# Protector debuger/debugee estilo Armadillo

# Introducción

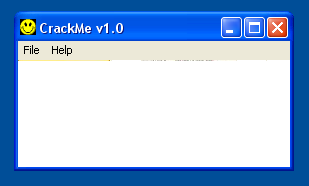
Este solucionario fue construido para el ciclo de capacitación semanal (desde ahora CCIS), en el mismo se expondrá el funcionamiento del protector de software como también la solución del crackme obteniendo un serial valido.

El software protector consta de una protección que hace correr la aplicación en dos procesos, conocida comúnmente como protección padre e hijo y también cuenta con una redirección de IAT la cual fue eliminada. Dicho protector fue construido por deroko, un ingeniero inverso muy respetado en la scene.

Deroko solo nos otorgo un crackme protegido con este packer. Luego de un extenso trabajo sobre la muestra logre alterar el funcionamiento del protector para poder usarlo en cualquier software.

# Primer acercamiento al crackme

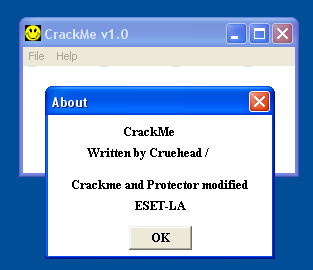
Lo primero que hacemos es cargar el crackme para ver si vemos algo conocido, no todo arranca con un debugger.



Este es el crackme de cruehead versión 1.0, cualquiera que allá experimentado más de dos horas en el mundo del cracking lo conoce.

Es un crackme para nivel principiantes, así que va a ser muy fácil esto ☺

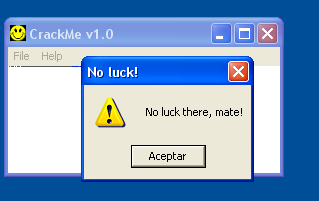
Miremos los menús:



Por lo visto el reto fue modificado. Pero hagamos caso omiso a todo esto y bajemos uno de los tantos keygen que crearon para este programa. En mi caso voy a probar con este que lo encontré en la página de Ricardo Narvaja:



Ya tenemos nuestro password, ahora a probarlo a ver si anda.

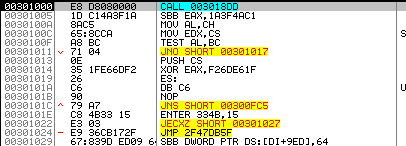


FAIL!!!

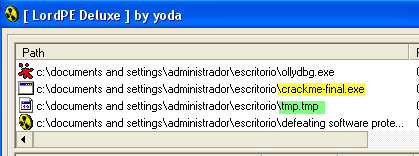
Como era de esperar el talentoso reverser que construyo el reto ha modificado el algoritmo.

# Primer análisis

Vamos a abrirlo con el debugger para ver que secretos se esconden:



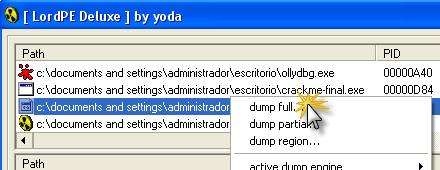
La experiencia nos dice de que este programa está protegido, varios indicios son una dirección base de ejecutable modificada (por lo general es 0x00400000 y en este es 0x00300000), después hay un montón de instrucciones que no tienen mucho sentido, aunque esto lo notarían cuando ya tienen mucha experiencia. Miremos la lista de procesos activos con LordPE para ver si encontramos algo:



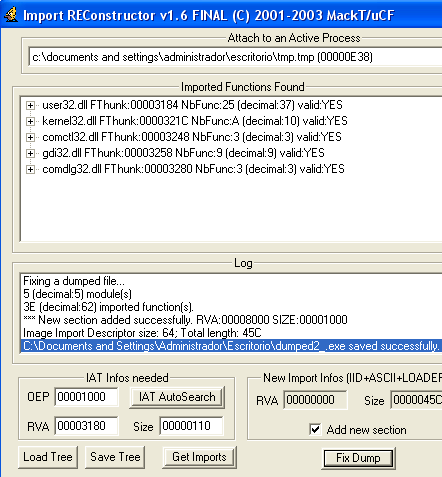
El cracking también tiene su etapa de information gathering, y es la que nos da los posibles vectores de ataque a la protección, como por ejemplo acá vemos el proceso del crackme (crackme-final.exe) y vemos un tmp.tmp muy sospechoso.

