Aprende el Arte de la Ingeniería Inversa

Manuel Rey Vilar

http://masteracsi.ual.es/

Prefacio

Es increíble, y más bien desconcertante, la cantidad de software que ejecutamos sin saber a ciencia cierta lo que hace. Compramos el software situados en las estanterías envueltos en plástico transparente. Ejecutamos utilidades de configuración que instalan numerosos archivos, cambian la configuración del sistema, borran o desactivan las versiones anteriores, y modifican archivos críticos del registro. Cada vez que accedemos a un sitio Web, se puede invocar o interactuar con docenas de programas y segmentos de código que son necesarios para conseguir el aspecto y el comportamiento deseado. Compramos CDs con cientos de juegos y utilidades o los descargamos como shareware.

Descargamos actualizaciones e instalamos parches, confiando en que los vendedores están seguro de que los cambios son correctos y completos. Confiamos ciegamente que los últimos cambios aplicados al programa mantiene la compatibilidad con todo el resto de los programas en nuestro sistema. En definitiva confiamos en software que no entendemos y no conocemos del todo bien.

La realidad es que el Software de hoy en día se ha vuelto tan complejo e interconectado que el desarrollador a menudo no conoce todas las características y repercusiones de lo que ha sido creado en una aplicación. Con frecuencia es demasiado caro y consume mucho tiempo probar todas las rutas de control de un programa y todos los grupos de opciones de usuario.

Bajo este escenario aparece la ingeniería inversa como un conjunto de técnicas y herramientas para la comprensión de lo que realmente representa el software. Esto nos permite visualizar la estructura del software, sus modos de funcionamiento y las características que impulsan su comportamiento. Las técnicas de análisis y la aplicación de herramientas automatizadas para examinar el software, nos dan una forma razonable de ver y comprender la complejidad del software y para descubrir su verdad.

Tanto la memoria de este trabajo como el software desarrollado se distribuyen bajo la licencia GNU GPL v3.

La Licencia Pública General GNU (GNU GPL) es una licencia libre, sin derechos para software y otro tipo de trabajos.

Las licencias para la mayoría del software y otros trabajos prácticos están destinadas a suprimir la libertad de compartir y modificar esos trabajos. Por el contrario, la Licencia

Pública General GNU persigue garantizar su libertad para compartir y modificar todas las versiones de un programa--y asegurar que permanecerá como software libre para todos sus usuarios.

Cuando hablamos de software libre, nos referimos a libertad, no a precio. Las

Licencias Públicas Generales están destinadas a garantizar la libertad de distribuir copias de software libre (y cobrar por ello si quiere), a recibir el código fuente o poder conseguirlo si así lo desea, a modificar el software o usar parte del mismo en nuevos programas libres, y a saber que puede hacer estas cosas.

Para obtener más información sobre las licencias y sus términos puede consultar:

- http://www.gnu.org/licenses/gpl.html (Licencia original en inglés)

- http://www.viti.es/gnu/licenses/gpl.html (Traducción de la licencia al castellano)

ÍNDICE

CAPÍTULO 1 LENGUAJE ENSAMBLADOR...

CAPÍTULO 2 ARQUITECTURA X86...
2.1. La Pila ....
2.2. Los Registros...
2.2.1. Generales...
2.2.2. De Base ....
2.2.3. De Indice ....
2.2.4. De Puntero ....
2.2.5. De Segmento...
2.2.6. Flags...
2.3. Las Instrucciones..
2.3.1. Instrucciones de la pila ....
2.3.2. Instrucciones de transferencia de datos ....
2.3.3. Instrucciones aritméticas ....
2.3.4. Instrucciones lógicas..
2.3.5. Instrucciones de comprobación y verificación ....
2.3.6. Instrucciones de salto...
2.3.7. Instrucciones de subrutinas...
2.3.8. Instrucciones de bucle..
2.3.9. Instrucciones de cadenas...
2.3.10. Instrucciones de Entrada / Salida..
2.3.11. Instrucciones de rotación y desplazamiento..
2.3.12. Instrucciones de conversión..
2.3.13. Instrucciones de flags...
2.3.14. Instrucciones de interrupción...
2.3.15. Instrucciones del procesador..

CAPÍTULO 3 INTRODUCCIÓN A LA INGENIERIA INVERSA.......................................
3.1. ¿Qué es la Ingenieria Inversa? ....
3.2. Aplicaciones de la Ingeniería Inversa ....
3.3. Conocimientos previos necesarios ....
3.4. Herramientas utilizadas en Ingenieria Inversa ....
3.4.1. Desensambladores ....
3.4.2. Depuradores...
3.4.3. Editores Hexadecimales..

3.4.4. PE y Editores de recursos ....
3.4.5. Herramientas de monitorización del sistema ....
3.4.6. Herramientas e información misceláneos ....

CAPÍTULO 4 INTRODUCCIÓN A OLLYDBG....................
4.1. ¿Qué es Ollydbg? ....
4.2. Visión general ....
4.2.1.Desensamblador ....
4.2.2.Registros ....
4.2.3.La Pila (I) ....
4.2.4.Dump ....
4.2.5.La barra de herramientas ....
4.2.6.El menú contextual ....

CAPÍTULO 5 OLLYDBG (I) ....
5.1. Cargando una aplicación ....
5.2. Ejecutando una aplicación ....
5.3. Ejecutando paso a paso ....
5.4. Puntos de interrupción (Breakpoints) ....
5.4.1.Software Breakpoints ....
5.4.2.Hardware Breakpoints ....
5.4.3.Memory Breakpoints ....
5.5. Utilizando el panel Dump ....

CAPÍTULO 6 OLLYDBG (II) ....
6.1. ¿Qué son los DLL's? ....
6.2. ¿Cómo se usan los DLL's? ....
6.3. La tabla de saltos ....
6.4. Saltando dentro y fuera de los DLL's ....
6.5. La Pila (II) ....
6.6. Mostrando argumentos y variables locales ....
6.7. Ollydbg Cheatsheet ....

CAPÍTULO 7 EJERCICIOS RESUELTOS ....
7.1. Caso práctico 1: Buscando cadenas de texto ....
7.2. Caso práctico 2: Parches ....
7.3. Caso práctico 3: Repasando conceptos ....
7.4. Caso práctico 4: Resource Hacker ....
7.5. Caso práctico 5: Marco de Referencias ....
7.6. Caso práctico 6: Intermodular Calls ....
7.7. Caso práctico 7: Niveles de parcheo ....
7.8. Caso práctico 8: Introducción al Nivel 2 (Noob) ....
7.9. Caso práctico 9: Ejemplo de Noob Avanzado ....
7.10. Caso práctico 10: Crackeando un programa real ....
7.11. Caso práctico 11: NAGS ....
7.11.1 Nag1.exe ....
7.11.2 Nag2.exe ....
7.12. Caso práctico 12: Usando el Call Stack ....
7.13. Caso práctico 13: Los mensajes de las ventanas ....
7.14. Caso práctico 14: Auto-modificación del código ....
7.15. Caso práctico 15: Fuerza bruta ....
7.16. Caso práctico 16: Los binarios de Delphi ....
7.16.1 DelphiCrackme.exe ....
7.16.2 exif2htm.exe ....
7.17. Caso práctico 17: Periodos de prueba y Hardware Breakpoints ....

7.18. Caso práctico 18: Generador de parches ....
7.19. Caso práctico 19: Trabajando con binarios de Visual Basic (I) ....
7.20. Caso práctico 20: Trabajando con binarios de Visual Basic (II) ....
7.21. Caso práctico 21: Técnicas anti-depurador (anti-debugging) ....

CAPÍTULO 8 EJERCICIOS COMPLEMENTARIOS ....
8.1. Otra forma de poner un parche ....
8.2. Bypasear un archivo de claves ....
8.3. Bypasear un serial en Delphi ....
8.4. Cracking Driverfinder ....
8.5. Desembalaje rápido de PECompact ....
8.6. Crackme .NET ....
8.7. Multi-Parche ....

ANEXO I SISTEMAS DE NUMERACIÓN ....
I.I. Decimal ....
I.II. Binario ....
I.III Hexadecimal ....
I.IV. Octal ....
I.V. Conversión ....
I.VI. Identificación ....

ANEXO II SISTEMAS DE REPRESENTACIÓN ....
II.I. Números negativos ....
II.II Coma (o punto) Flotante ....
II.III. Formato BCD…………..………………………………………………………
II.IV. Carateres ASCII - ANSI ....

ANEXO III OPERACIONES LÓGICAS ....
III.I. And ....
III.II. Or ....
III.III. Xor ....
III.IV. Not ....

Capítulo 1

LENGUAJE ENSAMBLADOR

Se define como un lenguaje de programación que se usa para dar directamente órdenes al ordenador. A diferencia de otros lenguajes, que usan el sistema operativo como intermediario para ejecutar las tareas (le dicen al sistema operativo que haga una cosa y este es quien se encarga de hacérselo saber al ordenador), el lenguaje de ensamblador (en inglés assembly) no usa intermediarios, sino que directamente le da las órdenes a la máquina.

Evidentemente este lenguaje tiene varias ventajas e inconvenientes. Entre las ventajas tenemos que por supuesto, es más rápido y eficiente, más estable y requiere menos recursos.

Las desventajas son sin embargo mucho mayores: para empezar está limitado a las órdenes que el ordenador traiga memorizadas en su placa base, nada más. Y requiere unos conocimientos técnicos muy avanzados, toda vez que cada CPU incluye órdenes distintas e incluso formas distintas de darle esas órdenes, variando según el modelo, el fabricante... etc... Es por ello, entre otras cosas, que no se ha impuesto en el mercado como lenguaje de programación para aplicaciones o juegos.

Como se ve es un lenguaje de programación orientado a profesionales que trabajan con hardware, muy útil sobre todo para detectar fallos e incompatibilidades entre piezas del PC o chequear el estado de estas.

Como ya dijimos el lenguaje ensamblador debe ser traducido a código máquina para funcionar, y por tanto requiere un ensamblador que lo traduzca a código binario (a ceros y unos). Pero es el lenguaje más cercano al propio código que usan los ordenadores. Es por ello que se dice que es un lenguaje de bajo nivel, debido a que está sólo un escalón por encima del código máquina, y solo permite ejecutar instrucciones sencillas. Los lenguajes de alto nivel son más elaborados y permiten realizar tareas múltiples y complejas con una sola orden, mientras que en el que nos ocupa una orden equivale a una instrucción directa.

Existen como hemos visto muchas variantes del lenguaje ensamblador, de hecho una por cada tipo de arquitectura del CPU. Las primeras CPUs apenas tenían instrucciones y por tanto eran mucho más sencillas. En la actualidad están en boga los llamados ensambladores de

alto nivel, que permiten instrucciones más complejas e incluso cadenas de instrucciones (macros).

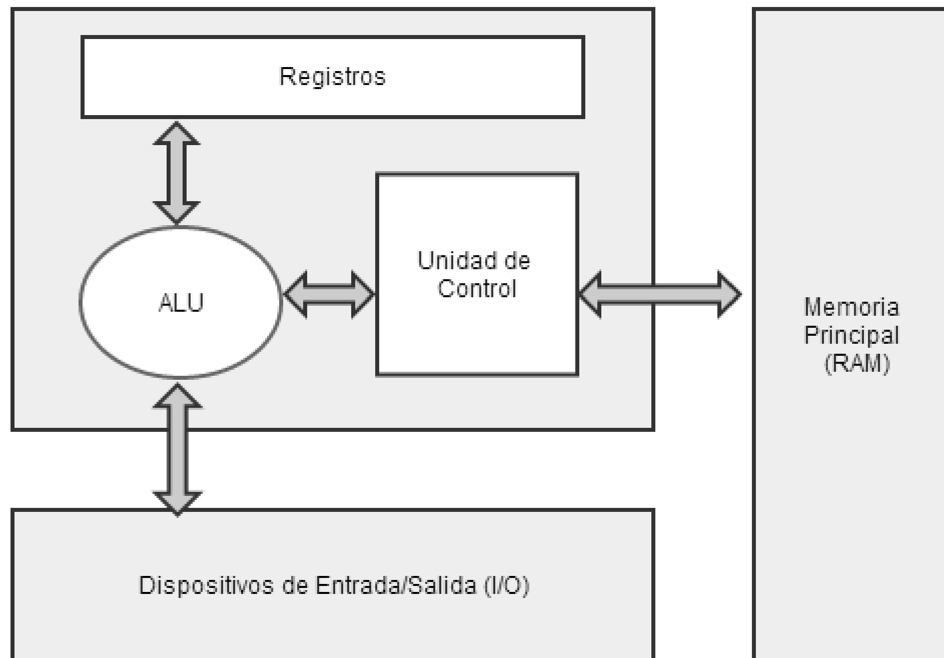
Capítulo 2

ARQUITECTURA X86

Las arquitecturas de las computadoras más modernas siguen, hasta hoy en día, la arquitectura de Von Neumann, cuentan con tres componentes principales, que a pesar del paso de los años no han cambiado:

- **CPU**: Es la unidad central de procesamiento, se encarga de ejecutar el código del programa o Sistema Operativo.
- **Memoria Principal (RAM)**: Almacena los datos y el código que se cargarán en la CPU para su ejecución.
- **Sistema de Entrada y Salida**: Es la interfaz con los dispositivos como discos rígidos, teclados, monitores y demás.

