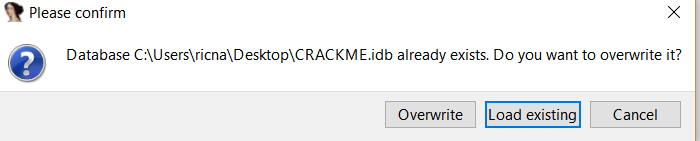
INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 10

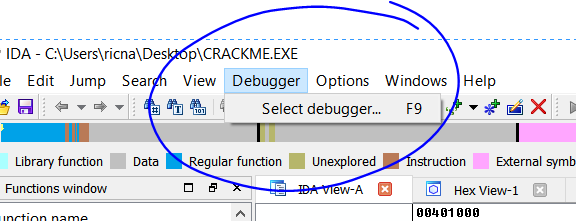
Bueno hemos visto algo del LOADER y seguiremos viendo, pero veremos algunos aspectos del DEBUGGER para poder ir complementando ambos.

IDA soporta múltiples DEBUGGERS para verlo abrimos el CRACKME DE CRUEHEAD original sin parchear en el IDA.

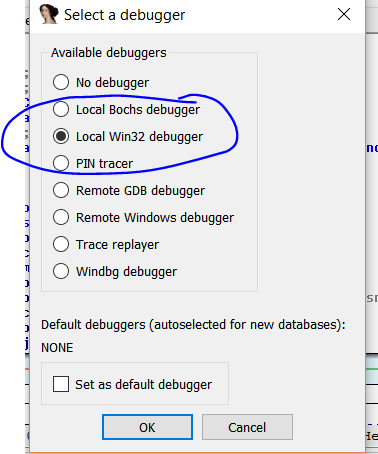
Obviamente elegimos OVERWRITE si nos pregunta si queremos crear una nueva database y sobrescribir la vieja, para que haga un análisis nuevo, si ya teníamos en el mismo lugar la anterior database o archivo idb del parcheado.



No ponemos la tilde en MANUAL LOAD, así que aceptamos las ventanas que siguen hasta que abra.



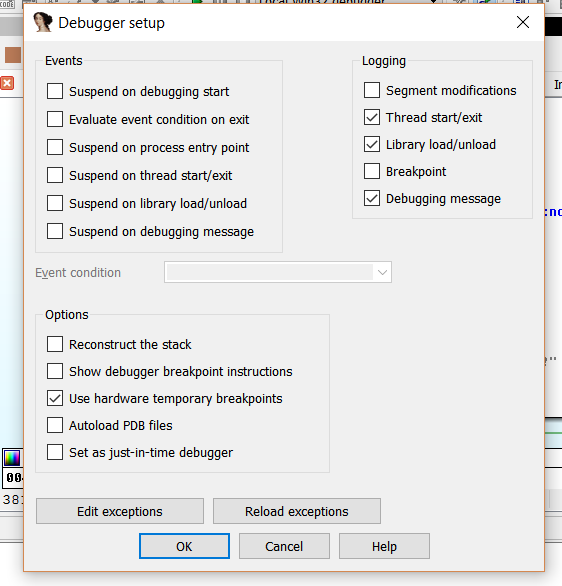
Obviamente para empezar por el principio entre las diferentes posibilidades de DEBUGGERS elegiremos LOCAL WIN32 DEBUGGER servirá para comenzar, más adelante veremos las otras.



Veremos algunos puntos del DEBUGGER incluido en IDA que tiene sus particularidades y que si uno no lo aprende a usar ya que es un poco diferente del resto de los debuggers se puede complicar.

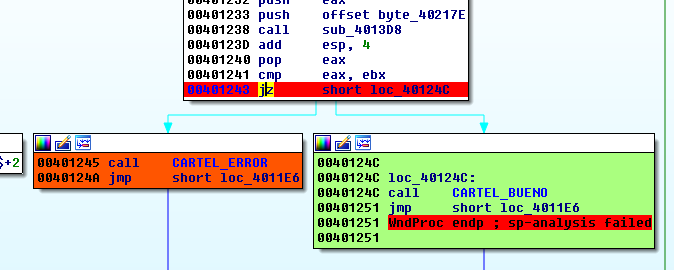
La diferencia surge creo yo en que IDA comenzó siendo solo lo que hoy llamamos el LOADER o sea un muy buen desensamblador estático y con grandes posibilidades de interactividad para el reversing, luego se le añadió el DEBUGGER sobre el mismo lo cual trajo algunos problemas que se fueron solventando pero que hacen que la forma de trabajar a veces sea un poco distinta a DEBUGGERS como OLLYDBG.

En DEBUGGERS – DEBUGGERS OPTIONS tenemos las opciones para el DEBUGGER.

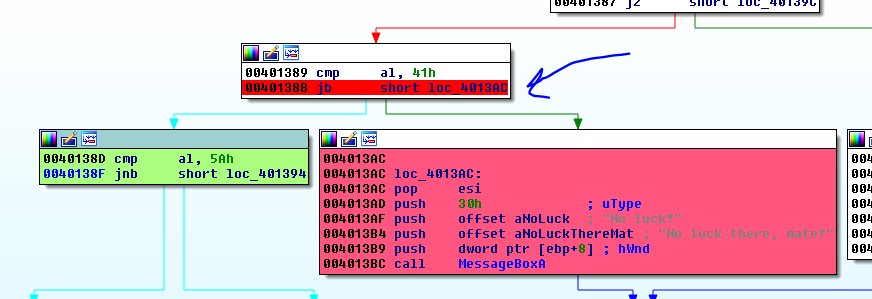


Pondremos la tilde en SUSPEND ON PROCESS ENTRY POINT para que pare en el punto de entrada del mismo.