Este es un programa corriendo en la memoria de nuestra PC, así que nada nos impide volcar esa memoria para tratar de obtener un binario valido.

Entonces procedemos a dumpear la memoria al disco para ver si podemos reconstruir el ejecutable:

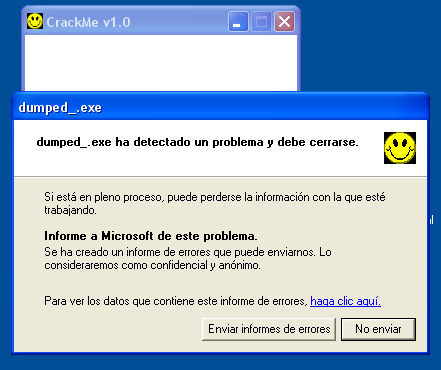


Le ponemos un nombre y lo guardamos convenientemente. Ahora viene la parte de reparar la tabla de importaciones del archivo, la verdad es que no es necesario explicar el porqué de este paso porque no tiene nada que ver con el reto. Solo tómenlo como un punto necesario para obtener un binario valido y que corra correctamente.



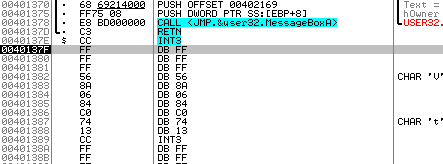
La IAT es una especie de CALL TABLE que usa el loader de Windows para hacer el programa portable a otras plataformas.

Ya con el programa preparado para correr lo ejecutamos, parece que anda todo correcto pero a la hora de tratar de registrarlo nos produce un error.



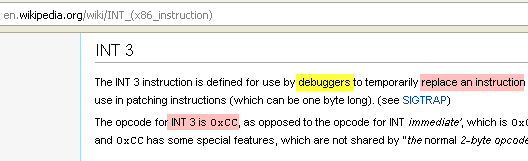
Windows no sabe cómo manejar la excepción que paso y mata el programa inescrupulosamente.

Abramos el binario con el olly para ver qué es lo que le duele:



Cuando llegamos a la etapa de registro y probamos cargar nuestros datos, el olly rompe misteriosamente en la instrucción INT3.

Miremos San Wikipedia para ver que es esa instrucción:

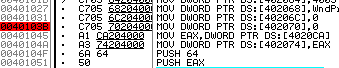


Bla bla bla, nos dice algo así como que es utilizada por debuggers para remplazar una instrucción y su opcode es INT3, la misma genera una excepción que el debugger tendrá que manejar.

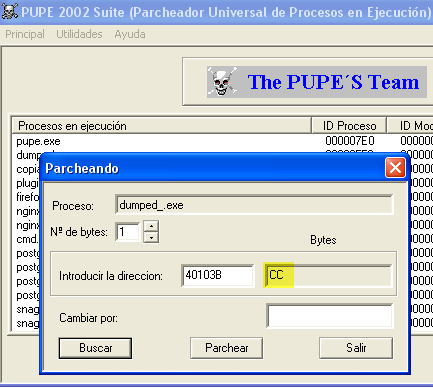
Hasta acá todo bien. El ollydbg paro, lo que significa que está tratando de manejar la excepción como dice el texto. Si pensamos bien, esto vendría a ser como un breakpoint.

Chequeemos el funcionamiento de OllyDBG para ver si estamos en lo correcto.

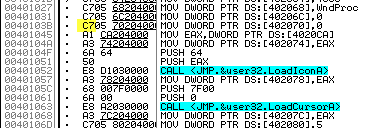
Abrimos cualquier ejecutable y le ponemos un bp con F2 en cualquier zona que nos agrade.