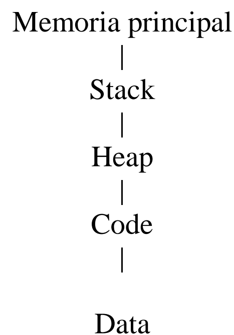
La figura siguiente muestra una presentación gráfica de la estructura de la memoria:



A modo general, dentro de la estructura de esta arquitectura, la CPU contiene ciertos componentes para realizar tareas específicas. La *unidad de control* recibe desde la memoria RAM las instrucciones que debe ejecutar a través de un registro en particular, el *Instruction Pointer* (IP), que apunta a la próxima instrucción a ejecutar. Los registros, son utilizados por la CPU para almacenar datos, valores o direcciones de memoria que acortan el tiempo que tardaría la CPU en ir a buscarlos directamente a la RAM. Existen registros que cumplen funciones específicas y otros de uso general.

Una vez que se cuentan con las instrucciones la ALU (*Aritmetic Logic Unit*) es la encargada de ejecutarlas y almacenar el resultado directamente en la memoria RAM o en los registros. Este proceso de búsqueda y ejecución de una instrucción tras otras se repite a medida que se ejecuta un programa.

La memoria de un programa se puede dividir en 4 secciones principales:



- **Data**: La sección de datos de un programa hace referencia a una región específica de memoria. Contiene lo que se conoce como las variables estáticas que no cambian con la ejecución del programa. También en esta sección se encuentran las variables globales, que están disponibles desde cualquier parte del programa.
- **Code**: En esta región de memoria se almacena el código que se ejecuta del programa donde se alojan todas las instrucciones que se van a ejecutar.
- **Heap**: El *heap* es una región de memoria que se utiliza para asignar nuevos valores durante la ejecución del programa así como para eliminarlos una vez que se dejaron de utilizar. El heap es una memoria dinámica y su contenido varía a medida que se ejecuta el programa
- **Stack** (Pila): La pila se utiliza para alojar las variables locales, parámetros y valores de retorno de una función como así también contiene las direcciones de retorno entre una llamada a una función y otra, siendo muy útil para controlar el flujo de ejecución del programa.

Algo importante a tener en cuenta es que estas regiones de memoria no se encuentran en zonas contiguas de memoria y que su ubicación puede cambiar, estando en regiones más bajas o altas de memoria.

2.1 La Pila

La pila es un lugar de la memoria donde se van guardando determinados valores para recuperarlos posteriormente. Por esta razón, cuando se introduce un nuevo valor no se hace en el lugar ocupado por el valor introducido anteriormente, sino que se pone justo en la posición o posiciones inmediatamente anteriores a la ocupada por ese valor previo.

La pila sigue la norma LIFO (last in, first out) y funciona como una pila de platos. Si colocamos uno a uno cinco platos sobre una pila, y luego los vamos cogiendo, tomaremos en primer lugar el último que hayamos puesto, luego el penúltimo etc., hasta llegar al que hayamos puesto en primer lugar, que será el último que cogeremos. La dirección para acceder al plato superior de la pila, es decir al valor en que podemos acceder en cada momento, está contenida en un registro (ESP) que va variando según se van añadiendo o retirando valores.

Hay diferentes maneras de modificar el estado de la pila: directamente, mediante las instrucciones que se emplean para poner o quitar valores, o mediante instrucciones que tienen el efecto de modificarla, como la instrucción call (llamada a subrutina) que tiene como efecto secundario el guardar en la pila la dirección a la que ha de volver el programa cuando, una vez terminada la ejecución de la subrutina, se encuentra con la instrucción ret, la cual retira la dirección introducida de la pila, dejándola como estaba antes de la ejecución de la subrutina.

La finalidad principal de la pila es liberar registros de forma temporal para que puedan realizar otras funciones para luego, una vez terminadas las mismas, reintegrarles su valor primitivo. Otra utilidad puede ser la que a veces emplean programadores temerosos de que los chicos malos puedan robarles sus tesoros, y es la de saltar de un punto a otro durante la ejecución de un programa sin que esto quede reflejado en el desensamblado de dicho programa.

2.2 Los Registros

Los registros son elementos de almacenamiento de datos contenidos en el procesador y que tienen la ventaja de la rapidez de acceso y la finalidad de contener datos necesarios para la ejecución del programa. En principio, casi todos ellos pueden utilizarse libremente, pero cada uno de ellos tiene sus funciones específicas. Existen distintos tipos de registro:

2.2.1. Generales

El registro **EAX** (Acumulador) además de su empleo como registro para uso general, es utilizado en algunas instrucciones como las de multiplicar y dividir, que como se verá más adelante lo utilizan como factor y como resultado. También se utiliza para contener el valor de retorno después de la ejecución de una API, por ejemplo, al regreso de lstrlen, EAX contiene la longitud de la cadena de texto examinada, y al regreso de RegQueryValueExA, EAX estará puesto a cero si el registro se ha leído correctamente.

El registro **EBX** (Base) además de su uso general, suele utilizarse para direccionar el acceso a datos situados en la memoria.

El registro **ECX** (Contador) además de su uso general, se utiliza como contador en determinadas instrucciones.

El registro **EDX** (Datos) además de su uso general, se utiliza junto con EAX para formar números mayores de 32 bits en algunas instrucciones, como las de multiplicar y dividir. También se utiliza en operaciones de Entrada/Salida.

Todos estos registros son de 32 bits, aunque se puede operar con subdivisiones de ellos. Así por ejemplo, los 16 bits de menor valor del registro EAX, se pueden manejar como AX, y este AX, a su vez, se divide en AH (bits 5º a 8º contados a partir de la derecha) y AL (bits 1º a 4º).

2.2.2. De base

El registro **EBX** se utiliza como registro de uso general.

El registro **EBP** se utiliza para direccionar el acceso a datos situados dentro del espacio ocupado por la pila.

El registro EBP también es de 32 bits y se puede utilizar para uso general. Puede operar con los 16 bits inferiores: BP.

2.2.3. De índice

Los registros **ESI** y **EDI** se utilizan en instrucciones que acceden a grupos de posiciones contiguas de la memoria que funcionan como origen y destino. El registro ESI apunta a la dirección del primer (o último, como veremos al tratar el flag D) byte del origen y el EDI al primero (o último) del destino.

Como los anteriores, estos registros son de 32 bits y se pueden utilizar para uso general.

Puede operarse con los 16 bits inferiores de cada uno: SI y DI.

2.2.4. De puntero

El registro **ESP** apunta a la dirección del último valor introducido en la pila (o sea, el primero que podemos sacar) y aunque puede ser modificado mediante las instrucciones del programa (por ejemplo: sumándole o restándole un determinado valor), es más corriente que se modifique automáticamente según las entradas y salidas de valores en la pila.

El registro **EIP** apunta a la dirección del código de la instrucción a ejecutarse y se va modificando según se va ejecutando el código del programa.

Estos registros también son de 32 bits, pero tanto ESP como EIP no se pueden utilizar para otros usos que los previstos.

2.2.5. De segmento

El registro **CS** contiene el segmento que corresponde a la dirección del código de las instrucciones que forman el programa. Así la dirección de la siguiente instrucción a ejecutarse de un programa vendría determinada por los registros CS:EIP

Los registros **DS** y **ES** se utilizan en instrucciones que tienen origen (DS:ESI) y destino (ES:EDI).

El registro **SS** es el que contiene el segmento que corresponde a la dirección de la pila. Así la dirección del valor que se puede extraer de la pila en un momento dado, estaría indicada por SS:ESP.

Los registros **FS** y **GS** son de uso general.

Estos registros son de 16 bits y no se pueden utilizar para otros usos que los previstos.

2.2.6. Flags

Están situados en un registro (EFLAGS) de 32 bits de los cuales se utilizan 18, y de estos vamos a ver sólo 8.

O (Overflow o desbordamiento)

Este flag se pone a uno, cuando se efectúa una operación cuyo resultado, debido a operar con números en complemento a dos, cambia de signo, dando un resultado incorrecto.

D (Dirección)

Si está a cero, las instrucciones que utilizan los registros ESI y EDI por operar con una serie de bytes consecutivos, los tomarán en el orden normal, del primero al último; en caso

contrario, los tomaran del último al primero. Por lo tanto este flag no varía de acuerdo a los resultados de determinadas operaciones, sino que se le pone el valor adecuado a voluntad del programador, mediante las instrucciones std y cld que se verán más adelante.

I (Interrupción)

Cuando se pone a uno, el procesador ignora las peticiones de interrupción que puedan llegar de los periféricos. La finalidad de esto es la de evitar que durante la ejecución de algún proceso más o menos crítico, una interrupción pudiera provocar la inestabilidad del sistema.

S (Signo)

Se pone a uno, cuando se efectúa una operación cuyo resultado es negativo.

Z (Cero)

Se pone a uno, cuando se efectúa una operación cuyo resultado es cero.

A (Auxiliar)

Este flag es similar al de carry que veremos enseguida, pero responde a las operaciones efectuadas con números en formato BCD.

P (Paridad)

Se pone a uno, cuando se efectúa una operación cuyo resultado contiene un número par de bits con el valor 1.

C (Carry o acarreo)

Se pone a uno, cuando se efectúa una operación que no cabe en el espacio correspondiente al resultado.

2.3 Las Instrucciones

Es el momento de empezar a explicar con cierto detenimiento las instrucciones de los procesadores 80X86.

Antes de hablar de las instrucciones, se explicará los distintos sistemas que tienen las instrucciones para acceder a los datos necesarios para su ejecución.

2.3.1. Modos de direccionamiento

La terminología que se emplea aquí no es standard, por lo que puede ser que algunos términos no sean los mismos que los empleados en otros lugares.

Directo El dato se refiere a una posición de la memoria, cuya dirección se escribe entre corchetes.

```
mov dword ptr [00513450], ecx

mov ax, word ptr [00510A25]

mov al, byte ptr [00402811]
```

En el primer ejemplo, se copia el contenido del registro ECX, no en la dirección 513450, sino en la dirección que hay guardada en la posición 513450 de la memoria.

Cabe mencionar aquí el tratamiento de los typecasts, es decir, de la especificación del tamaño del dato que se obtiene de la dirección de memoria indicada entre corchetes. En este ejemplo no es necesaria esta especificación, ya que la longitud del dato viene dada por el tipo de registro. En otros casos puede existir una ambigüedad que haga necesaria esta información. Por ejemplo, si tratamos de ensamblar la instrucción mov [edx], 00000000 el compilador nos dará error porque no se ha indicado el tamaño del valor a introducir en EDX (aunque se hayan puesto ocho ceros), por lo que se deberá especificar: mov dword ptr [edx], 0 lo que generaría una instrucción que pondrá cuatro bytes a cero a partir de la dirección indicada por el registro EDX.

Indirecto Aquí el dato también se refiere a una posición de la memoria, pero en vez de indicar la dirección como en el caso anterior, se indica mediante un registro que contiene dicha dirección.

mov eax, 00513450

mov dword ptr [eax], ecx

En este ejemplo, el resultado es el mismo que en el primer ejemplo del modo directo.

Registro Los parámetros de la instrucción están contenidos en registros.

inc eax

push ebx

xchg eax, ebx

Inmediato Se le asigna un valor numérico a un operando de la instrucción:

cmp eax, 00000005

mov cl, FF

De base Este direccionamiento emplea exclusivamente los registros EBX y EBP, que sumados a un valor numérico, nos dan la posición de memoria origen o destino del dato que maneja la instrucción.

cmp ebx, dword ptr [ebp+02]

mov edx, dword ptr [ebx+08]

De índice Similar al anterior, pero empleando los registros de índice: ESI y EDI.

mov cl, byte ptr [esi+01]

cmp cl, byte ptr [edi+01]

2.3.2. La reina de las instrucciones

Las instrucciones se representan de dos maneras: mediante el opcode o código de operación y el nombre o mnemónico. El código de operación es un grupo de cifras más o menos largo, según la instrucción, que en principio está pensado para uso del ordenador. El mnemónico es el nombre de la instrucción, que trata de representar una descripción del significado de la misma. Para apreciar mejor esto, junto al mnemónico de la instrucción se va a poner el nombre en inglés.

A partir de ahora, en los ejemplos, se pondrán los códigos de operación de las instrucciones (simplemente a título orientativo) y sólo cuando sea conveniente (en el caso de saltos), la dirección en que están situadas.

nop (No Operation)

Esta instrucción es de gran utilidad: no hace nada. Su verdadera utilidad reside en rellenar los huecos provocados por la eliminación de instrucciones o su substitución por instrucciones de menor longitud. Su nombre da origen al verbo "nopear" que tan a menudo utilizan los chicos malos. Hay corrientes filosóficas dentro del cracking que sostienen que poner nops es poco elegante y por ello substituyen por ejemplo, una instrucción de salto condicional de dos bytes como:

```
:0040283E 7501      jne 00402841
:00402840 C3        ret
:00402841 31C0      xor eax, eax
```

por:

```
:0040283E 40        inc eax
:0040283F 48        dec eax
:00402840 C3        ret
:00402841 31C0      xor eax, eax
```

en lugar de:

```
:0040283E 90        nop
:0040283F 90        nop
:00402840 C3        ret
:00402841 31C0      xor eax, eax
```

Otro forma más elegante de solucionarlo es cambiando un solo byte, consiguiendo de esta forma que tanto si se efectúa el salto como si no, se vaya a la instrucción siguiente.

```
:0040283E 7500      jne 00402840
```

:00402840 C3 ret

:00402841 31C0 xor eax, eax

Cabe mencionar que el opcode que corresponde a dicha instrucción (90) es en realidad el correspondiente a la instrucción xchg eax,eax, y por lo tanto la instrucción nop no existe, sino que se trata sólo de una una interpretación convencional. Otra cosa curiosa es que el opcode 6690 que corresponde a la instrucción xchg ax,ax, se desensambla también como nop.

