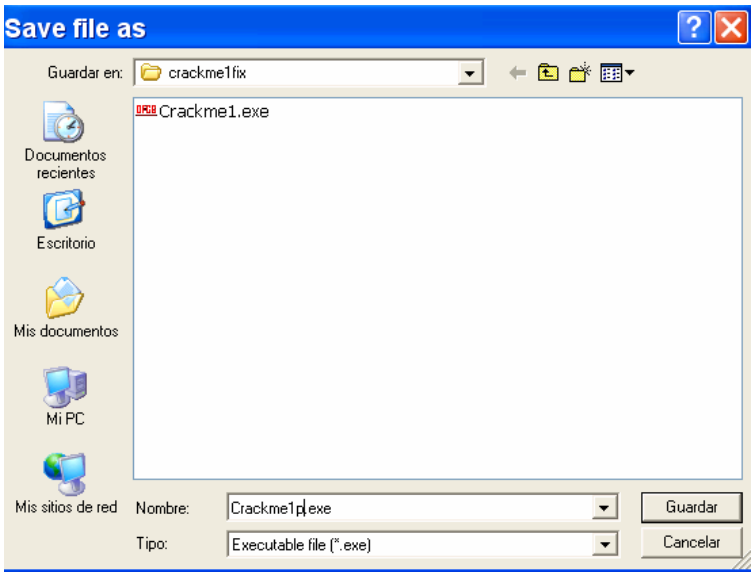
Allí esta, vemos que ahora evita el JMP que va a cerrar el programa y el mismo continua funcionando.

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



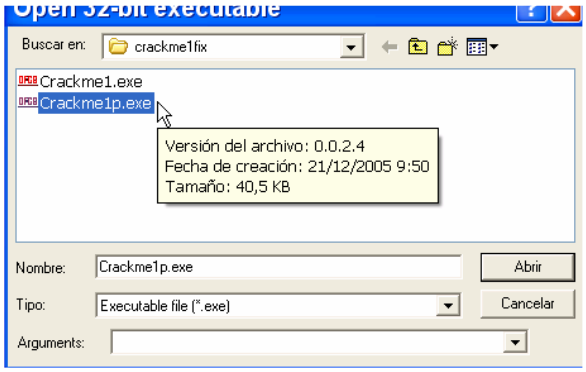
Y en la nueva ventana que se abre hagamos nuevamente click derecho





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

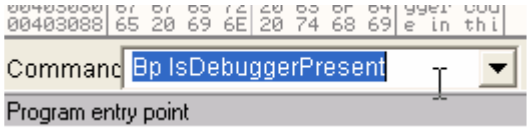
Ahora abramos el parcheado en OLLYDBG para ver si corre en el mismo, sin poner ningún BP ni nada

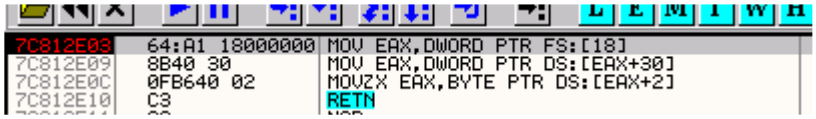


Corre perfectamente, así que ya sabemos hacer correr un programa cambiando EAX al volver de IsDebuggerPresent, también sabemos parchear para que corra y no tengamos que tomarnos el trabajo de hacerlo a mano.

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.