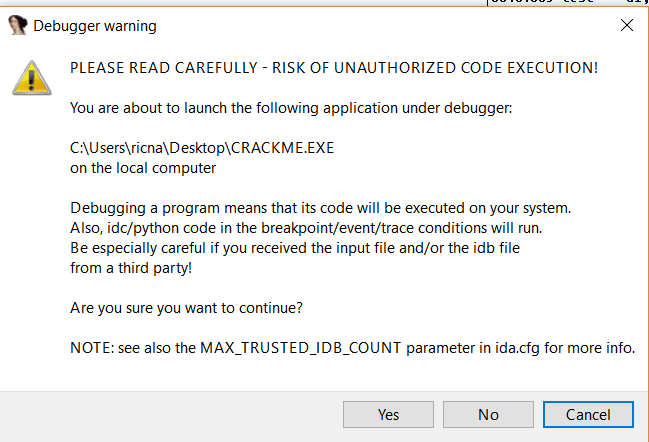
Haremos los cambios en el analisis que habíamos hecho antes, COLOREAREMOS y RENOMBRAREMOS la zona de los saltos decisivos.



Allí ponemos un BREAKPOINT con F2 en el salto que toma la decisión en 0x401243 y vamos al otro salto que habíamos parcheado y también F2.

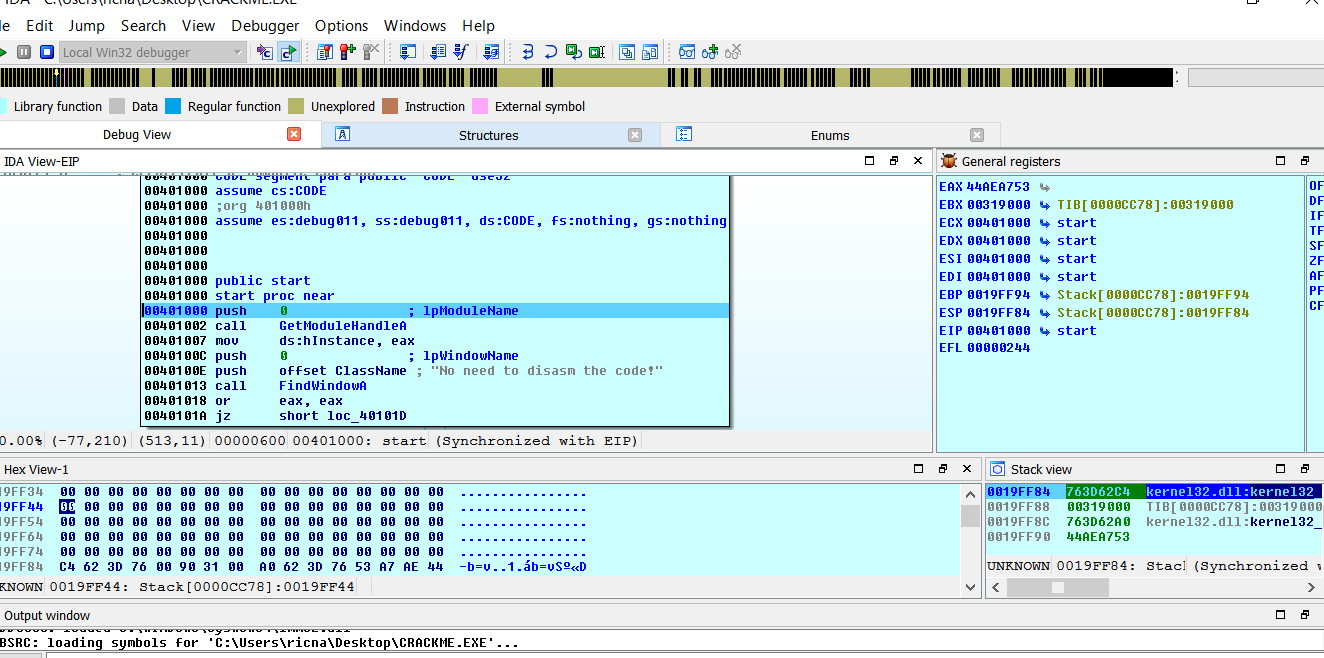


Ya estamos como lo habíamos dejado pero sin cambiar codigo, ahora podemos arrancar el DEBUGGER con DEBUGGER-START PROCESS.



Esto siempre nos aparecerá cuando vayamos a DEBUGGEAR un ejecutable en nuestra maquina local, ya que mientras analizamos en el LOADER jamás se ejecuta en nuestra máquina, pero ahora si se ejecutara, por lo tanto siempre se debe tener cuidado al ejecutar si es un VIRUS o algo potencialmente peligroso conviene usar el DEBUGGER REMOTO y ejecutarlo en una máquina virtual, ya veremos eso a su tiempo.

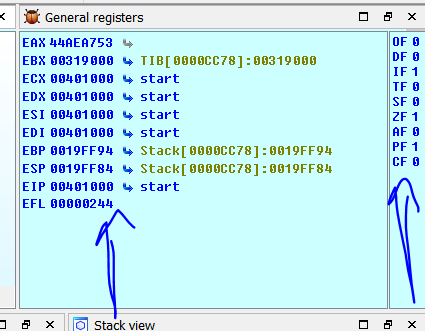
Como el CRACKME DE CRUHEAD es más bueno que LASSIE medicada, apretamos YES.



Como le pusimos que se detenga en el ENTRY POINT así lo hizo, está parado en 0x401000 si apretó la barra espaciadora pasara a modo grafico como en el LOADER.

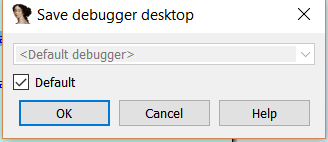
El color celeste de fondo de los bloques me dice que estoy en el DEBUGGER, en el LOADER los bloques son blancos por default.