2.3.3. Instrucciones de la pila

En la descripción de la pila, se ha visto que hay instrucciones que se utilizan para poner y quitar valores de ella, cambiando su dirección de inicio contenida en el registro ESP, de manera que siempre apunte al último valor introducido.

push (Push Word or Doubleword Onto the Stack)

Esta instrucción resta del registro ESP, la longitud de su único operando que puede ser de tipo word o doubleword (4 u 8 bytes), y a continuación lo coloca en la pila.

pop (Pop a Value from the Stack)

Es la inversa de la anterior, es decir que incrementa el registro ESP y retira el valor disponible de la pila y lo coloca donde indica el operando.

pushad /pusha (Push All General-Purpose Registers 32 / 16 bits)

Estas instrucciones guardan el contenido de los registros en la pila en un orden determinado. Así pues, pushad equivale a: push EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI y pusha equivale a: push AX, CX, DX, BX, SP, BP, SI, DI.

popad / popa (Pop All General-Purpose Registers 32 / 16 bits)

Estas instrucciones efectúan la función inversa de las anteriores, es decir, restituyen a los registros los valores recuperados de la pila en el orden inverso al que se guardaron. Así popad equivale a: pop EDI, ESI, EBP, ESP, EBX, EDX, ECX, EAX y popa equivale a: pop DI, SI, BP, SP, BX, DX, CX, AX (los valores recuperados correspondientes a ESP y SP, no se colocan en los registros sino que se descartan).

pushfd /pushf (Push EFLAGS Register onto the Stack 32 / 16 bits)

popfd / popf (Pop Stack into EFLAGS Register 32 / 16 bits)

Estas parejas de instrucciones colocan y retiran de la pila el registro de flags.

2.3.4. Instrucciones de transferencia de datos

mov (Move)

Esta instrucción tiene dos operandos. Copia el contenido del operando de origen (representado en segundo lugar) en el de destino (en primer lugar), y según el tipo de estos operandos adopta formatos distintos. He aquí unos ejemplos:

6689C8	mov ax, cx
8BC3	mov eax, ebx
8B5BDC	mov ebx, dword ptr [ebx-24]
893438	mov dword ptr [eax+edi], esi

movsx (Move with Sign-Extension)

Copia el contenido del segundo operando, que puede ser un registro o una posición de memoria, en el primero (de doble longitud que el segundo), rellenándose los bits sobrantes por la izquierda con el valor del bit más significativo del segundo operando. Aquí tenemos un par de ejemplos:

33C0	xor eax, eax	
BB78563412	mov ebx, 12345678	
0FBFC3	movsx eax, bx	EAX=00005678
33C0	xor eax, eax	
BBCDAB3412	mov ebx, 1234ABCD	
0FBFC3	movsx eax, bx	EAX=FFFFABCD

Vemos como en el primer ejemplo los espacios vacíos se rellenan con ceros y en el segundo con unos.

movzx (Move with Zero-Extend)

Igual que movsx, pero en este caso, los espacios sobrantes se rellenan siempre con ceros. Veamos como el segundo ejemplo de la instrucción anterior da un resultado distinto:

33C0 xor eax, eax

BBCDAB3412 mov ebx, 1234ABCD

0FB7C3 movzx eax, bx EAX=0000ABCD

lea (Load Effective Address)

Similar a la instrucción mov, pero el primer operando es un registro de uso general y el segundo una dirección de memoria. Esta instrucción es útil sobre todo cuando esta dirección de memoria responde a un cálculo previo.

8D4638 lea eax, dword ptr [esi+38]

xchg (Exchange Register/Memory with Register)

Esta instrucción intercambia los contenidos de los dos operandos.

87CA xchg edx, ecx

870538305100 xchg dword ptr [00513038], eax

8710 xchg dword ptr [eax], edx

En el primer ejemplo, el contenido del registro ECX, se copia en el registro EDX, y el contenido anterior de EDX, se copia en ECX. Se obtendría el mismo resultado con:

51 push ecx

52 push edx

59 pop ecx

5A pop edx

La instrucción pop ecx toma el valor que hay en la pila y lo coloca en el registro ECX, pero como podemos ver por la instrucción anterior push edx, este valor, procedía del registro EDX. Luego se coloca en EDX el valor procedente de ECX.

bswap (Byte Swap)

Esta instrucción se emplea exclusivamente con registros de 32 bits como único parámetro. Intercambia los bits 0 a 7 con los bits 24 a 31, y los bits 16 a 23 con los bit 8 a 15.

B8CD34AB12 mov eax, 12AB34CD EAX=12AB34CD

0FC8 bswap eax EAX=CD34AB12

2.3.5. Instrucciones aritméticas

inc (Increment by 1) / **dec** (Decrement by 1)

Estas dos instrucciones incrementan y decrementan respectivamente el valor indicado en su único operando.

FF0524345100	inc dword ptr [00513424]
FF0D24345100	dec dword ptr [00513424]
40	inc eax
4B	dec ebx

En los dos primeros ejemplos, se incrementaría y decrementaría el valor contenido en los cuatro bytes situados a partir de la dirección 513424.

add (Add)

Esta instrucción suma los contenidos de sus dos operandos y coloca el resultado en el operando representado en primer lugar.

02C1	add al, cl	AL + CL -> AL
01C2	add edx, eax	EDX + EAX -> EDX
8345E408	add dword ptr [ebp-1C], 0000008	dword ptr [EBP-1C]+ 8 -> [EBP-1C]

En el tercer ejemplo, como el resultado se coloca siempre en el primer operando, no sería aceptada por el compilador una instrucción como add 00000008, dword ptr [ebp-1C]

adc (Add with Carry)

Esta instrucción es similar a la anterior, con la diferencia de que se suma también el valor del flag de acarreo. Se utiliza para sumar valores mayores de 32 bits. Supongamos que queremos sumar al contenido de los registros EDX:EAX (EDX=00000021h y EAX=87AE43F5), un valor de más de 32 bits (3ED671A23). Veamos cómo se hace:

add eax, ED671A23	EAX=75155E18	(87AE43F5+ED671A23)	CF=1
adc edx, 0000003	EDX=25h	(21h+3+1)	

sub (Subtract)

Esta instrucción resta el contenido del segundo operando del primero, colocando el resultado en el primer operando.

83EA16	sub edx, 00000016	EDX - 16 -> EDX

29C8	sub eax, ecx	EAX - ECX -> EAX
2B2B	sub ebp, dword ptr [ebx]	EBP - dword ptr [EBX] -> EBP

sbb (Integer Subtraction with Borrow)

Esta instrucción es una resta en la que se tiene en cuenta el valor del flag de acarreo. Supongamos que del contenido de los registros EDX:EAX después del ejecutado el ejemplo de la instrucción adc (EDX=00000025h y EAX=75155E18), queremos restar el valor 3ED671A23. El resultado es el valor que tenían inicialmente los dos registros en el ejemplo citado:

sub eax, ED671A23	EAX=87AE43F5	(75155E18-ED671A23)	CF=1
sbb edx, 0000003	EDX=21h	(25h-3-1)	

mul (Unsigned Multiply) / **imul** (Signed Multiply)

Estas dos instrucciones se utilizan para multiplicar dos valores. La diferencia más importante entre las dos, es que en la primera no se tiene en cuenta el signo de los factores, mientras que en la segunda sí. Como veremos en algunos ejemplos, esta diferencia se refleja en los valores de los flags.

En la instrucción mul, hay un solo operando. Si es un valor de tamaño byte, se multiplica este valor por el contenido de AL y el resultado se guarda en EAX, si el valor es de tamaño word (2 bytes), se multiplica por AX, y el resultado se guarda en EAX y finalmente, si el valor es de tamaño dword (4bytes), se multiplica por EAX y el resultado se guarda en EDX:EAX. De esta forma el espacio destinado al resultado siempre es de tamaño doble al de los operandos.

F7E1	mul ecx	EAX*ECX -> EDX:EAX
F72424	mul dword ptr [esp]	EAX*[ESP] -> EDX:EAX

En la instrucción imul, hay también una mayor variedad en el origen de sus factores. Además de la utilización de los registros EAX y EDX, así como de sus subdivisiones, pueden especificarse otros orígenes y destinos de datos y puede haber hasta tres operandos. El primero, es el lugar donde se va a guardar el resultado, que debe ser siempre un registro, el segundo y el tercero son los dos valores a multiplicar. En estos ejemplos vemos como estas instrucciones con dos o tres operandos, tienen el mismo espacio para el resultado que para cada uno de los factores:

F7EB	imul ebx	EAX x EBX -> EDX:EAX
696E74020080FF	imul ebp, dword ptr [esi+74], FF800002	[ESI+74] x FF800002 -> EBP
0FAF55E8	imul edx, dword ptr [ebp-18]	EDX x [EBP-18] -> EDX

Veamos finalmente la diferencia entre la multiplicación con signo y sin él:

66B8FFFF	mov ax, FFFF	AX=FFFF		
66BBFFFF	mov bx, FFFF			
66F7E3	mul bx	AX=0001	OF=1	CF=1

66B8FFFF	mov ax, FFFF			
66F7EB	imul bx	AX=0001	OF=0	CF=0

En la primera operación, como se consideran los números sin signo, se ha multiplicado 65535d por 65535d y el resultado ha sido 1. Debido a este resultado "anómalo", se han activado los flags OF y CF. En cambio, en la segunda operación, en la que se toma en cuenta el signo, se ha multiplicado -1 por -1 y el resultado ha sido 1. En este caso no se ha activado ningún flag. Otro ejemplo:

B8FFFFFF7F	mov eax, 7FFFFFFF			
BB02000000	mov ebx, 00000002			
F7E3	mul ebx	EAX=FFFFFFFE	OF=0	CF=0
B8FFFFFF7F	mov eax, 7FFFFFFF			
F7EB	imul ebx	EAX=FFFFFFFE	OF=1	CF=1

Esta vez, en el primer caso, el resultado ha sido correcto, porque 2147483647d multiplicado por dos ha dado 4294967294d, por tanto, no se ha activado ningún flag. Pero en el segundo caso, teniendo en cuenta el signo, hemos multiplicado 2147483647d por dos y nos ha dado como resultado -2. Ahora si se han activado los flags.

div (Unsigned Divide) / **idiv** (Signed Divide)

El caso de la división es muy parecido al de la multiplicación. Hay dos instrucciones: div para números en los que no se considere el signo e idiv para números que se consideren con signo. El dividendo está formado por una pareja de registros y el divisor es el único operando. He aquí varios ejemplos de una y otra instrucción:

66F7F3	div bx	DX:AX : BX -> AX	resto -> DX
F7F3	div ebx	EDX:EAX : EBX -> EAX	resto -> EDX
F77308	div dword ptr [ebx+08]	EDX:EAX : [EBX+8] -> EAX	resto -> EDX
F7F9	idiv ecx	EDX:EAX : ECX -> EAX	resto -> EDX

Ahora, como hemos hecho con la multiplicación, vamos a ver el diferente resultado que se obtiene empleando una u otra instrucción:

33D2	xor edx, edx		
66B80100	mov ax, 0001		
66BBFFFF	mov bx, FFFF		
66F7F3	div bx	AX=0000	DX=0001
33D2	xor edx, edx		
66B80100	mov ax, 0001		

66F7FB idiv bx AX=FFFF DX=0000

En el primer caso, al no considerar el signo de los números, se ha dividido 1 por 65535, que ha dado un cociente de 0 y un resto de 1. En el segundo caso, se ha dividido -1 por 1, lo que ha dado un cociente de -1 y un resto de 0. No ha habido overflow ni acarreo en ninguno de los dos casos.

xadd (Exchange and Add)

Intercambia los valores de los dos operandos y los suma, colocando el resultado en el primer operando. El primer operando puede ser un registro o una posición de memoria, pero el segundo sólo puede ser un registro.

C703CDAB3412 mov dword ptr [ebx], 1234ABCD

En la dirección indicada por EBX tendremos el valor CD AB 34 12.

Vemos el valor que hemos puesto en la memoria invertido, porque el paso del valor de un registro a la memoria y viceversa se hace empezando por el último byte y terminando por el primero.

B8CD34AB12 mov eax, 12AB34CD

B934120000 mov ecx, 00001234

0FC1C8 xadd eax, ecx

EAX contiene el valor 12AB4701 (12AB34CD+1234) y ECX el valor 12AB34CD.

B934120000 mov ecx, 00001234

0FC10B xadd dword ptr [ebx], ecx

La dirección indicada por EBX contiene el valor 01 BE 34 12 (1234ABCD+1234) y el registro ECX el valor 1234ABCD.

aaa (Ascii Adjust for Addition) / **daa** (Decimal Adjust AL after Addition)

Antes hemos mencionado el formato BCD, ahora vamos a operar con él. Mediante la primera de estas dos instrucciones, se ajusta el resultado de una suma de números en formato BCD desempaquetado, contenido en AL. La segunda instrucción hace lo mismo pero con formato BCD empaquetado. Vamos a ver un par de ejemplos:

33C0 xor eax, eax

33DB xor ebx, ebx

B007 mov al, 07

B309 mov bl, 09

02C3	add al, bl	AL=10	10h=7+9
37	aaa	AX=0106	16d(7+9)en BCD desempaq.)
33C0	xor eax, eax		
B065	mov al, 65		
B328	mov bl, 28		
02C3	add al, bl	AL=8D	(8Dh=65h+28h)
27	daa	AL=93	93d(65d+28d)en BCD empaquetado.)