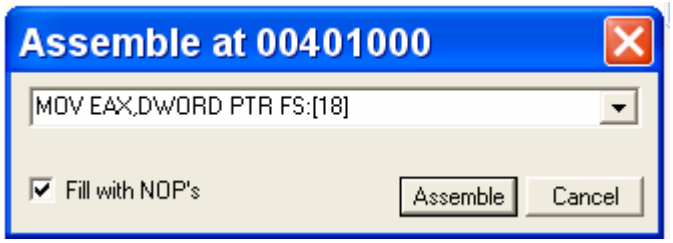


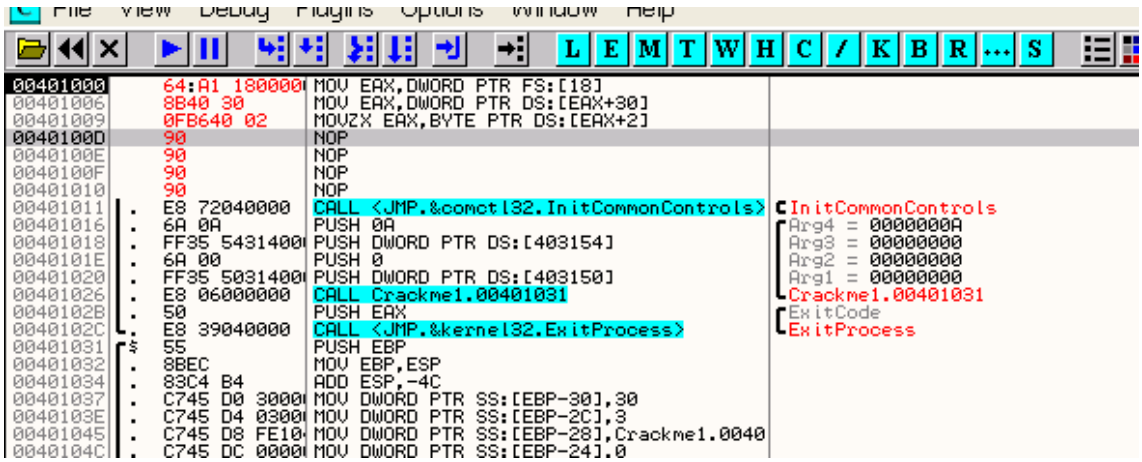
Son tres tristes MOV y con eso la api determina si esta siendo debuggeado el programa o no.

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.

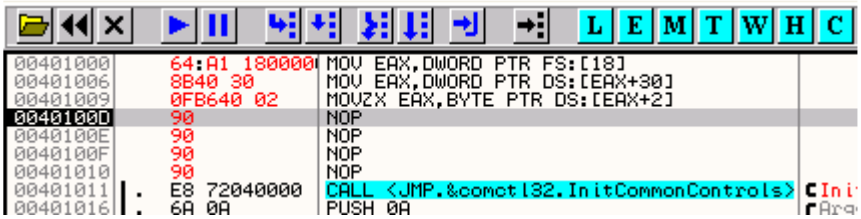
MOV EAX,DWORD PTR FS:[18]
MOV EAX,DWORD PTR DS:[EAX+30]
MOVZX EAX,BYTE PTR DS:[EAX+2]

Podemos copiar y pegar cada línea de aquí.

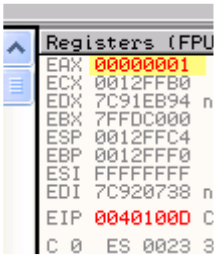




Allí tenemos las tres líneas escritas.



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



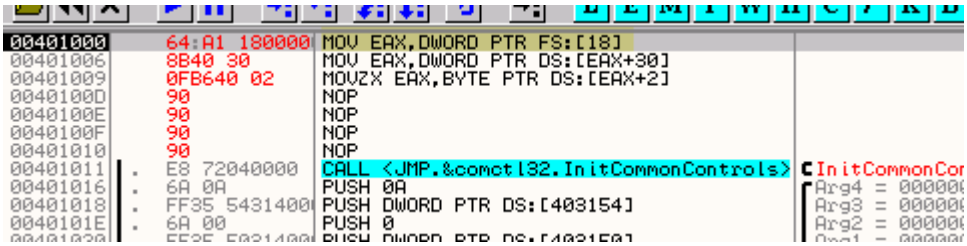
Y obviamente nunca el programa parara en la api IsDebuggerPresent, pues nunca fue llamada, e igual detecto que esta siendo debuggeado.

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

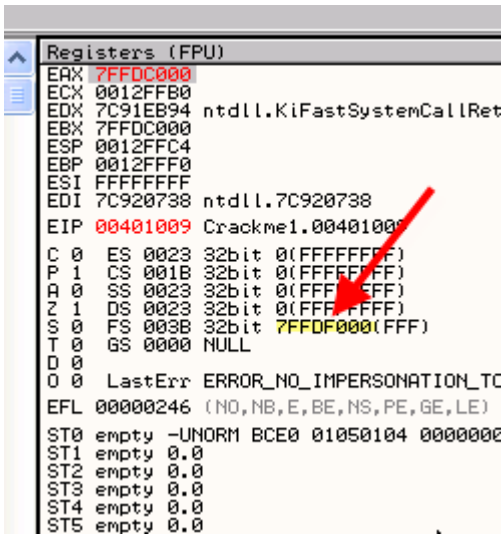
En concreto cuando un programa arranca el sistema guarda en una dirección de memoria determinada un uno si detecta que el programa esta siendo debuggeado y un cero si no lo esta, ese byte es el que leen las tres líneas estas y por supuesto la api también.