Acomodo las ventanas sobre todo achicando un poco la parte del OUTPUT WINDOW, arrastrando su margen hacia abajo, agrando un poco el stack y que se vean los registros arriba a la izquierda y los flags

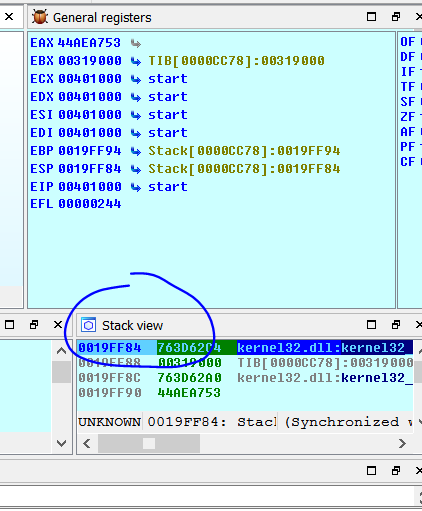


Allí está la vista de REGISTROS y FLAGS,

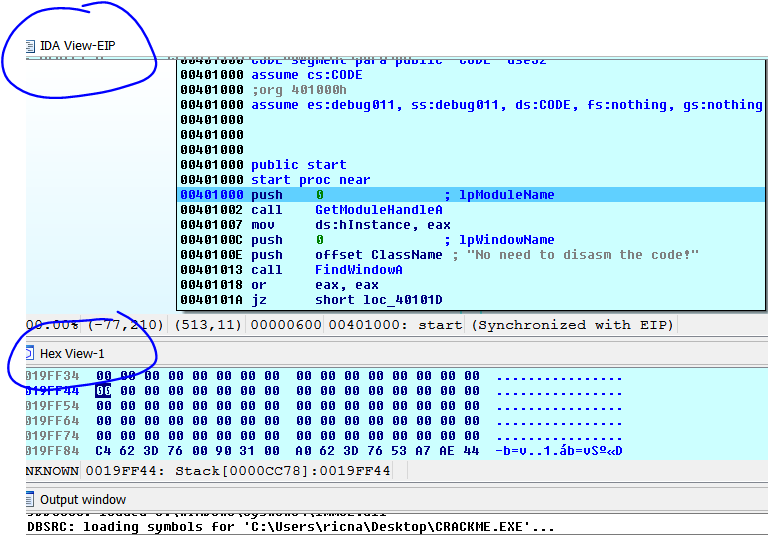
Una vez que logre una vista cómoda, la guardare por default para el DEBUGGER en WINDOWS-SAVE DESKTOP poniendo la tilde en DEFAULT siempre que arranquemos el DEBUGGER arrancara de la forma que elegimos y si queremos volver a cambiarlo, podemos volver a hacerlo sin problemas.



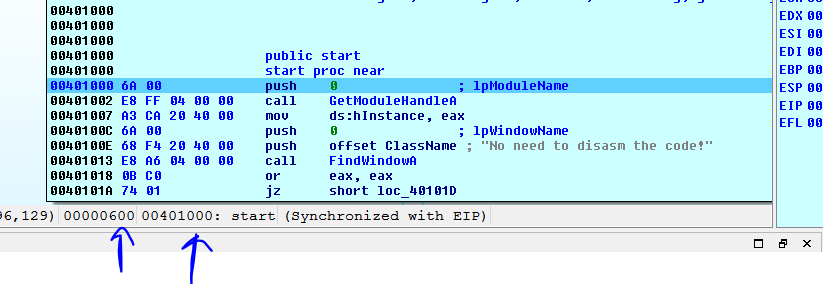
Debajo de los registros esta la vista de STACK.



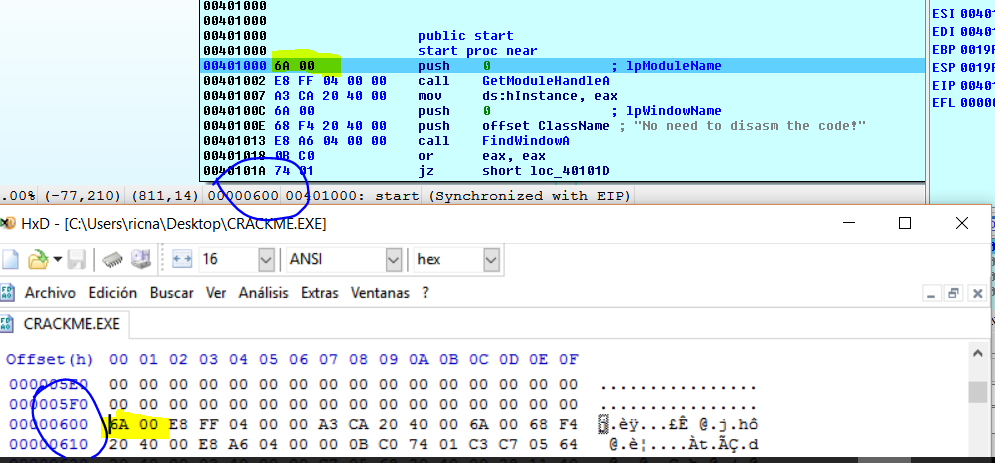
Y tenemos también el listado IDA-VIEW EIP y abajo el HEW VIEW o HEX DUMP la vista en hexadecimal de la memoria.



Si vemos en la parte inferior del listado.

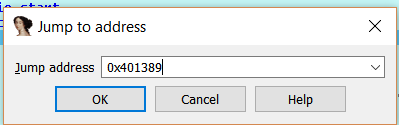


Siempre tendremos la dirección de memoria y el FILE OFFSET que es el OFFSET en el archivo ejecutable, si lo abrimos en un editor HEXA por ejemplo el HXD.