Podemos ampliar un poco las posibilidades de daa aprovechando el flag de acarreo:

B065	mov al, 65		
B398	mov bl, 98		
02C3	add al, bl	AL=FD	(FDh=65h+98h)
27	daa	AL=63	CF=1
80D400	adc ah, 00	AX=0163	163d(65d+98d)en BCD empaquetado.)

aas (ASCII Adjust AL After Subtraction) / **das** (Decimal Adjust AL after Subtraction)

Estas instrucciones hacen lo mismo, pero después de una resta.

aam (Ascii Adjust AX After Multiply)

Esta instrucción ajusta el resultado de una multiplicación entre dos números de una cifra en formato BCD desempaquetado. Sin embargo, tiene una interesante posibilidad: la de poder trabajar en la base que se defina modificando MANUALMENTE (resulta que no hay un mnemónico para eso) el segundo byte del código que normalmente tiene el valor 0Ah (10d). Veamos unos ejemplos:

33C0	xor eax, eax	
33DB	xor ebx, ebx	
B009	mov al, 09	
B305	mov bl, 05	
F6E3	mul bl	AL=2D

D40A	aam	AX=0405	45d(9x5)en BCD desempaq.)
33C0	xor eax, eax		
B009	mov al, 09		
B305	mov bl, 05		
F6E3	mul bl	AL=2D	
D408	aam (base8)	AX=0505	55q=45d(9x5)en BCD desempaq.)

aad (Ascii Adjust AX Before Division)

Igual, pero con la división. La diferencia está en que se ejecuta la instrucción antes de dividir.

33C0	xor eax, eax		
33DB	xor ebx, ebx		
66B80307	mov ax, 0703		
D50A	aad	EAX=0049	49h=73d
B308	mov bl, 08		
F6F3	div bl	EAX=0109	73d/8=9 (AL) resto=1 (AH)
33C0	xor eax, eax		
33DB	xor ebx, ebx		
66B80307	mov ax, 0703		
D508	aad (base=8)	EAX=003B 3Bh=73q	
B308	mov bl, 08		
F6F3	div bl	AL=0307	73q/8=7 (AL) resto=3 (AH)

La única utilidad de estas instrucciones, es la de pasar un número en base hexadecimal a otra base.

neg (Two's Complement Negation)

Esta instrucción tiene la finalidad de cambiar de signo el número representado por su único operando, mediante una operación de complemento a dos.

B81C325100 mov eax, 0051321C

F7D8 neg eax EAX=FFAECDE4

neg 0000 0000 0101 0001 0011 0010 0001 1100 = 0051321C

1111 1111 1010 1110 1100 1101 1110 0011

\+ 1

1111 1111 1010 1110 1100 1101 1110 0100 = FFAECDE4

2.3.6. Instrucciones lógicas

and (Logical AND)

Efectúa una operación AND entre cada uno de los bits de los dos operandos.

B8DAC70704 mov eax, 0407C7DA

25C30E6F00 and eax, 006F0EC3 EAX=000706C2

0407C7DA = 0000 0100 0000 0111 1100 0111 1101 1010

006F0EC3 = 0000 0000 0110 1111 0000 1110 1100 0011

and 0000 0000 0000 0111 0000 0110 1100 0010 = 000706C2

25FF000000 and eax, 000000FF EAX=000000C2

000706C2 = 0000 0000 0000 0111 0000 0110 1100 0010

000000FF = 0000 0000 0000 0000 0000 0000 1111 1111

and0000 0000 0000 0000 0000 0000 1100 0010 = 000000C2

Hacer un AND de cualquier byte con FF lo deja como estaba. Por esta razón, se utilizan frecuentemente instrucciones de este tipo para eliminar bytes que no se van a utilizar de un registro.

or (Logical Inclusive OR)

Efectúa una operación OR entre cada uno de los bits de los dos operandos.

B8DAC70704 mov eax, 0407C7DA

0DC30E6F00	or eax, 006F0EC3	EAX=046FCFDB

0407C7DA = 0000 0100 0000 0111 1100 0111 1101 1010

006F0EC3 = 0000 0000 0110 1111 0000 1110 1100 0011

or 0000 0100 0110 1111 1100 1111 1101 1010 = 046FCFDB

xor (Logical Exclusive OR)

Efectúa una operación XOR entre cada uno de los bits de los dos operandos.

B8DAC70704	mov eax, 0407C7DA	
35C30E6F00	xor eax, 006F0EC3	EAX=0468C919

0407C7DA = 0000 0100 0000 0111 1100 0111 1101 1010

006F0EC3 = 0000 0000 0110 1111 0000 1110 1100 0011

xor 0000 0100 0110 1000 1100 1001 0001 1001 = 0468C919

not (One's Complement Negation)

Efectúa una operación NOT con cada uno de los bits del único operando.

B8DAC70704	mov eax, 0407C7DA	
F7D0	not eax	EAX= FBF83825

0407C7DA = 0000 0100 0000 0111 1100 0111 1101 1010

not 1111 1011 1111 1000 0011 1000 0010 0101 = FBF83825

2.3.7. Instrucciones de comprobación y verificación

cmp (Compare Two Operands)

La comparación entre dos valores es en realidad una resta entre ambos. Según cual sea el resultado, podemos saber si los valores son iguales y en caso contrario, cuál de ellos es el mayor. Así, se podría utilizar la instrucción sub ecx, ebx para comparar el resultado de estos dos registros, sin embargo el hacerlo así tiene el problema de que el resultado de la resta se colocaría en el registro ECX, cuyo valor anterior desaparecería. Para evitar este problema el programador

dispone de la instrucción cmp.

Esta instrucción resta el segundo operando del primero. El resultado no se guarda en ningún sitio, pero según cual sea este resultado, pueden modificarse los valores de los flags CF, OF, SF, ZF, AF y PF. Es en base al estado de estos flags, que se efectúa o no el salto condicional que suele acompañar a esta instrucción. Veremos esta instrucción en los ejemplos de saltos condicionales.

cmpxchg (Compare and Exchange)

Esta instrucción compara el valor de AL, AX o EAX (según el tamaño de los operandos) con el primer operando. Si son iguales, se pone a uno el flag de cero y el segundo operando se copia en el primero; en caso contrario, se pone a cero el flag de cero y el segundo operando se copia en AL, AX o EAX. Ejemplo:

```
B81CA23456      mov eax, 5634A21C

BB1CA23456      mov ebx, 5634A21C

B9CDAB3412      mov ecx, 1234ABCD

0FB1CB          cmpxchg ebx, ecx      EBX=1234ABCD    ZF=1

0FB1CB          cmpxchg ebx, ecx      EAX=1234ABCD    ZF=0
```

cmpxchg8b (Compare and Exchange 8 Bytes)

Esta instrucción es parecida a la anterior, pero funciona con parejas de registros (64bits). Se comparan EDX:EAX con el único operando, que debe ser una zona de memoria de 8 bytes de longitud. Si son iguales, se copia el valor de los registros ECX:EBX en la zona de memoria indicada por el operando; en caso contrario, el contenido de dicha zona se guarda en EDX:EAX.

test (Logical Compare)

El principio de esta instrucción es, en cierto modo, el mismo de cmp, es decir, una operación entre dos valores que no se guarda, sino que puede modificar el estado de algunos flags (en este caso, SF, ZF y PF) que determinan si debe efectuarse el salto que también suele acompañar a esta instrucción. La diferencia está en que en este caso, en vez de tratarse de una resta, se trata de una operación AND.

Esta instrucción se utiliza para averiguar si determinados bits de uno de los operandos están a 1 o 0, empleando una máscara como segundo operando. También se utiliza para saber si un valor es cero, comprobándolo consigo mismo. Es el caso de este código que resulta tan familiar:

```
E89DFFFFFF      call 0040143C

85C0            test eax, eax

7505            jne 004014A8
```

Si EAX es igual a cero, el resultado será cero; pero si es distinto de cero, al hacer un

AND de un bit de valor uno consigo mismo, el valor será uno y, en este caso concreto, se produciría el salto. Nos hemos adelantado al poner en el ejemplo la instrucción jne, pero enseguida veremos su significado.

bt (Bit Test)

Esta instrucción comprueba el valor del bit indicado por el segundo operando en el primer operando y lo copia en el flag de acarreo (CF).

btc (Bit Test and Compliment)

Esta instrucció es similar a la anterior, pero se diferencia en que después de copiar en el flag de acarreo el valor del bit comprobado, pone en este bit el complemento de su valor, es decir, el valor inverso. Veamos en un ejemplo la comparación con la instrucción anterior:

```
B8CD34AB12      mov eax, 12AB34CD

0FBAE010        bt eax, 10
```

```
        bit 31                  bit 16 (10h)            bit 0
        |                       |                       |
EAX = 0001 0010 1010 1011 0011 0100 1100 1101
```

Después de efectuar esta instrucción, el registro EAX queda como estaba y el flag de acarreo queda con el valor del bit 16 (CF=1).

```
0FBAF810        btc eax, 10
```

Después de efectuar esta otra instrucción, el bit 16 del registro EAX invierte su valor, pasando a valer 0 (EAX=12AA34CD) y el flag de acarreo queda con el valor anterior del bit 16 (CF=1).

btr (Bit Test and Reset)

Una nueva variante. Esta vez guarda en el flag de acarreo el valor del bit comprobado, pero pone este bit a cero.

```
B8CD34AB12      mov eax, 12AB34CD

0FBAF010        btr eax, 10
```

El resultado de efectuar esta instrucción es el mismo que el del anterior ejemplo (EAX=12AA34CD y CF=1).

bts (Bit Test and Set)

Igual a la anterior, pero pone el bit comprobado a uno.

```
B8CD34AB12      mov eax, 12AB34CD
```

0FBAE810 bts eax, 10

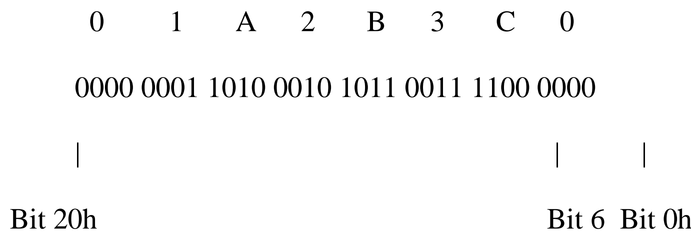
En este caso, EAX no varía y el flag de acarreo queda puesto a uno.

bsf (Bit Scan Forward)

Esta instrucción busca en el segundo operando, que puede ser un registro o una posición de memoria, el bit menos significativo cuyo valor sea igual a uno y coloca en el primer operando, que debe ser un registro, la posición que ocupa este bit empezando por el cero, desde la derecha. Con un ejemplo se verá mejor:

BBC0B3A201 mov ebx, 01A2B3C0

0FBCC3 bsf eax, ebx EAX=6

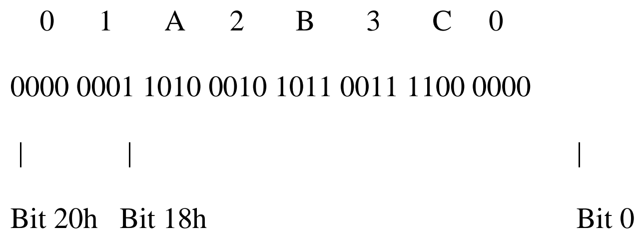


bsr (Bit Scan Reverse)

Esta instrucción es similar a la anterior, pero buscando el bit más significativo puesto a uno. Veamos un ejemplo:

BBC0B3A201 mov ebx, 01A2B3C0

0FBDC3 bsr eax, ebx EAX=18



2.3.8. Instrucciones de salto

El listado de un programa consiste en una sucesión de instrucciones. Sin embargo a la hora de ejecutarlo, la ejecución del mismo no sigue el orden del listado, sino que, de acuerdo con distintas circunstancias y mediante las instrucciones de salto, se interrumpe la ejecución

lineal del programa para continuar dicha ejecución en otro lugar del código.

Las instrucciones de salto son básicamente de dos tipos: de salto condicional y de salto incondicional. En el primer tipo, la instrucción de salto suele ponerse después de una comparación y el programa decide si se efectúa o no el salto, según el estado de los flags (excepto en un caso, en el que el salto se efectúa si el valor del registro ECX o CX es cero). En el segundo, el salto se efectúa siempre.

Tipos de salto

Según la distancia a la que se efectúe el salto, estos se dividen en tres tipos: corto, cercano y lejano. También se dividen en absolutos o relativos, según como se exprese en la instrucción la dirección de destino del salto. Como veremos, los saltos cortos y cercanos son relativos, y los largos absolutos.

En el salto corto, la dirección de destino se expresa mediante un byte, que va a continuación del código de la instrucción. Este byte contiene un número con signo que, sumado a la dirección de la instrucción siguiente a la del salto, nos da la dirección de destino del mismo. Como este número es con signo, podemos deducir que un salto corto sólo puede efectuarse a una distancia hacia adelante de 127 bytes y hacia atrás de 128. Veamos dos ejemplos de salto corto:

```
:004011E5 83FB05          cmp ebx, 00000005

:004011E8 7505            jne 004011EF

:004011EA C645002D        mov [ebp+00], 2D

...

:004011EF 83FB09          cmp ebx, 00000009

:004011F2 72E2            jb 004011D6

:004011F4 58              pop eax
```

En el primer salto, la dirección de destino es la suma de la dirección de la instrucción siguiente y el desplazamiento: 4011EA+5=4011EF. En el segundo caso, E2 es un número negativo, por lo que la dirección de destino es la de la instrucción siguiente menos la equivalencia en positivo de E2 (1E): 4011F4-1E=4011D6. Estos cálculos, por supuesto, no los hace el programador, que simplemente dirige el salto a una etiqueta para que luego el compilador coloque los códigos correspondientes.