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

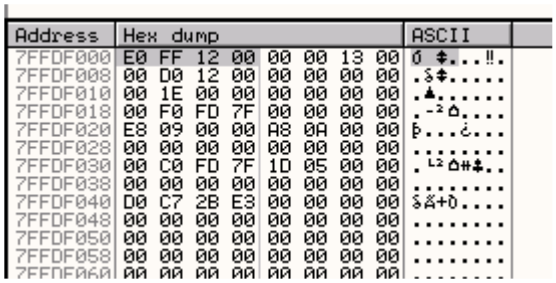
La primera línea es



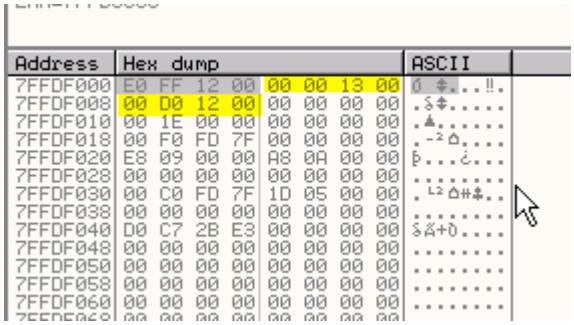
Pues aquí explicaremos lo más sencillo posible algo de teoría, en el OLLYDBG vemos aquí



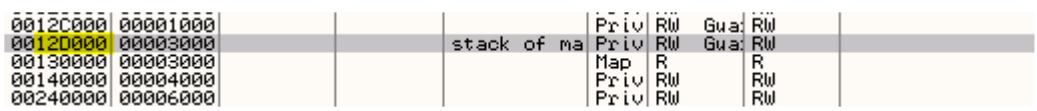
En la ventana de los registros un valor muy importante, no quiero dar nombres difíciles pero ese valor apunta a una estructura que guarda información muy importante sobre el programa que arranco, vayamos a ver en el DUMP dicha dirección (en su maquina puede estar en otra dirección solo vayan a la dirección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



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.

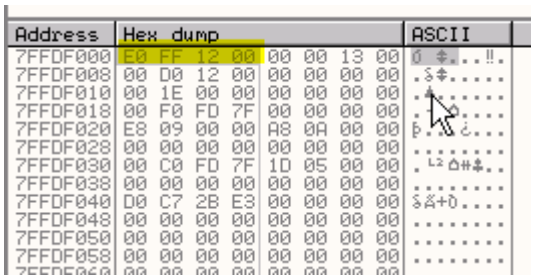


Si vemos en M la sección del stack

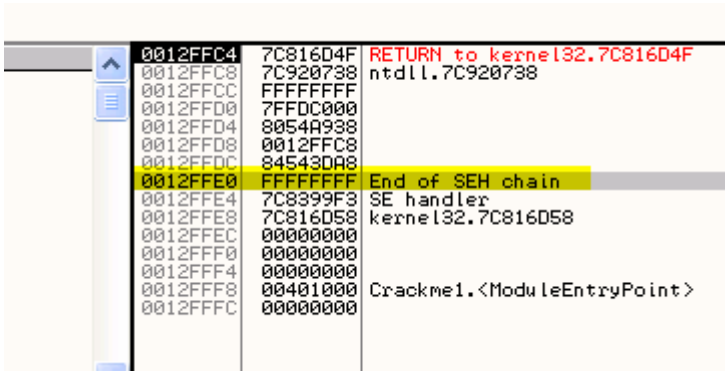


Vemos que comienza en 12d000 y termina justo antes de donde se inicia la sección siguiente en 130000.

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.