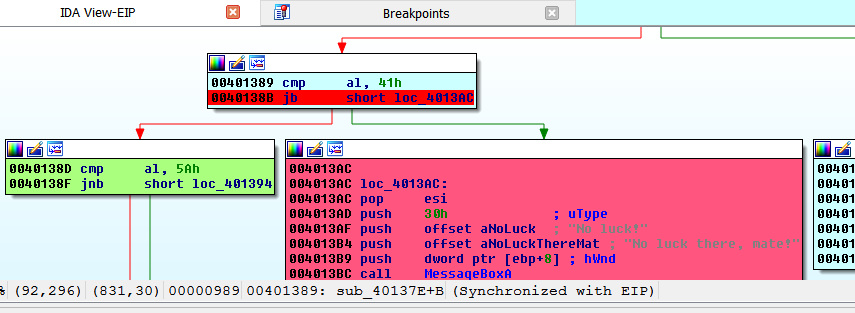


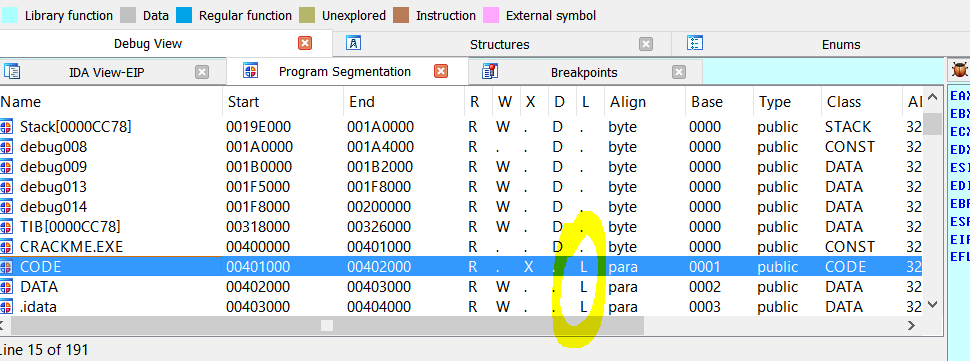
Vemos que en el FILE OFFSET 0x600 están los mismos bytes.

Ya sabemos que por DEFAULT en el LOADER y el DEBUGGER viene configurado el atajo de teclado G para ir a una dirección de memoria, si apretó G, y pongo 0x401389.



Voy a la zona donde estaba un breakpoint, vemos los colores, le quito para que no se vean los bytes, para no ensuciar la vista, y veo que todo está tal cual lo dejamos en el LOADER si reverseamos, cambiamos nombres etc, aquí en el DEBUGGER se mantienen, asimismo los cambios que hagamos aquí aparecerán en el LOADER, ya que es un módulo que está marcado como los que cargan en el LOADER.

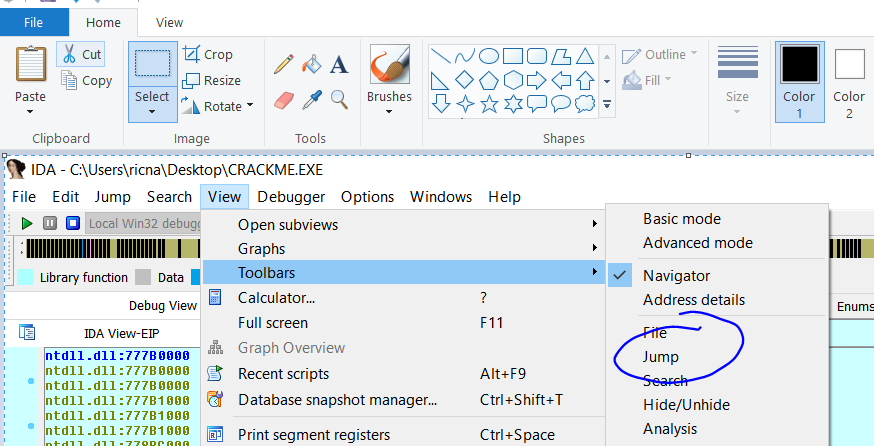




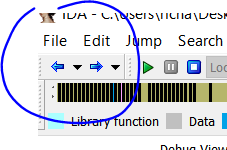
En VIEW-OPEN SUBVIEW-SEGMENTS vemos ahora los tres segmentos que carga el LOADER cuando no activamos MANUAL LOAD, el de código o CODE que carga en 0x401000 el de DATA e IDATA, cualquier reversing que hagamos a estos tres, se mantendrán pues cargarán en el LOADER, fuera de esos tres, los cambios se perderán pues son solo módulos cargados por el DEBUGGER y no se guardaran en la database.

O sea que siempre los módulos que estemos reverseando y queremos debuggear deben estar en el LOADER o L allí, obviamente cargaran en el DEBUGGER también, pero que tengan la L significa que estarán en ambos y podre reversear estáticamente en el LOADER y DEBUGGEAR en ellos sin perder info ni mi trabajo de reversing estático.

Una de las barritas que me es siempre muy útil es

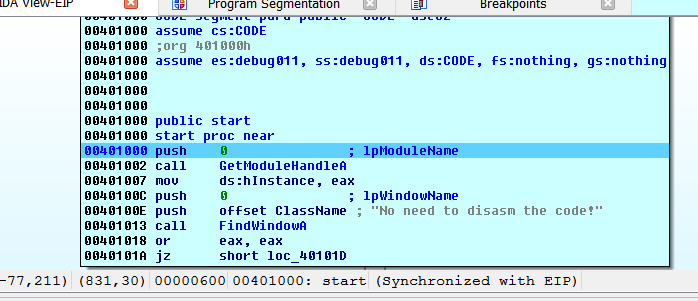


La agrego y vuelvo a guardar el con SAVE DESKTOP para que quede por default.

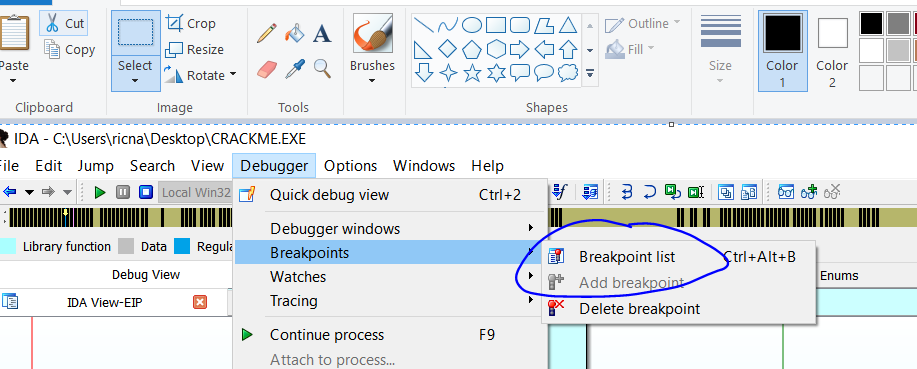


La posibilidad de volver para atrás al lugar donde estabas antes y hacia adelante es muy cómoda tenerla a mano.