El salto cercano es básicamente lo mismo que el salto corto. La diferencia está en que la distancia a que se efectúa es mayor, y no se puede expresar en un byte, por lo que se dispone de cuatro bytes (en programas de 16 bits) que permiten saltos de 32767 bytes hacia adelante y 32768 hacia atrás o de ocho bytes (en programas de 32 bits) que permiten vertiginosos saltos de 2147483647 bytes hacia adelante y 21474836478 bytes hacia atrás.

:0040194F 0F8E96000000 jle 004019EB

:00401955 8D4C2404 lea ecx, dword ptr [esp+04]

...

:004019CB 0F8566FFFFFF jne 00401937

:004019D1 8D4C240C lea ecx, dword ptr [esp+0C]

En la primera instrucción, la dirección de destino es: 401955+96= 4019EB. En la segunda la dirección es: 4019D1-9A= 401937. Los bytes que indican el desplazamiento están escritos al revés y que 9A es la equivalencia en positivo de FFFFFF66.

Los saltos largos se utilizan cuando la instrucción de destino está en un segmento distinto al de la instrucción de salto.

:0003.0C28 2EFFA72D0C jmp word ptr cs:[bx+0C2D]

Saltos Condicionales

Las instrucciones de salto condicional sólo admiten los formatos de salto corto y salto cercano, por lo que su código está formado por el código de la instrucción más un byte (cb), un word (cw) o un doubleword (cd) que determinan el desplazamiento del salto.

A continuación se podrá ver una relación de las instrucciones de salto condicional, desglosadas según el tipo de salto, en la que pueden apreciarse las equivalencias entre instrucciones de nombre distinto pero idéntica función.

Estas instrucciones equivalentes tienen el mismo código y se desensamblan con un nombre u otro, dependiendo del desensamblador.

Salto corto:

77 cb	JA rel8	Si es superior	(CF=0 y ZF=0)
	JNBE rel8	Si no es inferior o igual	(CF=0 y ZF=0)
73 cb	JAE rel8	Si es superior o igual	(CF=0)
	JNB rel8	Si no es inferior	(CF=0)
	JNC rel8	Si no hay acarreo	(CF=0)
76 cb	JNA rel8	Si no es superior	(CF=1 o ZF=1)
	JBE rel8	Si es inferior o igual	(CF=1 o ZF=1)
72 cb	JNAE rel8	Si no es superior o igual	(CF=1)

	JB rel8	Si es inferior	(CF=1)
	JC rel8	Si hay acarreo	(CF=1)
7F cb	JG rel8	Si es mayor	(ZF=0 y SF=OF)
	JNLE rel8	Si no es menor o igual	(ZF=0 y SF=OF)
7D cb	JGE rel8	Si es mayor o igual	(SF=OF)
	JNL rel8	Si no es menor	(SF=OF)
7E cb	JNG rel8	Si no es mayor	(ZF=1 o SF<>OF)
	JLE rel8	Si es menor o igual	(ZF=1 o SF<>OF)
7C cb	JNGE rel8	Si no es mayor o igual	(SF<>OF)
	JL rel8	Si es menor	(SF<>OF)
74 cb	JE rel8	Si es igual	(ZF=1)
	JZ rel8	Si es cero	(ZF=1)
75 cb	JNE rel8	Si no es igual	(ZF=0)
	JNZ rel8	Si no es cero	(ZF=0)
70 cb	JO rel8	Si hay desbordamiento	(OF=1)
71 cb	JNO rel8	Si no hay desbordamiento	(OF=0)
7A cb	JP rel8	Si hay paridad	(PF=1)
	JPE rel8	Si es paridad par	(PF=1)

7B cb	JNP rel8	Si no hay paridad	(PF=0)
	JPO rel8	Si es paridad impar	(PF=0)
78 cb	JS rel8	Si es signo negativo	(SF=1)
79 cb	JNS rel8	Si no es signo negativo	(SF=0)
E3 cb	JCXZ rel8	Si CX=0	
	JECXZ rel8	Si ECX=0	

Salto cercano:

0F 87 cw/cd	JA rel16/32	Si es superior	(CF=0 y ZF=0)
	JNBE rel16/32	Si no es inferior o igual	(CF=0 y ZF=0)
0F 83 cw/cd	JAE rel16/32	Si es superior o igual	(CF=0)
	JNB rel16/32	Si no es inferior	(CF=0)
	JNC rel16/32	Si no hay acarreo	(CF=0)
0F 86 cw/cd	JNA rel16/32	Si no es superior	(CF=1 o ZF=1)
	JBE rel16/32	Si es inferior o igual	(CF=1 o ZF=1)
0F 82 cw/cd	JNAE rel16/32	Si no es superior o igual	(CF=1)
	JB rel16/32	Si es inferior	(CF=1)
	JC rel16/32	Si hay acarreo	(CF=1)
0F 8F cw/cd	JG rel16/32	Si es mayor	(ZF=0 y SF=OF)
	JNLE rel16/32	Si no es menor o igual	(ZF=0 y SF=OF)

0F 8D cw/cd	JGE rel16/32	Si es mayor o igual	(SF=OF)
	JNL rel16/32	Si no es menor	(SF=OF)
0F 8E cw/cd	JNG rel16/32	Si no es mayor	(ZF=1 o SF<>OF)
	JLE rel16/32	Si es menor o igual	(ZF=1 o SF<>OF)
0F 8C cw/cd	JNGE rel16/32	Si no es mayor o igual	(SF<>OF)
	JL rel16/32	Si es menor	(SF<>OF)
0F 84 cw/cd	JE rel16/32	Si es igual	(ZF=1)
	JZ rel16/32	Si es cero	(ZF=1)
0F 85 cw/cd	JNE rel16/32	Si no es igual	(ZF=0)
	JNZ rel16/32	Si no es cero	(ZF=0)
0F 80 cw/cd	JO rel16/32	Si hay desbordamiento	(OF=1)
0F 81 cw/cd	JNO rel16/32	Si no hay desbordamiento	(OF=0)
0F 8A cw/cd	JP rel16/32	Si hay paridad	(PF=1)
	JPE rel16/32	Si es paridad par	(PF=1)
0F 8B cw/cd	JNP rel16/32	Si no hay paridad	(PF=0)
	JPO rel16/32	Si es paridad impar	(PF=0)
0F 88 cw/cd	JS rel16/32	Si es signo negativo	(SF=1)
0F 89 cw/cd	JNS rel16/32	Si no es signo negativo	(SF=0)

NOTA: La diferencia entre mayor/menor y superior/inferior es la de que mayor/menor se refiere al resultado de la comparación entre números con signo, y superior/inferior al de la comparación entre números sin signo.

Una vez relacionados todos los saltos habidos y por haber, pensemos que en general los únicos saltos que nos interesan son jz/je o jnz/jne, jg o jng y jle o jnle. La primera pareja de saltos se efectúa para aceptar que el programa está registrado o para admitir como correcto el serial number que hemos introducido. Las otras dos las podemos encontrar en comprobaciones de los días que restan del período de prueba de un programa.

Veremos a continuación tres ejemplos de comparación donde se podrá apreciar el estado de los flags afectados por el resultado y los saltos que se ejecutan según este estado. Como se puede ver, vamos a limitarnos a los saltos más corrientes.

```
:00401144 B81F126700          mov eax, 0067121F

:00401149 3D998C1900          cmp eax, 00198C99    SF=0    CF=0    OF=0    ZF=0
```

El estado de los flags hace que puedan efectuarse estos cinco saltos:

```
:0040114E 7700                ja 00401150

:00401150 7300                jae 00401152

:00401152 7F00                jg 00401154

:00401154 7D00                jge 00401156

:00401156 7500                jne 00401158
```

Otra comparación:

```
:00401158 3DE1019300          cmp eax, 009301E1    SF=1    CF=1    OF=0    ZF=0
```

Y los cinco saltos posibles:

```
:0040115D 7200                jb 0040115F

:0040115F 7600                jbe 00401161

:00401161 7C00                jl 00401163

:00401163 7E00                jle 00401165

:00401165 7500                jne 00401167
```

La última comparación:

```
:00401167 3D1F126700          cmp eax, 0067121F    SF=0    CF=0    OF=0 ZF=1
```

Y los últimos cinco saltos:

```
:0040116C 7400          je 0040116E
:0040116E 7300          jae 00401170
:00401170 7600          jbe 00401172
:00401172 7D00          jge 00401174
:00401174 7E00          jle 00401176
```

También se puede invertir los saltos condicionales para registrar un programa. Debemos obligar al programa a que haga lo que queremos que haga, por lo que si ha de saltar para quedar registrado, debemos substituir el salto condicional por uno incondicional y si no debe saltar, substituir el salto condicional por nops u otra solución más elegante. Por ejemplo:

```
:00401135 3BCB          cmp ecx, ebx
:00401137 7547          jne 00401180      No queremos que salte
:00401139 84C0          test al, al
:0040113B 7443          je 00401180       Queremos que salte
...
:00401149 3BCB          cmp ecx, ebx
:0040114B 0F85E3000000  jne 00401234      No queremos que salte
:00401151 84C0          test al, al
:00401153 0F84DB000000  je 00401234       Queremos que salte
:00401159 40            inc eax
```

Se puede substituir por:

```
:00401135 3BCB          cmp ecx, ebx
:00401137 EB00          jmp 00401139      Salta a la instrucción siguiente
:00401139 84C0          test al, al
:0040113B EB43          jmp 00401180      Salta siempre
...
:00401149 3BCB          cmp ecx, ebx
:0040114B EB00          jmp 0040114D      Salta a la instrucción siguiente
:0040114D EB00          jmp 0040114F      Salta a la instrucción siguiente
```

:0040114F EB00	jmp 00401151	Salta a la instrucción siguiente
:00401151 84C0	test al, al	
:00401153 90	nop	
:00401154 E9DB000000	jmp 00401234	Salta siempre
:00401159 40	inc eax	

Como se puede ver el resultado es el mismo. Especial atención requieren los tres saltos seguidos para evitar poner seis nops (o tres nops "dobles" 6690). Esta solución está dedicada a todos aquellos que no les gusta poner nops, aunque parece peor el remedio que la enfermedad.

Salto incondicional

Es un salto que no está sujeto a ninguna condición, es decir, que se efectúa siempre. Hay una sola instrucción: jmp. Esta instrucción admite los tres tipos de salto: corto, cercano y lejano. Los siguientes ejemplos ilustran estos saltos:

:004011DA	EB30	jmp 0040120C
:00402B35	E9F8000000	jmp 00402C32
:0001.02B4	E94D01	jmp 0404
:0003.0C28 2	EFFA72D0C	jmp word ptr cs:[bx+0C2D]

2.3.9. Instrucciones de subrutinas

Las subrutinas son grupos de instrucciones con una finalidad concreta, que pueden ser llamadas desde uno o varios puntos del programa, al que regresan después de la ejecución de dicha subrutina. La utilización de subrutinas permite una mejor estructuración del programa al dividirlo en bloques independientes y evita la repetición del código de procesos que se ejecutan varias veces a lo largo de la ejecución del programa.

Para la ejecución de una subrutina, son necesarios algunos parámetros que pueden estar en registros, en posiciones de la memoria o en la pila. En este último caso (generalmente en programas escritos en lenguajes de alto nivel), la llamada a la subrutina está precedida de algunas instrucciones push que tienen la misión de colocar esos parámetros en la pila. Según el lenguaje empleado en la programación, a veces es necesario ajustar la pila, modificando directamente el valor del registro ESP o añadiendo un parámetro a la instrucción ret, consistente en el número que hay que sumar al registro ESP para que la dirección de retorno sea la correcta

Los resultados obtenidos después de la ejecución de la subrutina suelen guardarse en los registros (especialmente en EAX) o bien en posiciones de memoria indicadas por éstos. Por esta razón, es conveniente al encontrarse con una subrutina "sospechosa" mirar los datos de entrada, viendo los valores colocados en la pila o las direcciones a las que apuntan estos valores y los datos de salida mirando el contenido de los registros o de las direcciones a que apuntan.

call (Call Procedure)

Es la instrucción que efectúa el salto al punto de inicio de la subrutina. Además de esto, coloca en la pila la dirección de la instrucción siguiente, que será el punto de regreso después de ejecutarse la subrutina.

ret (Return from Procedure)

Complementa a la anterior y tiene la misión de regresar a la instrucción siguiente a la de llamada a la subrutina. Para ello, efectúa un salto a la dirección contenida en la pila, quedando ésta como estaba antes del call. Como hemos visto, puede ir acompañada de un parámetro para ajuste de la pila.

Veamos un ejemplo de cómo va variando el registro ESP según se ejecutan las instrucciones descritas:

```
...                                              ESP=63FB1C
:00401144 53                push ebx             ESP=63FB18   (-4)
:00401145 6651              push cx              ESP=63FB16   (-2)
:00401147 6808304000        push 00403008        ESP=63FB12   (-4)
:0040114C E8B3000000        call 00401204        ESP=63FB0E   (-4)

    :00401204 40            inc eax
    ...                     ...                  ESP=63FB0E
    :00401296 C20A00        ret 000A             ESP=63FB1C   (+A+4)

:00401151     Continúa la ejecución del programa...
```

o bien:

```
...                                              ESP=63FB1C
:00401144 53                push ebx             ESP=63FB18   (-4)
:00401145 6651              push cx              ESP=63FB16   (-2)
:00401147 6808304000        push 00403008        ESP=63FB12   (-4)
:0040114C E8B3000000        call 00401204        ESP=63FB0E   (-4)

    :00401204 40            inc eax
    ...                     ...                  ESP=63FB0E
```

:00401296 C3 ret ESP=63FB12 (+4)

:00401151 83C40A add esp, 0000000A ESP=63FB1C (+A)

:00401154 Continúa la ejecución del programa...