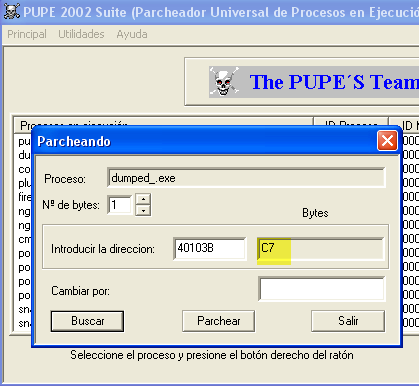
En este caso le clave un BP en 0x0040103B, ahora usemos PUPE (Parcheador Universal de Procesos en Ejecución) para ver que byte esta en esa dirección:



Ahí lo tenemos, el debugger cambio el opcode original por un 0xCC para cuando lo encuentre se genere una excepción y pare la ejecución. Probemos desactivar el breakpoint para ver si el olly restaura el byte original:



Miramos con PUPE:



Ahí vemos como borro el breakpoint y le puso el original nuevamente.

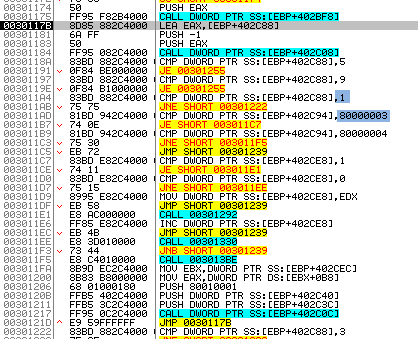
Ahora usemos nuestro cerebro un segundo para tratar de entender que es lo que está pasando en este programa:

* El ejecutable que dumpeamos tiene BP que si o si alguien los tiene que manejar para que no crashee.
* El protector genera 2 procesos.

La pregunta sería: ¿podrá ser que el primer proceso sea un debugger del segundo? Y que cuando encuentra un INT3 en el debugee, el debugger actué en consecuencia para que no crashee el programa.

Analicemos el binario protegido (no el dumpeado) para ver si estamos en lo cierto:

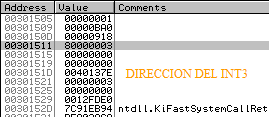
El protegido lanza el segundo proceso, por lo tanto obligatoriamente tiene que pasar por la API CreateProcessA, ahí es donde vamos a poner un BP y a partir de ahí vamos a tracear con F7 para verle las entrañas al programa.



Acá estamos en la zona caliente del programa. Cuando se topan con una protección así, ver un 0x80000003 ya los tiene que poner felices, esto se debe a que es el código correspondiente a SOFTWARE\_BREAKPOINT y con tantos CMP’s estamos más que seguros que aquí es donde se cocina el bacalao.

En la imagen les marque el 1 y el 0x80000003 porque es la zona caliente, el 1 es DEBUG\_EVENT.

Ejecutemos todo hasta que las 2 condiciones se cumplan, es decir que sea un DEBUG\_EVENT y que se trate de un Breakpoint:



Acá estamos, si miramos la imagen, tenemos el BP en 0x0040137E que es donde caímos cuando lo estábamos debugeando. Si no me creen suban para arriba y verifiquen la imagen antes de la explicación del INT3.

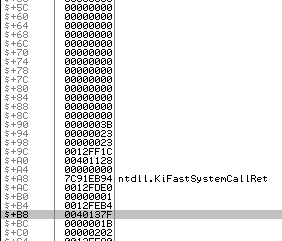
Ahora veamos como lo trata el debugger para que el debugee no muera:

08-03-2012 15-45-50.png

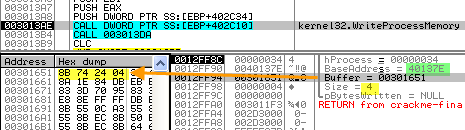
Hace una llamada a GetThreadContext, esta es una función de Windows que obtiene el estado de los registros en un programa freezado. Contiene todo, desde registros de depuración (algo avanzado y que no nos sirve ahora), como los registros de propósito general como son EAX, EDX, ECX, EBX. Los de puntero ESI y EDI y el registro EIP que es un puntero a la próxima instrucción a ejecutar.