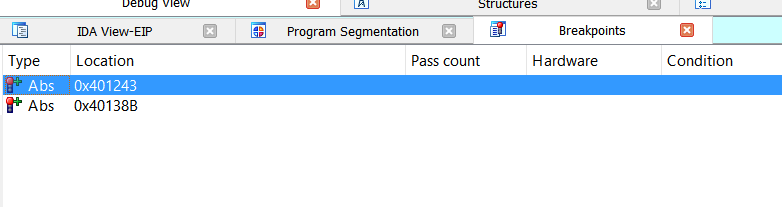
Apretando la flecha para atrás vuelvo el entry point donde estaba antes.

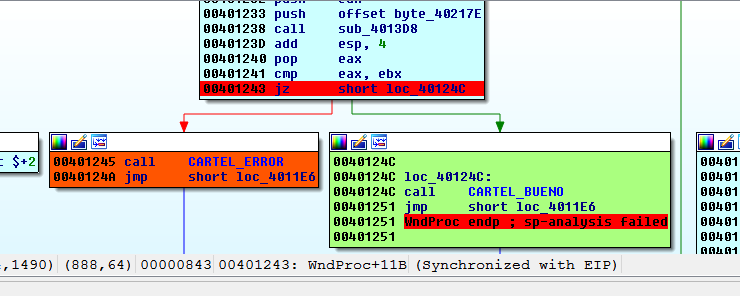


También en el menú DEBUGGER.

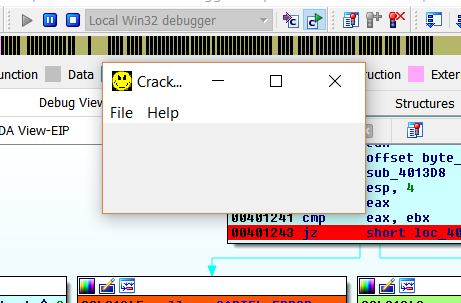


Puedo ver la lista de BREAKPOINTS e ir al que quiera doble clickeando en el.

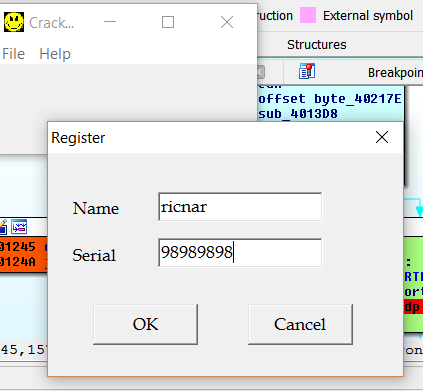




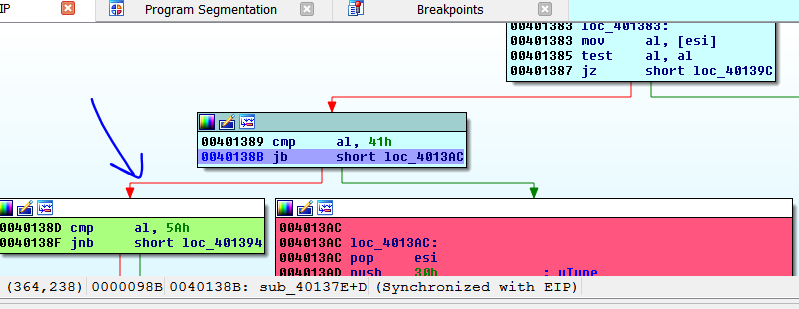
Así que estoy parado en el Entry Point y tengo los dos BREAKPOINTS puestos, así que podría apretar f9 para que corra.



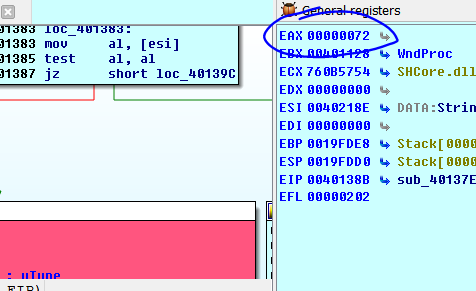
Vamos a HELP-REGISTER y pongamos una clave.



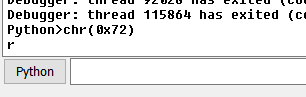
Al dar OK.



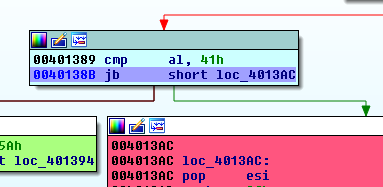
Queda titilando la flecha de la izquierda que es por donde va a seguir, vemos que EAX vale 0x72



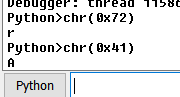
Si tipeo en la barra de Python chr(0x72) veo que es la r de ricnar.



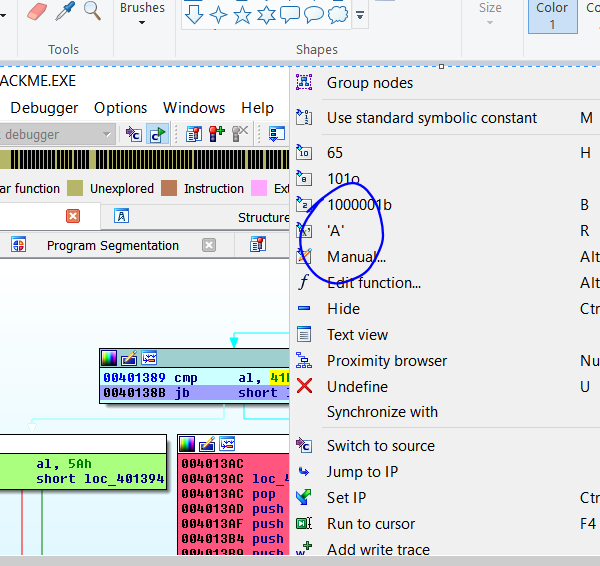
Y va a comparar si es más bajo que 0x41.