Como vemos, el resultado es el mismo; lo contrario causaría un crash del programa.

Hay algo que todos hemos visto muchas veces con emoción:

```
E89DFFFFFF      call 0040143C
85C0            test eax, eax
7505            jne 004014A8
```

Es el clásico call con la comprobación al regreso que decide si somos chicos buenos o chicos malos. Una forma de resolver la situación es la de substituir las primeras instrucciones del call por:

```
33C0     xor eax, eax
C3       ret
```

o bien:

```
33C0     xor eax, eax
40       inc eax
C3       ret
```

Con esto conseguimos que EAX, al regreso del call, tenga el valor cero o uno, según nos interese. Pero esto no siempre puede hacerse. Hay que asegurarse de que dentro del call no haya algún reajuste de la pila que nos lleve a la catástrofe. Para ello, debemos verificar el valor de ESP antes y después de la ejecución del call. Si son los mismos no hay problema, pero si son distintos podemos tratar de ajustar el valor de la pila y, según el resultado, buscar otro sistema. Hay que tener en cuenta que si hay alguna instrucción push antes del call, ésta también modifica el registro ESP y es posible que eliminándola se corrija el desajuste. Lo mismo vale para el clásico call que se quiere eliminar bien porque hace aparecer una nag, o bien por cualquier otra causa.

2.3.10. Instrucciones de bucle

loop (Loop According to ECX Counter)

Esta instrucción efectúa un bucle un número de veces determinado por el registro ECX. Equivale a las instrucciones dec ecx / jnz. Veamos un ejemplo doble:

:00401150 33C0	xor eax, eax
:00401152 B90A000000	mov ecx, 0000000A
:00401157 40	inc eax
:00401158 E2FD	loop 00401157

Se ejecutaría diez veces el bucle, por lo que el valor final de EAX seria de 0000000A. También se ejecutaría diez veces el bucle equivalente:

:00401150 33C0	xor eax, eax
:00401152 B90A000000	mov ecx, 0000000A
:00401157 40	inc eax
:00401158 49	dec ecx
:00401159 75FC	jne 00401157

Existen variantes de esta instrucción que, además de depender del registro ECX, están condicionadas al estado del flag de cero (ZF):

loope / loopz Se efectúa el bucle si ECX es distinto de cero y ZF=1

loopne / loopnz Se efectúa el bucle si ECX es distinto de cero y ZF=0

2.3.11. Instrucciones de cadenas

En este apartado se han agrupado instrucciones que podían haber estado igualmente en otros apartados, como de comparación, de entrada/salida, etc. pero con el objetivo de mantener cierta coherencia se incluyen todas las instrucciones de cadenas juntas.

rep (Repeat String Operation Prefix)

Esta instrucción se emplea en combinación con otras instrucciones de cadenas: MOVS, LODS, STOS, INS y OUTS. Tiene la finalidad de repetir la instrucción asociada una cantidad de veces determinada por los registros CX o ECX, según el programa trabaje con 16 ó 32 bits (En realidad el valor del registro se va decrementando en cada repetición, hasta llegar a cero).

Esta instrucción o, mejor dicho, las instrucciones asociadas, están relacionadas con los registros ESI y EDI, por lo que en cada repetición dichos registros se van incrementando o decrementando para apuntar siempre a la posición de memoria correspondiente.

repe (Repeat While Equal) / **repz** (Repeat While Zero)

Estas instrucciones se combinan con CMPS y SCAS. Son similares a rep, pero en este caso la repetición está condicionada a que el contenido del registro ECX sea distinto de cero y a

que el flag de cero (ZF) sea igual a uno.

repne (Repeat While Not Equal) / **repnz** (Repeat While Not Zero)

Instrucciones iguales a las anteriores, con la diferencia de qué, para que se efectúe la repetición, el flag de cero debe ser igual a cero, en vez de igual a uno.

movs / movsb /movsw / movsd (Move Data from String to String)

Estas instrucciones tienen la misión de copiar el contenido de un espacio de memoria a partir de la dirección DS:ESI a otro espacio direccionado por ES:EDI. La longitud del espacio de memoria a copiar está indicada por el sufijo de la instrucción (byte / word / doubleword) y por el contenido del registro ECX.

lods / lodsb / lodsw lodsd (Load String)

Estas instrucciones se encargan de copiar un byte, word o doubleword, de la dirección indicada por DS:ESI al registro AL, AX o EAX, según el tamaño de la información a copiar indicado por el sufijo de la instrucción. Veremos un ejemplo al tratar de la instrucción scas.

stos / stosb / stosw / stosd (Store String)

Estas instrucciones son, en cierto modo, las inversas de las anteriores, ya que colocan el contenido de AL, AX o EAX, según el tamaño de la información a copiar indicado por el sufijo de la instrucción, en la dirección indicada por ES:EDI.

cmps / cmpsb / cmpsw / cmpsd (Compare String Operands)

Estas instrucciones comparan un byte, word o doubleword de la dirección indicada por DS:ESI con el valor del mismo tamaño situado en la dirección ES:EDI. Veamos un ejemplo:

```
F3          repz

A6          cmpsb
```

En la dirección DS:ESI estaba el serial introducido, en la ES:EDI, el serial correcto y en ECX la longitud del serial introducido, que naturalmente era preciso que fuera igual que la del correcto.

scas / scasb /scasw / scasd (Scan String)

Estas instrucciones comparan un byte, word o doubleword de la dirección indicada por ES:EDI con el valor del registro AL, AX o EAX, según el sufijo de la instrucción. Veamos un ejemplo:

```
AC          lodsb

F2          repnz

AE          scasb
```

Primero colocaríamos en AL el contenido de la posición de memoria indicada por DS:ESI. A continuación compararíamos el contenido de ese registro con el byte situado en la dirección indicada por ES:EDI, repitiéndose la operación con los bytes siguientes hasta encontrar un byte del mismo valor o hasta que el contenido del registro ECX fuera cero.

ins / insb / insw / insd (Input from Port to String)

Estas instrucciones se emplean para colocar en la dirección ES:EDI un byte, word o doubleword según el sufijo de la instrucción, obtenido de un puerto de entrada / salida cuyo número es el indicado por el valor del registro DX.

outs / outsb / outsw / outsd (Output String to Port)

Estas instrucciones funcionan al revés que las anteriores, es decir que colocan un byte, word o doubleword, según el sufijo de la instrucción, situado en la dirección DS:ESI, en un puerto de entrada / salida cuyo número es el indicado por el valor del registro DX.

2.3.12. Instrucciones de Entrada / Salida

in (Input from Port)

Copia el valor procedente del puerto de entrada / salida indicado por el segundo operando, en el primero. El segundo operando puede ser el registro DX o un valor inmediato del tamaño de un byte (sólo para valores comprendidos entre 0 y 255). El primer operando serán los registros AL, AX o EAX, según el tamaño del valor a transferir.

out (Output to Port)

Copia el valor contenido en el segundo operando, en el puerto de entrada / salida indicado por el primer operando. El primer operando puede ser el registro DX o un valor inmediato del tamaño de un byte (sólo para valores comprendidos entre 0 y 255). El segundo operando serán los registros AL, AX o EAX, según el tamaño del valor a transferir.

Veamos una aplicación de estas dos instrucciones:

```
E461        in al, 61

0C03        or al, 03

E661        out 61, al
```

Inicia el sonido del altavoz interno.

```
33C0        xor eax, eax

E642        out 42, al

40          inc eax

3DFF00000   cmp eax, 000000FF

76F6        jbe 0040114D
```

La ejecución de este código permite oír sólo un chasquido pero, si se ejecuta paso a paso, se puede oír toda una escala de tonos.

```
E461        in al, 61
24FC        and al, FC
E661        out 61, al
```

Finaliza el sonido del altavoz interno. Este código es importante, porque si no estuviera deberíamos apagar el ordenador para dejar de oír el ruido.

2.3.13. Instrucciones de rotación y desplazamiento

Estas instrucciones toman los bits de un valor indicado en el primer operando y los desplazan un número de posiciones determinado por el segundo operando, que puede ser un valor determinado, o un valor contenido en el registro CL. Estos desplazamientos pueden ser aritméticos o lógicos. Los desplazamientos aritméticos se efectúan con números sin signo y los espacios vacíos que se crean se rellenan con ceros. Los lógicos se efectúan con números con signo y el valor con que se rellenan los espacios que se crean es el adecuado para que el resultado sea el correcto.

sal (Shift Arithmetic Left) / **shl** (Shift Left)

Estas dos instrucciones son de desplazamiento aritmético a la izquierda y desplazamiento lógico a la izquierda respectivamente. Colocan el bit menos significativo de los que desaparecen, en el flag de acarreo y rellenan los espacios que se crean a la derecha con ceros. Son equivalentes e incluso tienen el mismo código de operación. En la práctica, lo que hacen es multiplicar el primer operando por dos, tantas veces como indique el segundo.

```
B84B28D17C          mov eax, 7CD1284B
D1E0                shl eax,   1            EAX=F9A25096      CF=0
```

```
          7    C    D    1    2    8    4    B
<- 1   0111 1100 1101 0001 0010 1000 0100 1011
       1111 1001 1010 0010 0101 0000 1001 0110
          F    9    A    2    5    0    9    6
```

7CD1284Bx(2^1)=F9A25096

```
B83B52E48A      mov eax, 8AE4523B
C1E008          shl eax,    08          EAX=E4523B00      CF=0
```

8 A E 4 5 2 3 B

<- 8 1000 1010 1110 0100 0101 0010 0011 1011

1110 0100 0101 0010 0011 1011 0000 0000

E 4 5 2 3 B 0 0

8AE4523Bx(2^8)=E4523B00

sar (Shift Arithmetic Right)

Instrucción de desplazamiento aritmético a la derecha. El bit más significativo de los que desaparecen se coloca en el flag de acarreo y las posiciones que se crean a la izquierda se rellenan con el bit más significativo. En la práctica significa dividir con signo el primer operando por dos, tantas veces como indique el segundo, redondeando por defecto.

B83A52E48A mov eax, 8AE4523A

C1F802 sar eax, 02 EAX=E2B9148E CF=1 SF=1

8 A E 4 5 2 3 A

2 -> 1000 1010 1110 0100 0101 0010 0011 1010

1110 0010 1011 1001 0001 0100 1000 1110

E 2 B 9 1 4 8 E

8AE4523A:(2^2)=E2B9148E -1964748230d:4=-491187058d

shr (Shift Right)

Instrucción de desplazamiento lógico a la derecha. El bit más significativo de los que desaparecen, se coloca en el flag de acarreo y las posiciones que se crean a la izquierda se rellenan con ceros. En la práctica significa dividir sin signo el primer operando por dos, tantas veces como indique el segundo, redondeando por defecto.

B83A52E48A mov eax, 8AE4523A

C1E802 shr eax, 02 EAX=22B9148E CF=1

8 A E 4 5 2 3 A

2 -> 1000 1010 1110 0100 0101 0010 0011 1010

0010 0010 1011 1001 0001 0100 1000 1110

2 2 B 9 1 4 8 E

8AE4523A:(2^2)=22B9148E   2330219066:4=582554766

shld (Double Precision Shift Left)

Instrucción de desplazamiento lógico a la izquierda, parecida a shl pero con la diferencia de que los bits añadidos por la derecha no son ceros, sino que se toman del segundo operando, empezando por la izquierda, es decir por el bit más significativo. Vamos a ver un ejemplo:

B8CDAB3412   mov eax, 1234ABCD

BB78563512   mov ebx, 12355678

0FA4D808   shld eax, ebx, 08   EAX=34ABCD12

1 2 3 4 A B C D 1 2 3 5

<- 8 0001 0010 0011 0100 1010 1011 1100 1101 0001 0010 0011 0101 ...

0011 0100 1010 1011 1100 1101 0001 0010

3 4 A B C D 1 2

shrd (Double Precision Shift Right)

Instrucción de desplazamiento lógico a la derecha, parecida a shr pero con la diferencia de que los bits añadidos por la izquierda no son ceros, sino que se toman del segundo operando, empezando por la derecha, es decir por el bit menos significativo. Veamos un ejemplo:

B8CDAB3412   mov eax, 1234ABCD

BB78563412   mov ebx, 12345678

0FACD808   shrd eax, ebx, 08   EAX=781234AB

... 7 8 1 2 3 4 A B C D

8 -> 0111 1000 0001 0010 0011 0100 1010 1011 1100 1101

0111 1000 0001 0010 0011 0100 1010 1011

7 8 1 2 3 4 A B

rol (Rotate Bits Left)

Instrucción de rotación a la izquierda. Los bits que desaparecen por la izquierda, se van colocando por la derecha y en el flag de acarreo se coloca el valor del último bit que ha rotado.

B83A52E48A mov eax, 8AE4523A

C1C002 rol eax, 02 EAX=2B9148EA CF=0

```
            8    A    E    4    5    2    3    A
<- 2     1000 1010 1110 0100 0101 0010 0011 1010
          0010 1011 1001 0001 0100 1000 1110 1010
            2    B    9    1    4    8    E    A
```

ror (Rotate Bits Right)

Instrucción de rotación a la derecha. Los bits que desaparecen por la derecha, se van colocando por la izquierda y en el flag de acarreo se coloca el valor del último bit que ha rotado.