Quedense con eso en la cabeza, **EIP es la próxima instrucción a ejecutar.**

Esta función devuelve una estructura así:

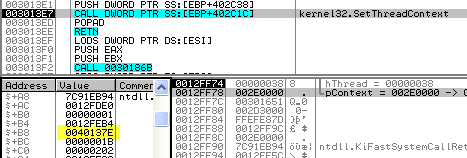


La zona que marcamos como $+B8 corresponde al valor del registro EIP, que como ven tiene la dirección de donde se produjo la excepción. Ahora si nosotros cambiáramos este valor, lograríamos que el programa siguiera su ejecución en cualquier zona que nosotros queramos. Eso dejémoslo para después, sigamos:



Ahora tenemos una llamada a WriteProcessMemory, no hace falta ser un genio para entender que hace la función, la misma escribe la cantidad de bytes que nosotros queramos en la zona de memoria del proceso debuggeado.

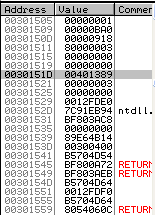
Miren atentamente los argumentos, tenemos que va a escribir 4 bytes en la dirección donde se produjo la excepción, y estos bytes son 0x8b742404. Sigamos:



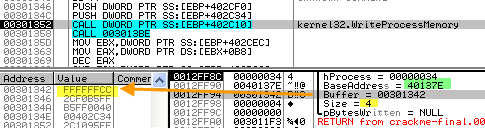
Ahora llama a SetThreadContext, que como pueden imaginar es el némesis de GetThreadContext, con uno obtenes los valores y con el otro lo seteas. Si miramos $+B8 vemos que lo decremento, esto lo hace para volver a ejecutar la instrucción que reescribió, luego de borrar el 0xCC que había en esa posición. Seguimos:

08-03-2012 15-50-03.png

Con ContinueDebugEvent hace como que nada paso y deja corriendo el programa debuggeado hasta que casque de nuevo y le pase nuevamente la ejecución al debugger. Esto pasa rápido, miremos la siguiente imagen:



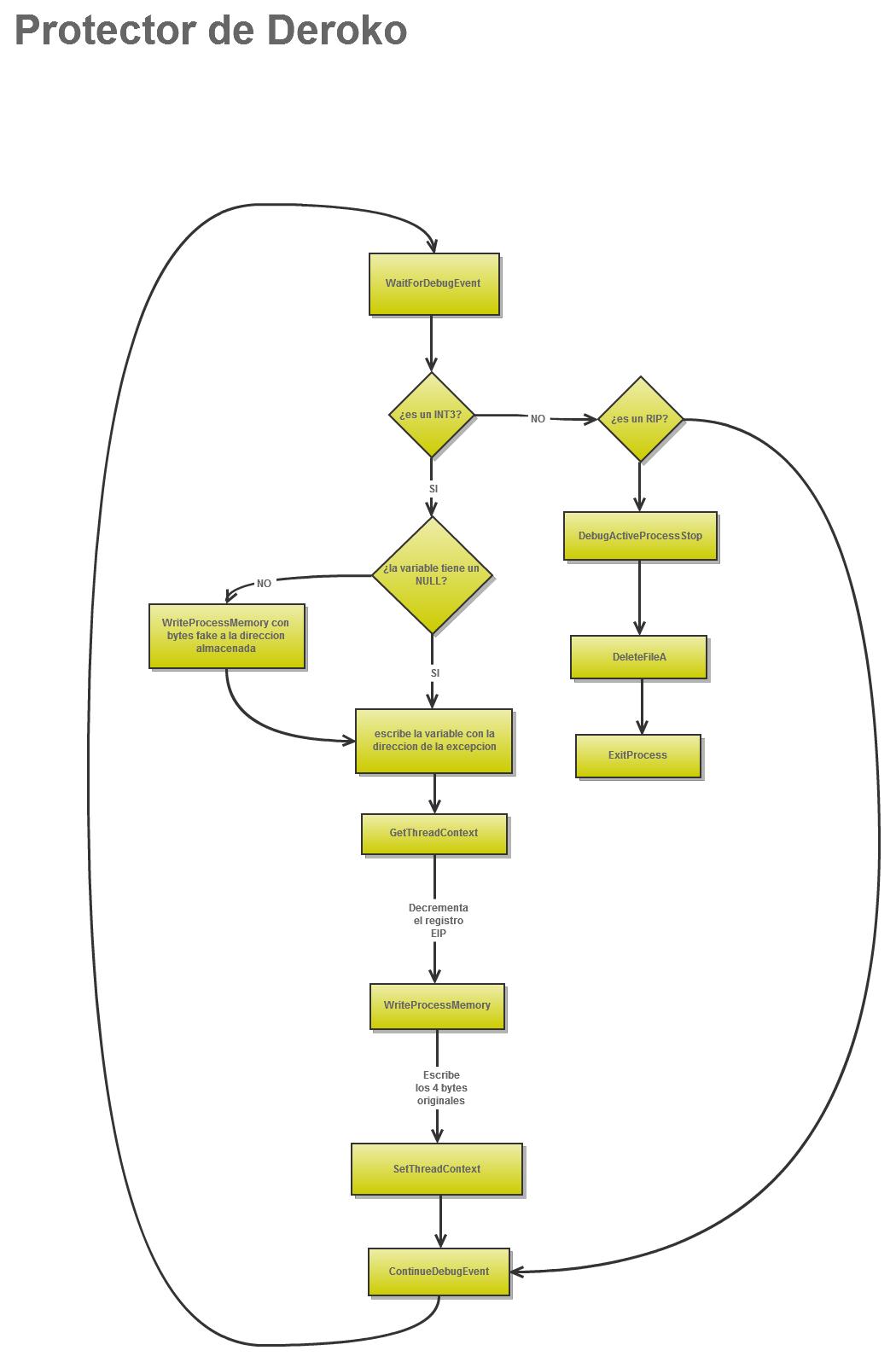
Luego de seguir con la ejecución salta un nuevo Breakpoint en la dirección 0x00401389, al tracearlo con F7 caemos en una zona que no habíamos pasado antes:



Esta porción de código está escribiendo los bytes 0xCCFFFFFF en la dirección de la excepción anterior.

Es decir que corrige la actual pero rompe la última que reparo.

Yo me tome el trabajo de hacer un diagrama de flujo completo del protector para que el concepto quede bien explicado.



# Rompiendo el protector

Ahora llego la parte más divertida para un cracker, que es la etapa posterior al análisis de la protección y en la cual el cracker se extasia de felicidad y comienza a hacer bailes como strippers alrededor de sus compañeros.

Hay muchas formas de romper esta protección:

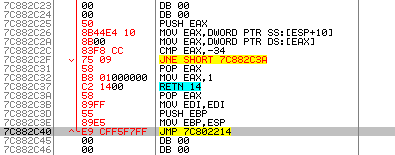
* nosotros podríamos reversear mas el algoritmo para obtener los bytes originales que deben estar en la memoria del padre.
* Podríamos modificar el programa para que no cargue con nada la variable que utiliza para preguntar si tiene que romper algo.
* Se podría Hookear la función WriteProcessMemory para que cuando llega como argumento un 0xCCFFFFFF no escriba nada.

De todas las opciones la tercera es la mas transparente al programa de protección y nos permite extrapolar el ataque hacia protecciones parecidas.

Hookear una función de una API es ponerle un JMP en el principio, y mandarlo a una zona de código que controlamos nosotros para posteriormente retomar la ejecución original del programa como si nada hubiera pasado.

07-03-2012 21-15-15.png

Como dijimos cambiamos los primeros bytes por un salto a nuestro código:

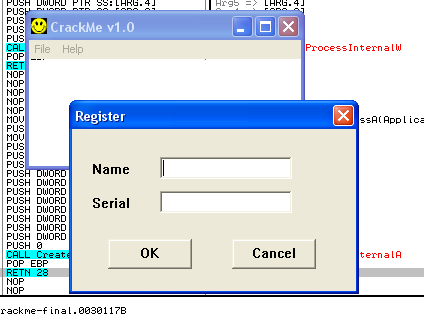


Ahí tenemos nuestro injerto, este código verifica que el buffer que se le pasa a WriteProcessMemory no sea -34, si es -34 entonces retorna la función poniendo en EAX un 1 como si todo hubiera salido bien.