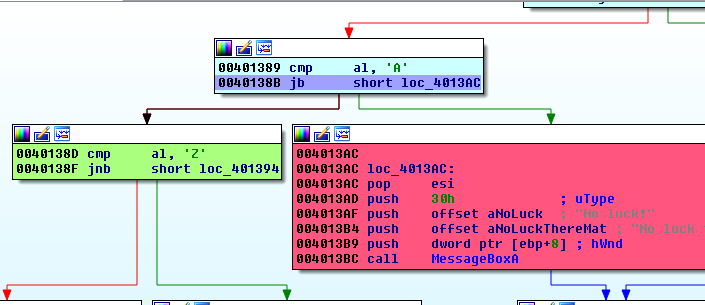


Podemos también ver con chr(0x41) en la barra de Python que es la A.

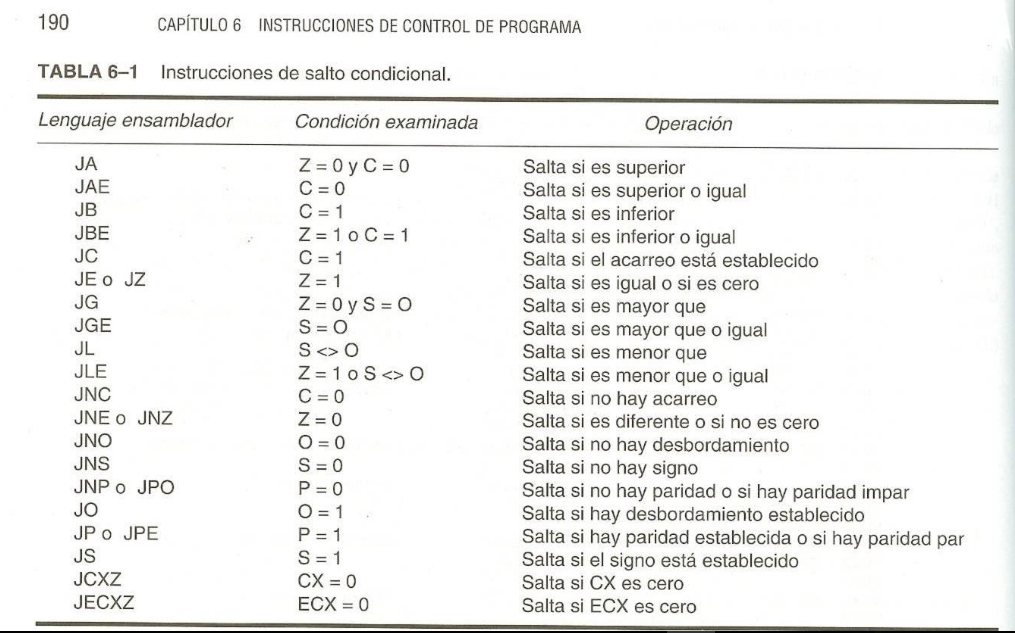


Pero también podemos para que quede más claro hacer click derecho en el 41h del listado y entre las opciones que podemos elegir de visualización me aparece la A.

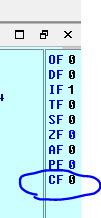




Vemos que compara contra A y contra Z por ahora no buscamos solucionar completamente el crackme, pero vemos que 0x72 es más grande que 0x41 por lo tanto no va al bloque rojo del cartel de error. Obviamente saltaría al bloque rojo si es más bajo, pero también se puede evaluar mirando los flags.



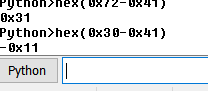
Allí vemos el salto JB el cual salta o sea sigue la flecha verde en IDA si es el primero es menor, pero también al realizar la comparación se activa el flag C (también llamado CF o C) , allí dice salta si C=1, si vemos en el IDA los flags.



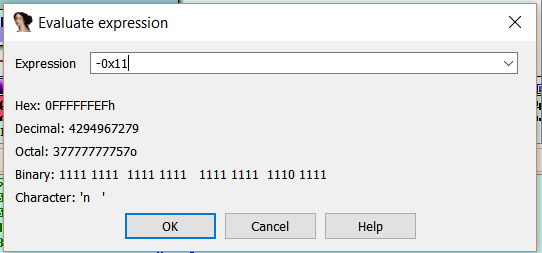
El flag C está a CERO por lo cual el salto no se tomará e ira por el camino de la flecha roja.

Ahora cual es la condición matemática en la cual se activa el CARRY FLAG?.

EL CARRY FLAG nos da información de que algo salió mal en una operación entre unsigned integers, si hago la resta ya que el CMP es una resta sin guardar el resultado, 0x72-0x41 el resultado será 0x31 que es positivo y no hubo problemas, sin embargo si mi valor fura 0x30 por ejemplo al restarle 0x41, me dará -0x11.



Lo cual es un valor negativo y no es aceptado como resultado de una operación de números positivos, pues si lo continua trabajando como hexadecimal.



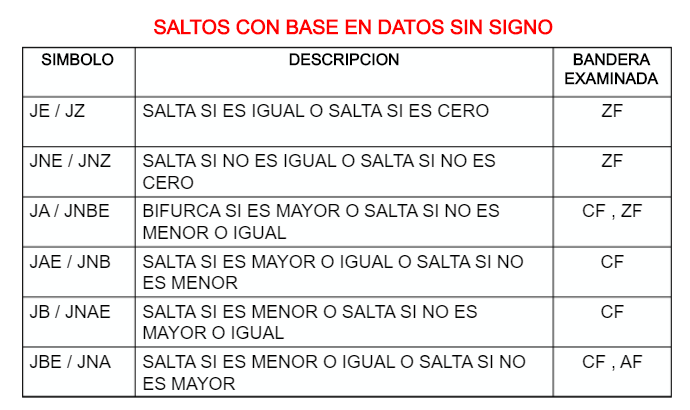
Sera 0xFFFFFFFEF y eso como positivo es un valor muy grande 4294967279 y de ninguna manera restar 0x30 – 0x41 es igual a 0xFFFFFFEF positivo.