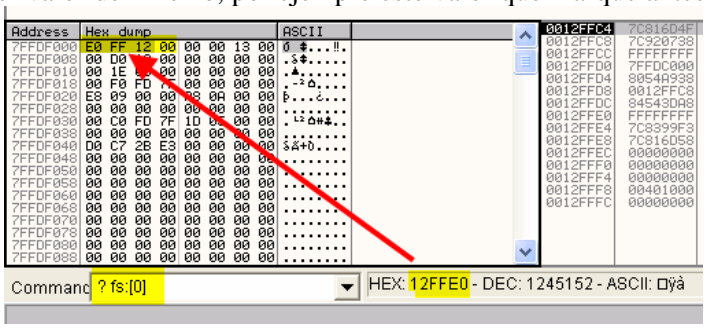


Lo busco en el stack

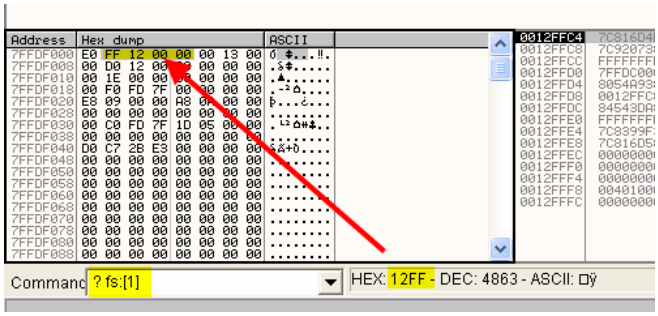


Vemos que esta marcado como END OF SEH CHAIN, por ahora no diremos mas de el, pero si que esta relacionado con las excepciones.

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] .



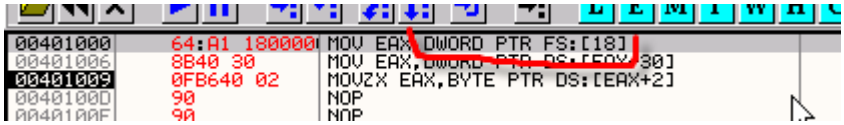
Ese seria Fs: [0], si en la command bar buscamos el valor de fs: [1], ese seria



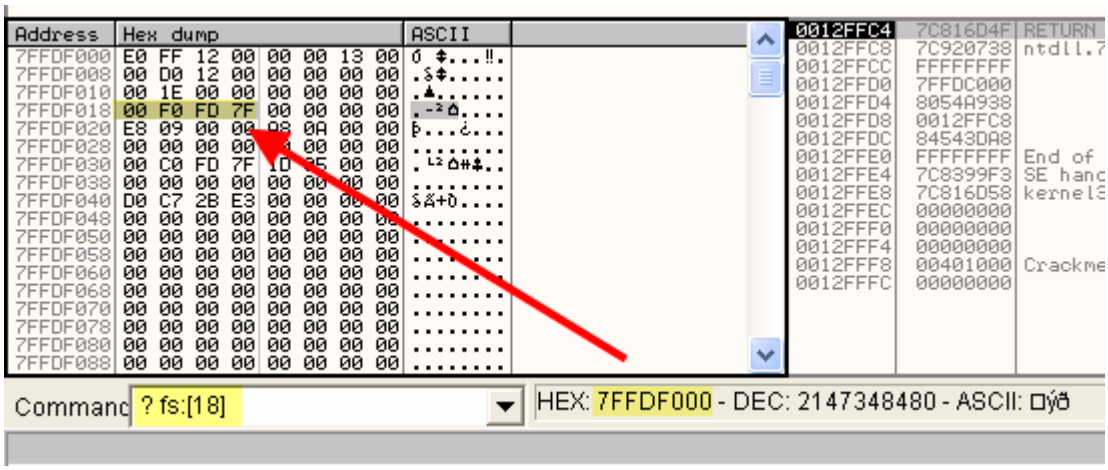
Por lo tanto:

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

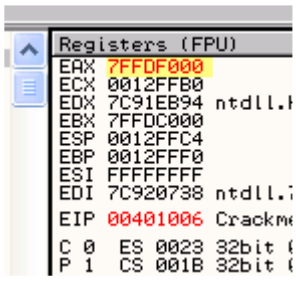


O sea veamos el contenido de 7ffdf018 o sea FS:[18]

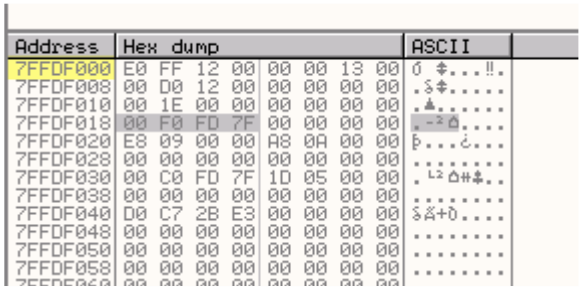


Es ese valor, o sea que el valor que guarda el TIB en FS : [18] es el valor que OLLYDBG nos mostraba en los registros el puntero al inicio del TIB, que lo almacena allí.