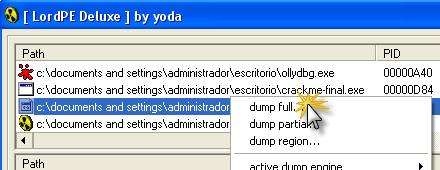
¿Pero que es el -34? Es la representación negativa del numero:

07-03-2012 21-18-25.png

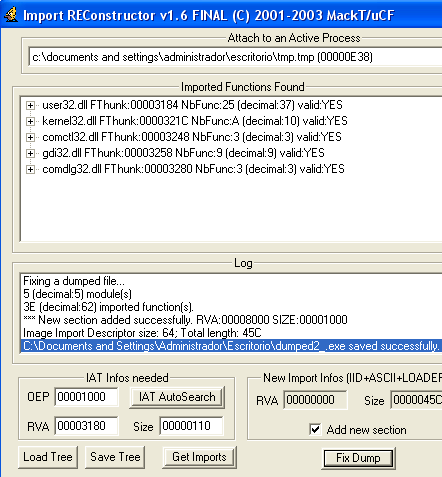
Le estamos diciendo que cuando lo quiera romper no haga nada ☺, entonces el programa se va corrigiendo por si solo a medida que lo utilizamos.



Lo ejecutamos y probamos el programa hasta cansarnos, sabiendo que nuestro injerto esta corrigiendo el programa debuggeado. Una vez que nos cansemos procedemos a dumpearlo y corregir la IAT como ya hicimos:



IAT



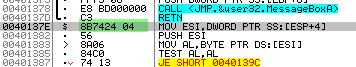
Y lo corremos:



Bueno, no rompió, lo que significa que rompimos la protección de los INT3 y ya podemos trabajar con el binario desprotegido.

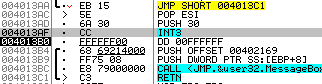
# Solucion al Crackme

Otra parte interesante seria encontrar un serial para este crackme. Lo cargamos en el olly y miramos la zona donde teníamos el INT3 (0xCC) que nos molestaba:



Vemos que tenemos los 4 bytes que escribió WriteProcessMemory.

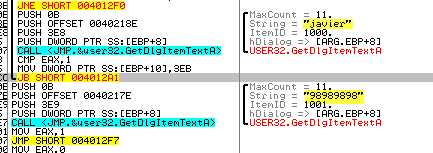
Si buscamos un 0xCC caemos acá, este nos quedo sin corregir porque el flujo de ejecución paso por otro lado, pero vamos a hacer caso omiso porque de seguro que con las pruebas que hacemos no caemos en esta porción de código.



Analicemos el algoritmo para obtener un serial valido, ponemos un Breakpoint en GetDlgItemTextA que es la API encargada de obtener nuestro texto del editbox.

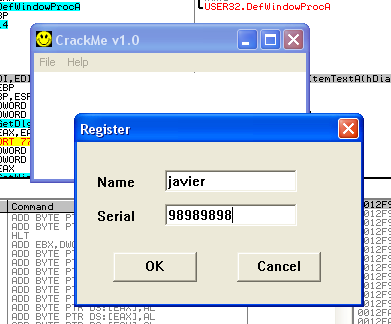
07-03-2012 21-29-45.png

Traceamos un poco con F7 luego de la llamada a la API y estamos en esta zona:

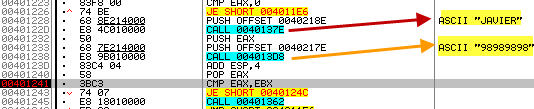


Vemos las llamadas a las API y también vemos nuestro nombre de usuario y serial fake.