Como sabemos si en una operación donde no se considera el signo o no?.

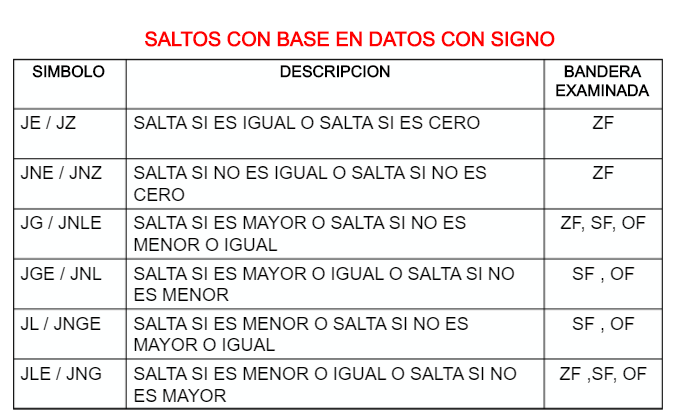
Eso está dado por el tipo de salto en este caso JB es un salto que se usa luego de una comparación SIN SIGNO para las operaciones entre números CON SIGNO usara JL.

Si comparo 0xFFFFFFFF contra 0x40 en un salto sin signo obviamente es mayor pues es el máximo positivo, pero si es un salto donde se consideran los signos será -1 y será menor a 0x40.

O sea que para evaluar si una comparación usa signo o no debemos ver el salto siguiente que toma la decisión.



Si el salto es cualquiera de estos se evalúa SIN SIGNO, mientras que cada uno tiene su contraparte como JB y JL en la tabla de los saltos CON SIGNO.

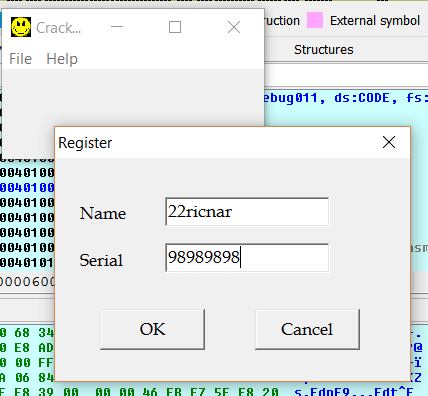


Vemos que JE que evalúa si son iguales está en ambas tablas pues en ese caso no importa el signo si son iguales se activará poniéndose a 1 el ZF.

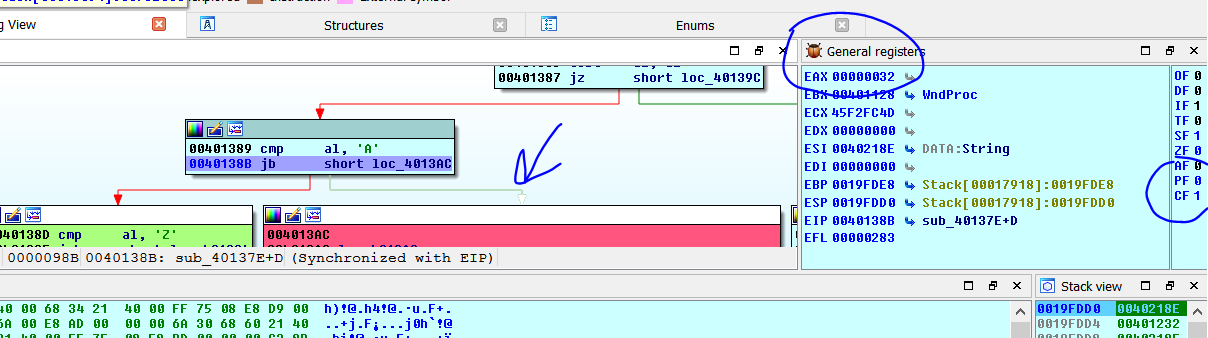
Vemos también que JG que salta si es mayor en la tabla de los con signo tiene su contraparte con el JA que salta si es mayor en la tabla de los SIN SIGNO.

Igual en el análisis diario uno no se pone a mirar las flags demasiado, ve un salto JB y sabe que es una comparación entre números positivos o SIN SIGNO y que si el primero es menor saltara, pero es bueno ver lo que hay debajo.

Si continúo parando en todos los breakpoints veré que estoy en un loop que va leyendo uno a uno mis caracteres del nombre y compara todos con 0x41 si hay alguno menor saldrá el cartel de error, como puse todas letras (ricnar) no será el caso, pero restarteemos el proceso con TERMÍNATE PROCESS y START PROCESS nuevamente y ahora pongamos como nombre 22ricnar y clave 98989898.



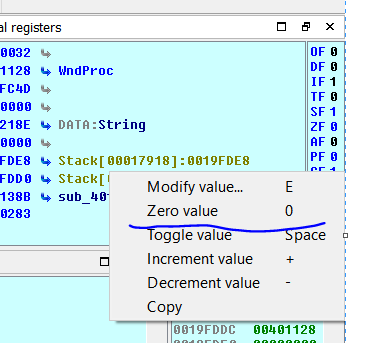
Al aceptar parara en el breakpoint.



Vemos que ahora mi primer carácter es el 0x32 que corresponde al 2 de 22ricnar.

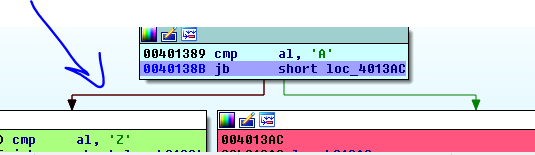
Veo que como 0x22 es menor que 0x41 se activa la flecha verde o sea que el salto se tomara y vemos que el flag C está activado porque al restar 0x32 menos 0x41 en una resta sin considerar signo nos da negativo el resultado y eso es un error que activa el flag C.