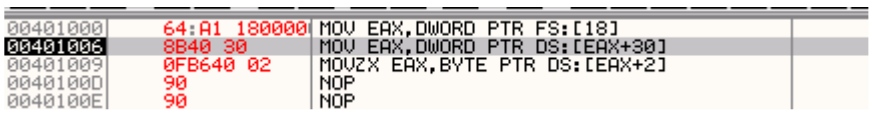
O sea que en la primera línea, solo muevea EAX el puntero al inicio del TIB, para enterarse donde esta ubicado en tu maquina, y eso esta en fs: [18] por lo tanto si ejecuto la línea con f8, moverá ese valor a EAX.



Bueno ahora en EAX ya tenemos el inicio del TIB

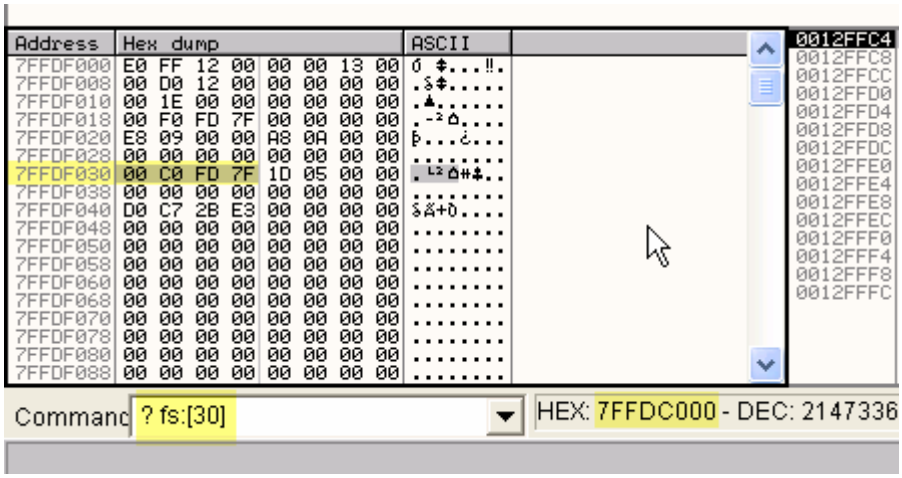


Ahora en la siguiente linea

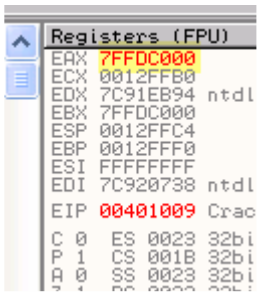


A EAX le suma 30 o sea seria en mi maquina 7ffdf000 + 30 =7ffdf030

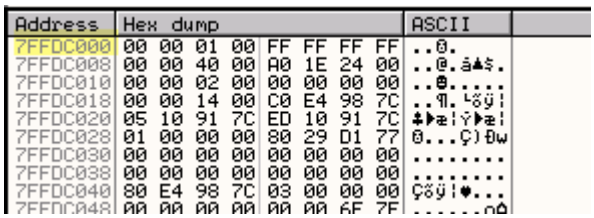
El contenido de dicho valor es fs:[30]



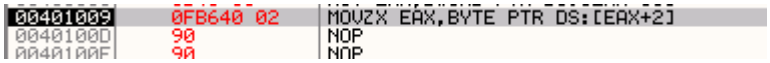
O sea mueve a EAX el contenido de 7ffdf030 o fs:[30], dicho contenido en mi maquina es 7ffdc000 no confundir con el inicio del TIB no es el mismo valor.



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



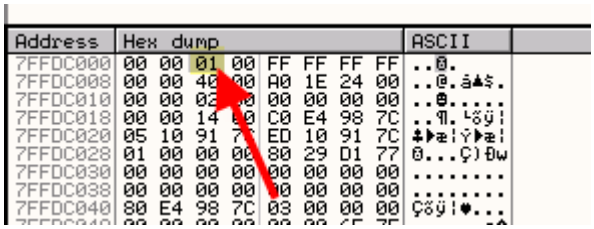
Allí la vemos la siguiente y ultima línea



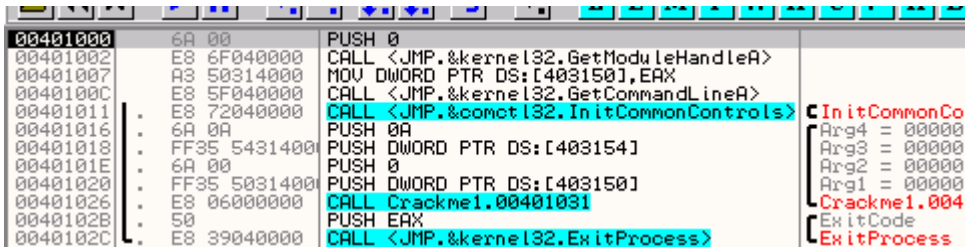
Le suma 2 a EAX o sea

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)