B83A52E48A mov eax, 8AE4523A

C1C802 ror eax, 02 EAX=A2B9148E CF=1

```
            8    A    E    4    5    2    3    A
2 ->     1000 1010 1110 0100 0101 0010 0011 1010
     1010 0010 1011 1001 0001 0100 1000 1110
         A    2    B    9    1    4    8    E
```

rcl (Rotate Bits Left with CF)

Instrucción de rotación a la izquierda pasando por el bit de acarreo. Los bits que desaparecen por la izquierda, se van colocando en el bit de acarreo y el valor anterior de éste, se coloca a la derecha.

Si CF=0:

B83A52E48A mov eax, 8AE4523A CF=0

C1D002 rcl eax, 02 EAX=2B9148E9 CF=0

```
            8    A    E    4    5    2    3    A
<- 2     1000 1010 1110 0100 0101 0010 0011 1010        CF=0
```

0010 1011 1001 0001 0100 1000 1110 1001

2 B 9 1 4 8 E 9

Si CF=1:

B83A52E48A	mov eax,	8AE4523A		CF=1
C1D002	rcl eax,	02	EAX=2B9148EB	CF=0

8 A E 4 5 2 3 A

<- 2 1000 1010 1110 0100 0101 0010 0011 1010 CF=0

0010 1011 1001 0001 0100 1000 1110 1011

2 B 9 1 4 8 E B

rcr (Rotate Bits Right with CF)

Instrucción de rotación a la derecha pasando por el bit de acarreo. Los bits que desaparecen por la derecha, se van colocando en el bit de acarreo y el valor anterior de éste, se coloca a la izquierda.

Si CF=0:

B83A52E48A	mov eax,	8AE4523A		CF=0
C1D802	rcr eax,	02	EAX=22B9148E	CF=1

8 A E 4 5 2 3 A

2 -> 1000 1010 1110 0100 0101 0010 0011 1010

0010 0010 1011 1001 0001 0100 1000 1110

2 2 B 9 1 4 8 E

Si CF=1:

B83A52E48A	mov eax,	8AE4523A		CF=1
C1D802	rcr eax,	02	EAX=62B9148E	CF=1

	8	A	E	4	5	2	3	A
2 ->	1000	1010	1110	0100	0101	0010	0011	1010
	0110	0010	1011	1001	0001	0100	1000	1110
	6	2	B	9	1	4	8	E

2.3.14. Instrucciones de conversión

Se utilizan para duplicar la longitud de un número con signo. Para esto, rellenan el espacio añadido con el valor del bit más significativo del número original.

cbw (Convert Byte to Word)

Esta instrucción convierte el byte contenido en el registro AL en un word contenido en AX. En el primero de los ejemplos siguientes, el espacio ampliado se rellena con unos porque el bit más significativo de 80h es uno. En el segundo ejemplo, se rellena con ceros porque el bit más significativo de 7832h es cero.

B880E3DA12	mov eax, 12DAE380	
6698	cbw	EAX=12DA**FF**80
B832782100	mov eax, 00217832	
6698	cbw	EAX=0021**00**32

cwde (Convert Word to Doubleword)

Esta instrucción tiene una función similar a cbw, pero entre AX y EAX.

B880E3DA12	mov eax, 12DAE380	
98	cwde	EAX=**FFFF**E380
B832782100	mov eax, 00217832	
98	cwde	EAX=**0000**7832

cwd (Convert Word to Doubleword)

Esta instrucción añade al word contenido en el registro AX, el word contenido en DX, formando el doubleword AX:DX. El registro DX se rellena con unos o ceros, igual que en los casos anteriores, como se puede ver en los ejemplos siguientes:

B880E3DA82	mov eax, 82DAE380		
BA11111111	mov edx, 11111111		
6699	cwd	EAX=82DAE380	EDX=1111**FFFF**

B832782100 mov eax, 00217832

BA11111111 mov edx, 11111111

6699 cwd EAX=00217832 EDX=1111**0000**

cdq (Convert Double to Quad)

Esta instrucción tiene una función similar a cwd, pero entre EAX y EDX, formando el quadword EAX:EDX.

B880E3DA82 mov eax, 82DAE380

BA11111111 mov edx, 11111111

99 cdq EAX=82DAE380 EDX=**FFFFFFFF**

B832782100 mov eax, 00217832

BA11111111 mov edx, 11111111

99 cdq EAX=00217832 EDX=**00000000**

2.3.15. Instrucciones de flags

clc (Clear Carry Flag)

Pone el flag de acarreo (CF) a cero.

stc (Set Carry Flag)

Pone el flag de acarreo (CF) a uno.

cmc (Complement Carry Flag)

Complementa el flag de acarreo (CF). Es decir, que invierte su valor.

cld (Clear Direction Flag)

Pone el flag de dirección (DF) a cero.

std (Set Direction Flag)

Pone el flag de dirección (DF) a uno.

cli (Clear Interrupt Flag)

Pone el flag de interrupción (IF) a cero.

sti (Set Interrupt Flag)

Pone el flag de interrupción (IF) a uno.

lahf (Load Status Flags into AH Register)

Guarda en AX los flags de signo, cero, auxiliar, paridad y acarreo. Los bits 1, 3 y 5 tienen valores fijos, por lo que el registro AX queda formado por: SF:ZF:0:AF:0:PF:1:CF. En el siguiente ejemplo, si suponemos que: SF=0, ZF=1, AF=0, PF=1 y CF=0:

```
9F          LAHF                EAX=xxxx46xx

                                SZ0A  0P1C

                                ||||  ||||

AH valdrá 46h que es igual a:   0100  0110
```

sahf (Store AH into Flags)

Esta instrucción efectúa la operación inversa a la anterior. Es decir, que pone los flags a uno o a cero, de acuerdo con el valor de AH.

2.3.16. Instrucciones de interrupción

int (Call to Interrupt Procedure)

Esta instrucción causa una interrupción de software determinada por su único operando y se desglosa en tres, según su código de operación:

```
CC          INT 3
```

La interrupción 3 hace que un debugger detenga la ejecución del programa. Es la instrucción que inserta el debugger en un programa cuando ponemos un breakpoint en un punto de la ejecución del mismo.

```
CD nn       INT nn
```

El número de la interrupción viene dado por un valor inmediato de un byte de longitud.

```
CE          INTO
```

Esta instrucción corresponde a la interrupción 4. Comprueba el valor del flag de desbordamiento (OF) y, si está activado, efectúa una llamada a dicha interrupción.

iretd / iret (Interrupt Return 32 / 16 bits)

Se produce el retorno de la rutina de interrupción y continúa la ejecución del programa.

2.3.17. Instrucciones del procesador

enter (Make Stack Frame for Procedure Parameters)

Esta instrucción crea un espacio en la pila requerido por algunos lenguajes de alto nivel. Está acompañada de dos parámetros, el primero indica la longitud del espacio reservado y el segundo el nivel de anidamiento léxico, o nivel de anidamiento de los procedimientos del programa. La longitud del espacio a reservar se resta de ESP, y el nuevo contenido de este registro se copia en EBP. Veamos un ejemplo para aclarar los efectos de esta instrucción:

```
...                                  ESP=63FB1C  EBP=63FB44

C8000404    enter 0400, 04           ESP=63F708  EBP=63FB18
```

La nueva dirección de la pila es el resultado de restarle a la anterior 404h, es decir, 4 bytes para guardar el contenido de EBP y 400h de espacio reservado. El nuevo contenido de EBP es la dirección en que está situado su anterior valor.

leave (High Level Procedure Exit)

Instrucción que complementa a la anterior. Restituye los valores de la pila anteriores a la última instrucción enter. Veamos la continuación del ejemplo anterior, suponiendo que posteriores instrucciones no hubiesen modificado los valores de EBP y ESP:

```
...                                  ESP=63F708  EBP=63FB18

C9          leave                    ESP=63FB1C  EBP=63FB44
```

bound (Check Array Index Against Bounds)

Esta instrucción tiene dos parámetros y se encarga de determinar si el primer operando (un registro), que actúa como índice de una array, tiene un valor comprendido entre los límites establecidos de los valores de dicha array.

```
:0040114C 622B          bound ebp, dword ptr [ebx]
```

En el ejemplo, EBX contiene el valor inferior establecido para la array (en este caso, 00000000) y EBX+4, el valor superior (en este caso, 000000FFh). Si el valor de EBP no estuviera comprendido entre ambos, se produciría una interrupción 5. En efecto, si le damos a EBP un valor superior a FFh el resultado es que nos aparece el siguiente aviso:

xxxx provocó una excepción 05H en el módulo xxxx.EXE de 0167:0040114c.

hlt (Halt)

Instrucción sin parámetros que detiene la ejecución del programa. Dicha ejecución se puede reanudar si ocurre una interrupción o mediante un reset del sistema.

wait (Wait)

Instrucción que suspende la ejecución del programa mientras espera que termine de efectuarse alguna operación de coma flotante.

Capítulo 3

INTRODUCCIÓN A LA INGENIERIA INVERSA

3.1 ¿Qué es la Ingeniería Inversa?

La ingeniería inversa consiste en el proceso de tomar un binario compilado e intentar recrear (o, simplemente, entender) el funcionamiento originario del programa. Un programador escribe inicialmente un programa, por lo general en un lenguaje de alto nivel como C ++ o Visual Basic. Debido a que el equipo no habla inherentemente estos lenguajes, el código que escribió el programador es ensamblado a un formato más específico, de manera que el equipo pueda entenderlo. A este código se le llama lenguaje de bajo nivel o simplemente lenguaje del equipo. El reto de la ingeniería inversa consiste en coger ese código poco amigable para el común de los mortales y averiguar lo que el programador tenía en mente a la hora de escribirlo.

3.2 aplicaciones de la ingeniería inversa

La ingeniería inversa se puede aplicar a muchas áreas de la informática, sin embargo vamos enumerar cinco categorías genéricas:

- Posibilidad de interactuar con el código heredado sin conocer el código fuente.
- De forma fraudulenta crackeando software.

- Estudiar virus y malware.
- Evaluar la calidad y la robustez del software.
- Añadir funcionalidad al software existente.

3.3 Conocimientos previos necesarios

Como se puede imaginar, es necesario poseer una gran cantidad de conocimiento para ser un ingeniero inverso eficaz. Afortunadamente, no es necesario tener muchos conocimientos para 'empezar' con la ingeniería inversa.

No obstante para sacarle el máximo provecho a este tutorial debemos estar familiarizados con unos conocimientos básicos acerca del flujo de un programa. Conocer las estructuras de control, como por ejemplo if…then, conocer lo que es un array, un bucle,… puede facilitarnos el trabajo.

Por otra parte, familiarizarse con el lenguaje ensamblador es muy recomendable. No es un requisito indispensable pero para entender realmente lo que estamos haciendo, ASM es de estudio obligado. A mayores, una gran cantidad de tiempo se dedica a aprender a usar herramientas. Estas herramientas son de gran valor para la ingeniería inversa, pero también requieren aprender atajos, defectos e idiosincrasias de cada herramienta. Finalmente, la ingeniería inversa requiere de una cantidad significativa de experimentación; jugando con diferentes empacadores / protectores / esquemas de cifrado, el aprendizaje acerca de los programas escritos originalmente en diferentes lenguajes de programación (incluso Delphi), descifrar trucos de ingeniería anti-inversa ... etc.

3.4 herramientas utilizadas en ingeniería inversa

En Ingeniería Inversa se pueden usar una gran cantidad de herramientas. Muchos están especializados en el tipo de protección de determinados binarios. Otros simplemente tratan de facilitar el trabajo de los ingenieros inversos. La mayoría de ellos se pueden englobar en las siguientes categorías.

3.4.1 DESENSAMBLADORES

Desensambladores intentan mostrar el código del lenguaje de la máquina en un formato más amigable a los ojos de los humanos. También extrapolan datos como llamadas a funciones, pasan variables y cadenas de texto. Esto hace que el aspecto del ejecutable sea un código más legible en lugar de un montón de números insertados aleatoriamente. Hay muchas tipos de desensambladores, algunos de ellos están especializados en binarios específicos (como los binarios escritos en Delphi). Ejemplos de desensambladores: IDA, Ollydbg.

3.4.2 DEPURADORES

Los depuradores (debuggers) son el pan de cada día para los ingenieros inversos. Primero analizan el código binario, para luego permitir el estudio del código, ejecutando las líneas de una en una para investigar los resultados. Esto es imprescindible para descubrir cómo funciona un programa. Por último, algunos depuradores permiten cambiar ciertas instrucciones en el código para luego correrlos otra vez con estos cambios. Ejemplos de depuradores son WinDbg y OllyDbg.