Le cargamos algo para comenzar el análisis bien:



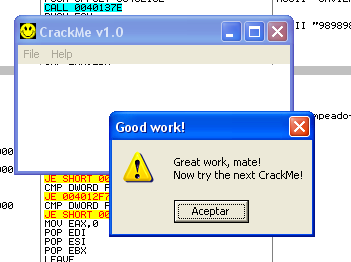
F7 hasta cansarnos y caemos en esta zona:



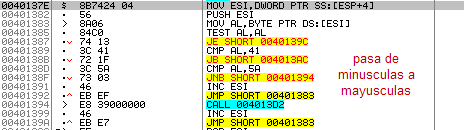
Esos CALL que se le pasan como argumento nuestro nombre de usuario y serial deben ser los que procesan nuestros datos y deciden si son buenos o no. Despues tenemos una comparación un tanto sospechosa y un JE que le sigue, para ver si es la zona caliente vamos a invertir el salto y ver si lo registramos con cualquier serial.

07-03-2012 21-34-46.png

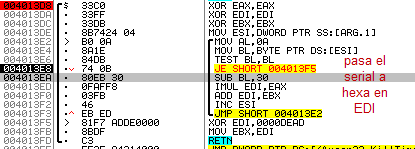
Probamos y…



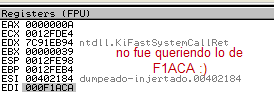
Lo registramos, pero el objetivo es obtener un serial valido, así que miremos un poco mas el algoritmo:



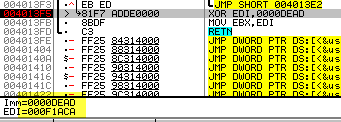
Esta parte pasa el serial de minúsculas a mayúsculas, obviamente si esta en mayúsculas este código no hace nada.



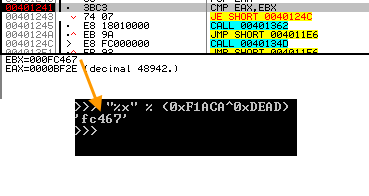
Pasa el serial a hexadecimal y una vez que termina el bucle deja el resultado en EDI



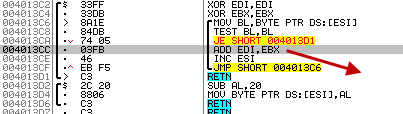
989898 en hexadecimal es 0x000F1ACA, eso fue pura casualidad. Ahora a ese valor le hace un XOR con 0x0000DEAD



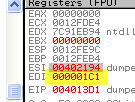
Y pasa a la comparación:



Ya tenemos lo que le aplico al serial pero falta la parte del procesamiento del nombre para revertir el algoritmo.

07-03-2012 21-49-10.png

Ese bucle suma letra por letra de nuestro nombre pasado a mayúsculas, miremos EDI como siempre:



Y después sale de la suma de letras y le hace un XOR con 0x0000BEEF:

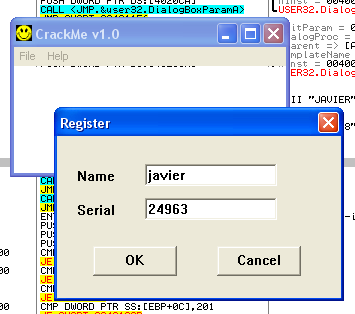
07-03-2012 21-49-49.png

Entonces tenemos que al serial lo pasa a entero y le hace un xor con 0x0000DEAD y al nombre suma todas las letras y lo xorea con 0x0000BEEF.

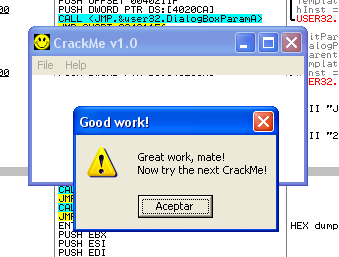
La inversa del xor es el propio xor así que usamos python para obtener un serial bueno:

07-03-2012 21-50-30.png

Lo probamos para ver si nuestros cálculos son correctos:



Y obtenemos:



Ya esta, problema solucionado. Rompimos la protección, analizamos el algoritmo y logramos hacer un keygen. Trabajo finalizado.