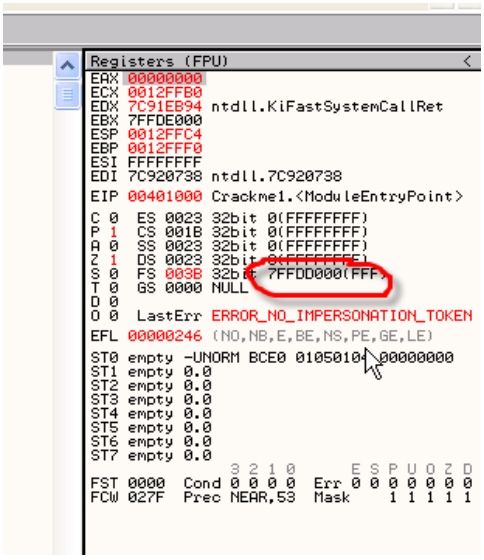


Allí el sistema guardo el byte 1 que lee la api, inclusive en su misma maquina puede variar de posición al reiniciar, pero siempre lo pueden hallar rápidamente con este método, reiniciemos el crackme.

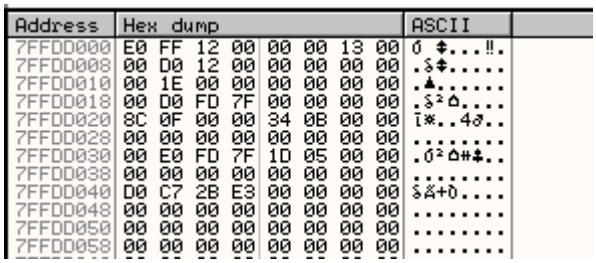


Esta vez no escribimos nada a ver si localizamos el byte a mano y si lo ponemos a cero a mano, y veremos si el programa corre, pues la api leerá dicho byte y si esta a cero, pues devolverá cero, jeje.

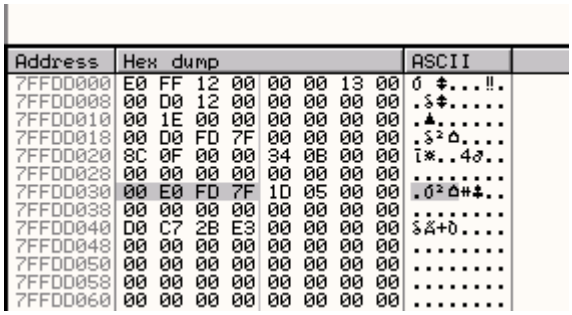
1)BUSCAR EL INICIO DEL TIB en la ventana de los registros del OLLYDBG



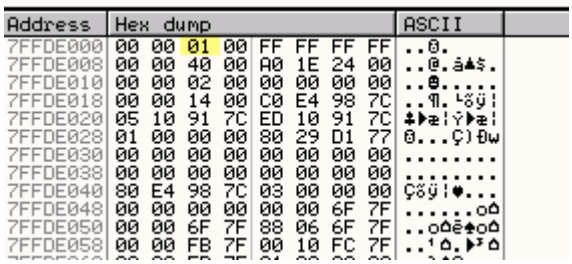
Ver el TIB en el dump (ya no esta en la misma posición que arranco la vez anterior)



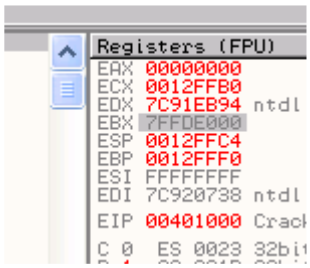
Busco fs: [30] o sea le sumo 30 al inicio del TIB y hallo su contenido