Si hago click derecho en el FLAG C



Podemos ponerlo a cero

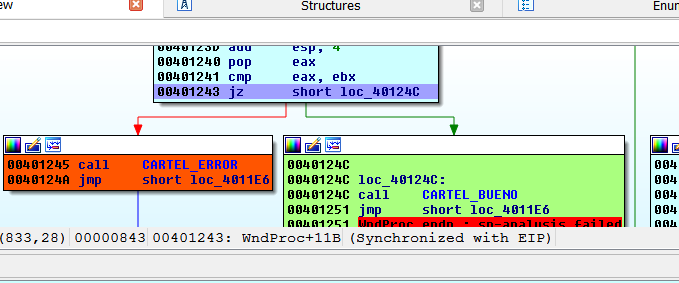
En ese mismo momento que lo cambiamos empieza a titilar la flecha roja porque lo invertimos.



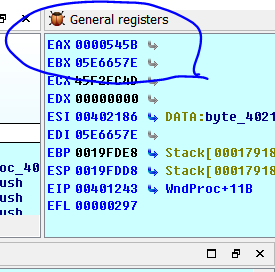
Si damos RUN volverá a parar cuando evalué el siguiente 2 de 22ricnar y volverá a titilar la flecha verde que llevaría al cartel de error, invertimos de nuevo CF poniéndolo a cero.

Las siguientes veces que para en este salto corresponden a ricnar por lo cual como son mayores que 0x41 no activan el CF y siguen por la flecha roja.

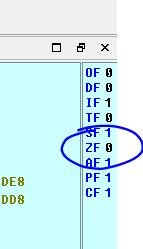
Después de pasar el chequeo de cada carácter del nombre llegamos al salto final.



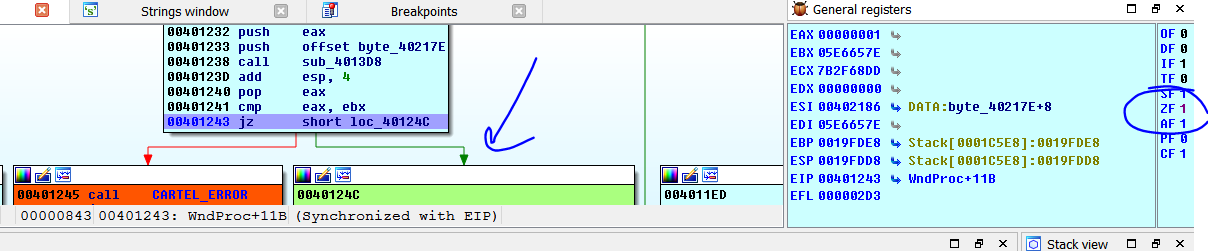
Allí compara que EAX y EBX sean iguales aquí no importa signo ni nada, ya me prendió la flecha roja de que no son iguales y me va a tirar al cartel de error.

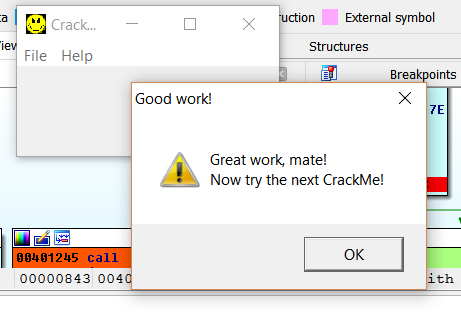


Ahí veo que no son iguales, y el flag Z no está activo.



Si lo activamos cambiara el salto e ira por la flecha verde al cartel de GOOD BOY, click derecho en el ZF y elijo INCREMENT VALUE.





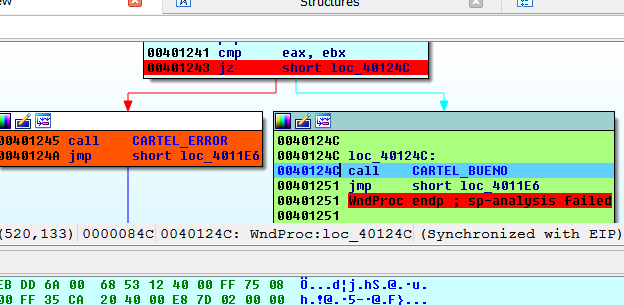
Hicimos lo mismo que cuando parcheamos, pero sin hacer modificaciones solo cambiando los flags en el DEBUGGER.

Muchas veces cuando uno no tiene ganas de invertir saltos directamente va al bloque bueno donde quiere continuar y pone el cursor allí y hace click derecho, aunque IDA 6.8 tiene un BUG que fue resuelto en la versión 6.9 que a veces hacer click derecho crashea el IDA si les ocurre buscan el shortcut que necesitan, algunos están en:

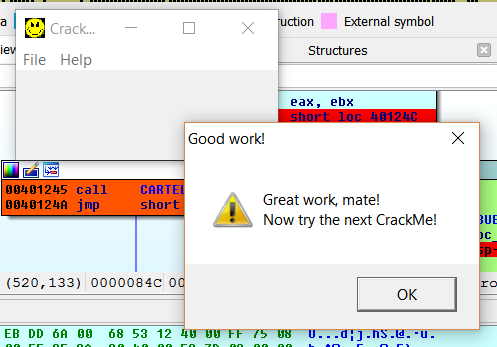
<https://www.hex-rays.com/products/ida/support/freefiles/IDA_Pro_Shortcuts.pdf>

Sino en el mismo IDA en OPTIONS-SHORTCUTS.

Si nos trae problemas el click derecho para setear EIP poner el cursor donde queremos ir por ejemplo 0x40124c y apretamos CTRL + N que es SET IP.



Y el programa continuara desde 0x40124c que es lo mismo que haber invertido el flag.



Hasta la parte 11.

Ricardo Narvaja