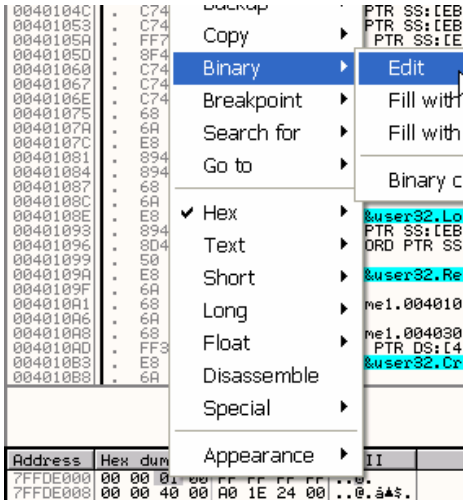
El contenido es 7ffde000 ese lo busco en el dump

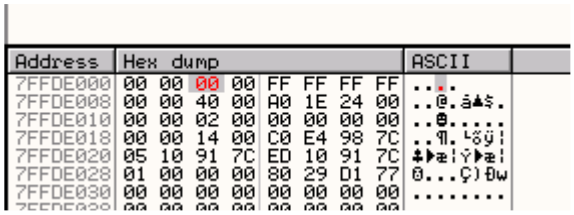


Y le sumo 2 y allí esta el byte (muchos dirán para que tamaña explicación si los programas cuando arrancan en EBX, esta siempre el puntero a esta zona, jeje o sea EBX= fs:[30], jeje)

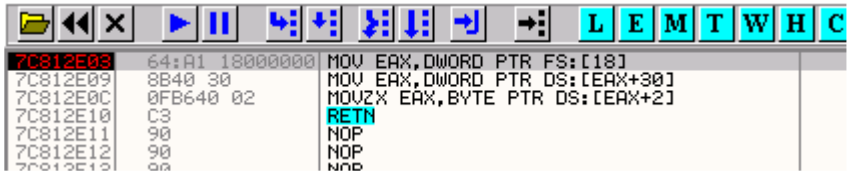


Bueno pero es bueno saber el método entero, porque por ahí uno no esta al inicio del programa y quiere hallar el byte a mano y no sabe que valor tenia EBX en el arranque, pero bueno este es el método completo, ahora pongamos a cero el byte.

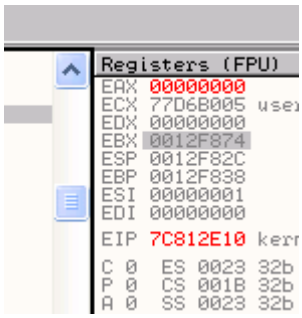




Bueno pongamos un BP IsDebuggerPresent a ver si funciona



Traceemos hasta el RET



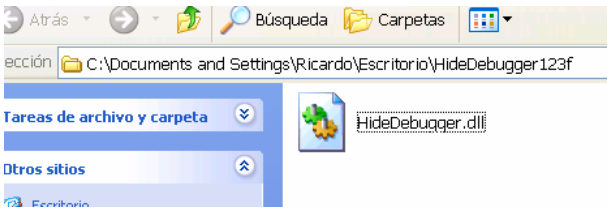
Y vemos que EAX que es el valor de retorno de la api vale cero o sea que al cambiar el bytea mano, el programa cree que no esta siendo debuggeado tanto por el uso de la api o por la lectura del byte sin la api.

Y por supuesto corre perfectamente.

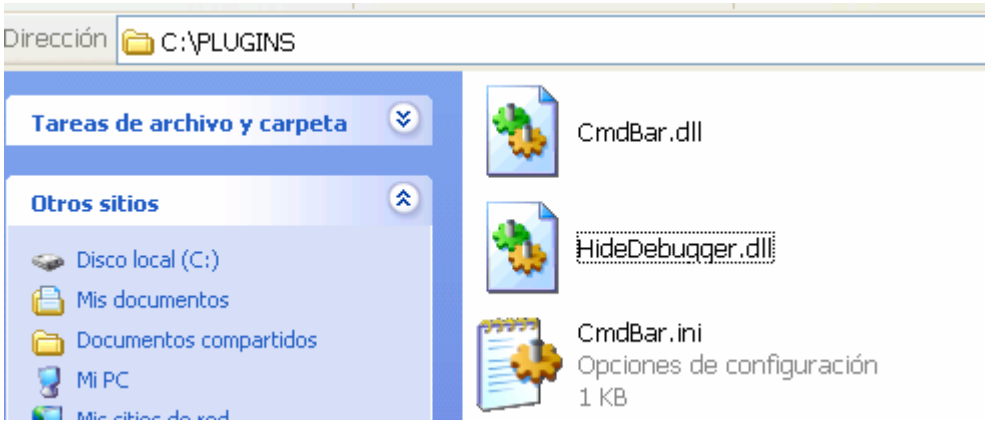
Obviamente hay muchos plugins que hacen este trabajo, uno de ellos es el plugin HideDebbuger 1.23f que pueden descargar desde mi http aquí

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

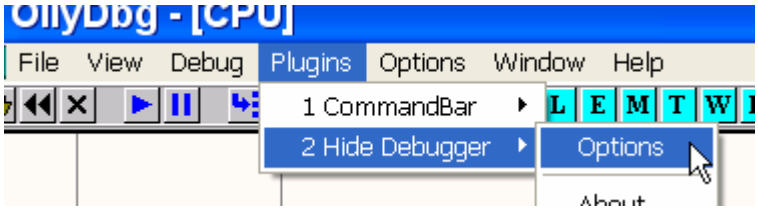


COPIANDO jeje

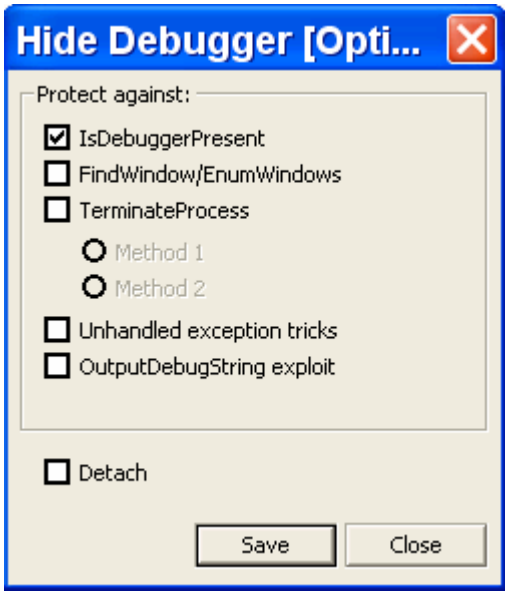


Allí esta debajo del otro

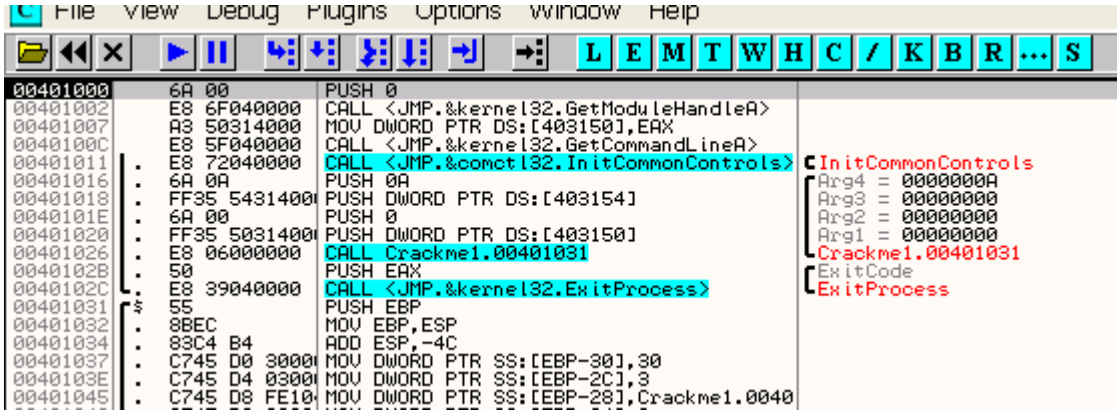
Ahora reinicio el OLLYDBG



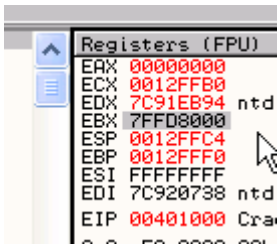
Veo que en las opciones del plugin



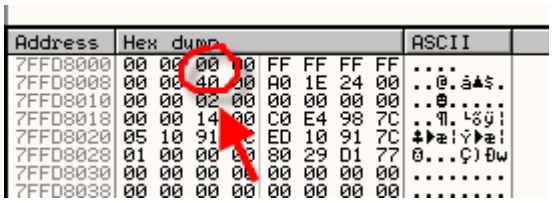
Viene ya marcado para evitar que detecte por el byte de IsDebuggerPresent o sea hace el trabajo que nosotros hicimos a mano lo pone a cero siempre, apretamos SAVE y las otras opciones por ahora las dejamos deshabilitadas, a medida que las vayamos estudiando, las veremos.



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



Hago EBX-FOLLOW IN DUMP



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