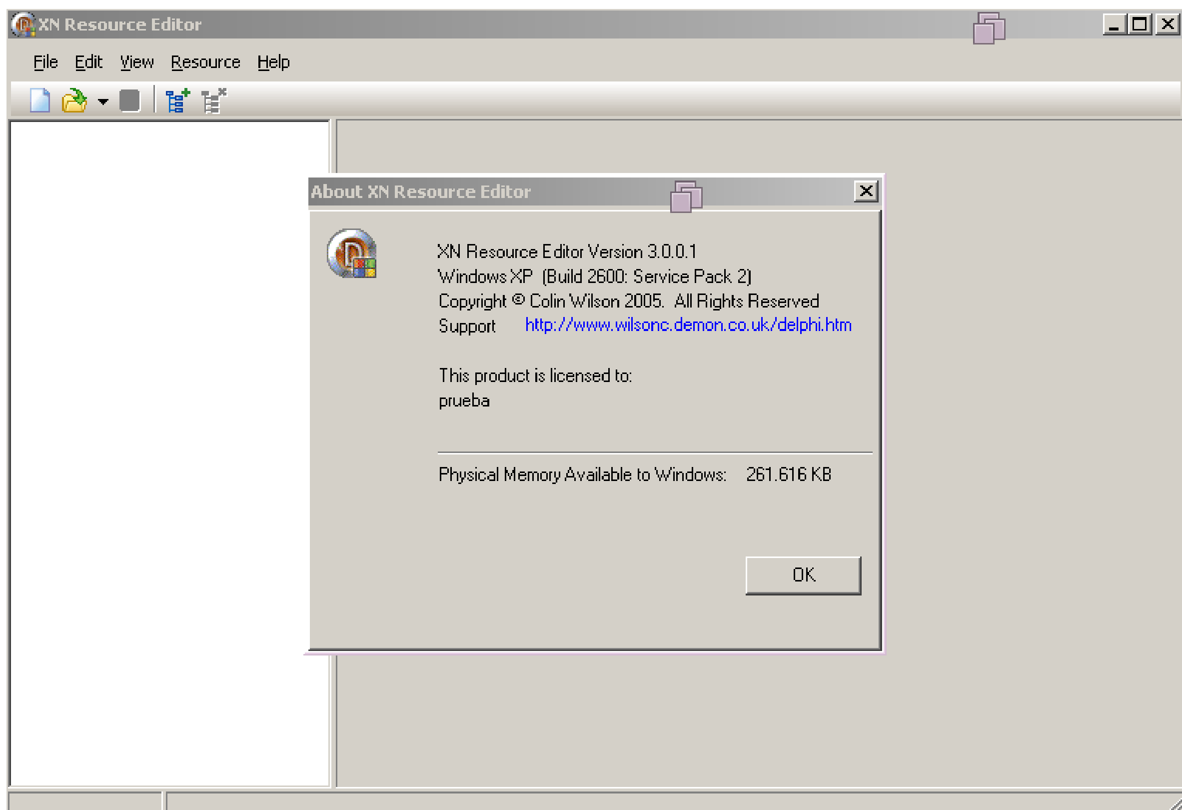
3.4.3 EDITORES HEXADECIMALES

Los Editores Hexadecimales permiten ver los bytes reales en un binario, y cambiarlos. También ayudan en la búsqueda de bytes específicos, guardando secciones de un binario en el disco, y mucho más. Hay muchos editores hexadecimales gratuitos en la red, no obstante haremos uso del editor hexadecimal que viene integrado en Ollydbg.

3.4.4 PE Y EDITORES DE RECURSOS

Cada binario diseñado para ejecutarse en una máquina Windows (y Linux en su caso) tiene una sección muy específica de los datos al inicio de la misma que indica al sistema operativo cómo configurar e inicializar el programa. Le dice al sistema operativo la cantidad de memoria requerida, qué DLL de código necesita para el programa, informa sobre los cuadros de diálogo etc. Esta sección se conoce como "Portable Executable", y todos los programas diseñados para funcionar en Windows deben tener uno. En el mundo de la ingeniería inversa, esta estructura de bytes se vuelve muy importante, ya que da al ingeniero la información necesaria acerca del binario. Con el tiempo, todo ingeniero inverso querrá (o necesitará) cambiar esta información, ya sea para hacer que el programa haga algo diferente a lo que hacía inicialmente, o para cambiar el programa a su estado original. Podemos encontrar PE y editores de recursos en:

(Http://www.ntcore.com/exsuite.php)
(Http://www.woodmann.com/collaborative/tools/index.php/LordPE).



3.4.5 HERRAMIENTAS DE MONITORIZACIÓN DEL SISTEMA

Al invertir los programas, a veces es importante ver los cambios que una aplicación hacen en el sistema (especialmente cuando se analizan virus y malware); ¿Existen claves de registro creadas o consultadas? ¿Se han creado archivos .ini? ¿Se crearon procesos separados, tal vez para frustrar la ingeniería inversa de la aplicación?

Ejemplos de herramientas de monitorización del sistema son procmon, RegShot.

3.4.6 HERRAMIENTAS E INFORMACIÓN MISCELÁNEOS

Existen herramientas que se irán usando por el camino, como scripts, desempaquetadores, identificadores de empaquetadores etc. También se hará referencia a la API de Windows. Ello es extremadamente útil a la hora de estudiar las llamadas de las funciones CALL.

Capítulo 4

INTRODUCCIÓN A OLLYDBG

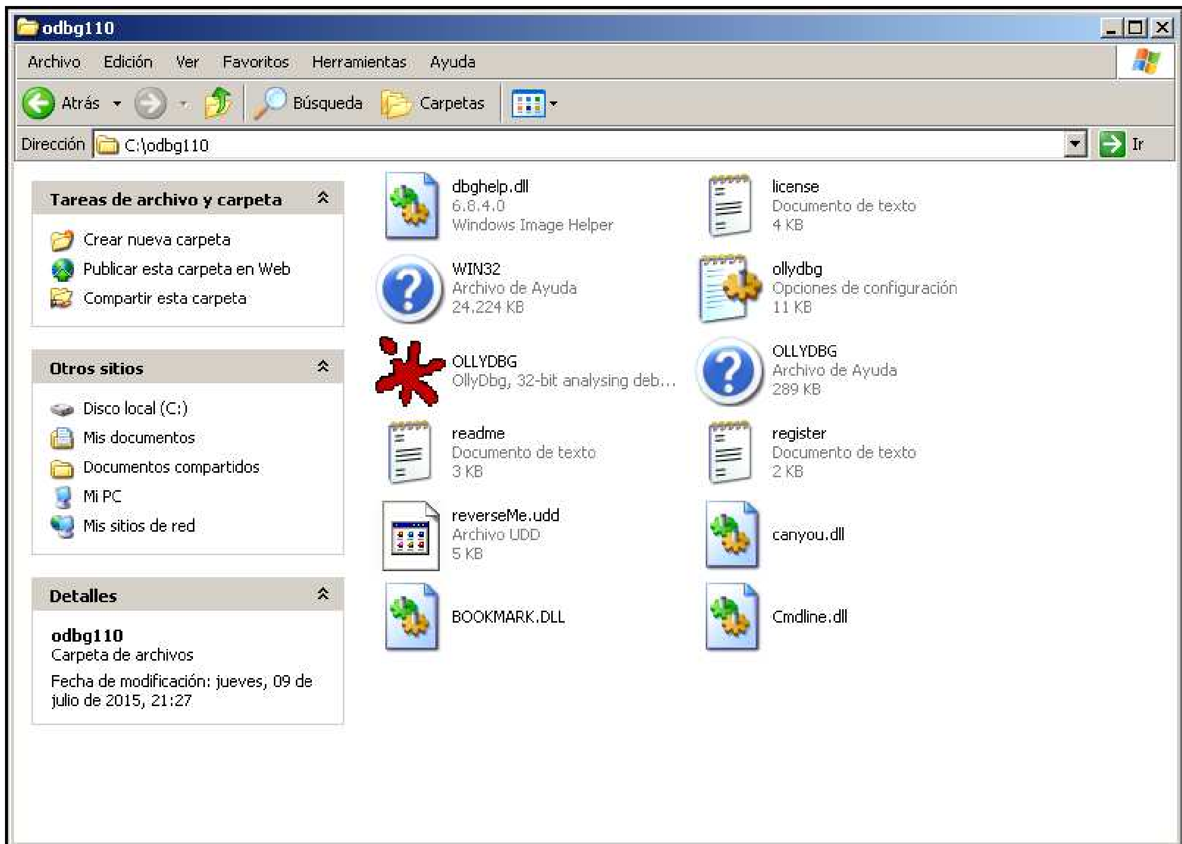
4.1 ¿Qué es Ollydbg?

Del autor, Oleh Yuschuk, OllyDbg es un depurador de código ensamblador de 32 bits para sistemas operativos Microsoft Windows. Pone especial énfasis en el análisis del código binario, esto lo hace muy útil cuando no está disponible el código fuente del programa. Además traza registros, reconoce procedimientos, llama a las API's, swiches, tablas, constantes y strings y localiza rutinas de archivos objeto y de bibliotecas.

4.2 Visión general

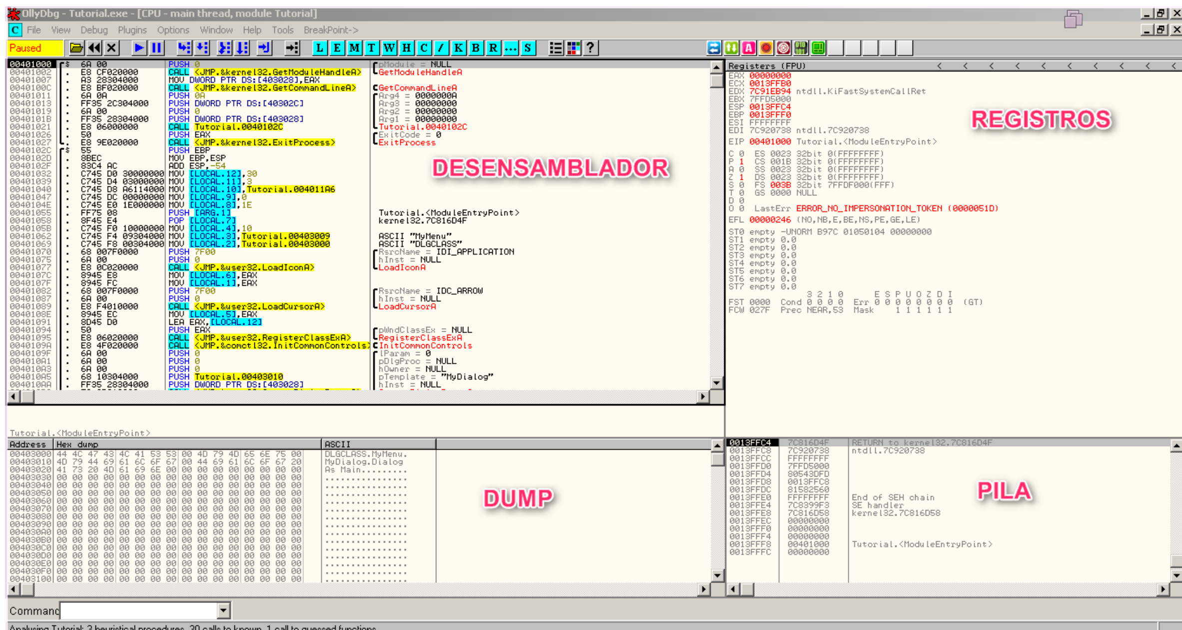
Descargamos la aplicación de la página oficial, http://www.ollydbg.de/, descomprimimos el fichero y lo guardamos preferiblemente en C:\

El contenido de la carpeta puede variar dependiendo de las distintas utilidades que se hayan instalado. Más adelante se describirá como añadir plugins a la aplicación. Para abrir la aplicación hacemos doble clic en OLLYDBG.



Al abrir Ollydbg y seleccionar el ejecutable Tutorial.exe, aparecerá en pantalla una ventana dividida en cuatro secciones:

1. Desensamblador
2. Los registros
3. La Pila o Stack
4. Dump

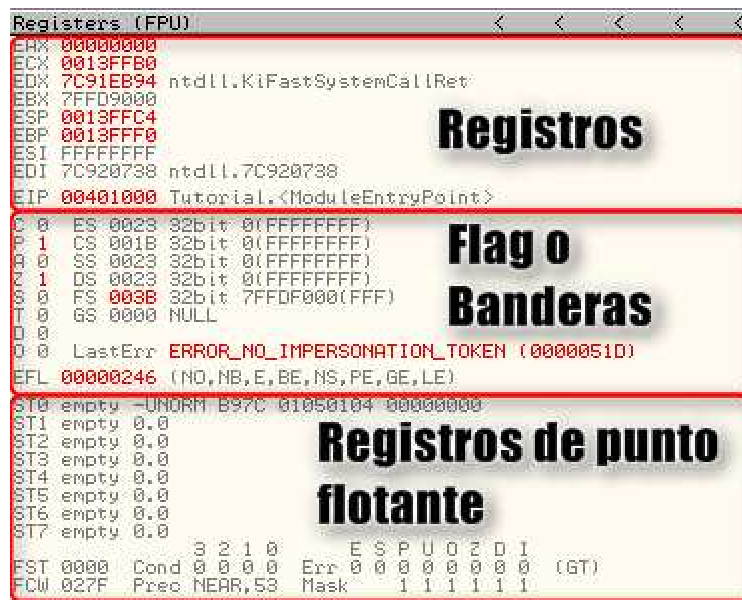


4.2.1 Desensamblador

Esta ventana contiene el desensamblaje principal del código binario. Aquí es donde Olly muestra información acerca del binario, incluyendo los códigos de operación (opcodes) y la traducción del lenguaje ensamblador. La primera columna es la dirección en memoria de la instrucción. La segunda columna corresponde a los opcodes. Cada instrucción tiene al menos un código asociado a él (muchos tienen múltiples).Este es el código que la CPU necesita para poder leer la aplicación. Estos códigos de operación constituyen el "lenguaje de la máquina". Si se quiere ver los datos en bruto de en un binario (utilizando un editor hexadecimal), solo se obtendrá una cadena de estos códigos de operación. Uno de los principales trabajos de Olly es desensamblar el "lenguaje de la máquina" en un lenguaje ensamblador más legible para el ser humano. En la tercera columna está el lenguaje ensamblador. Aunque a simple vista no parece que se haya obtenido una mejora en cuanto a su ilegibilidad, en conjunto ofrece mucho más datos que el código original. En la última columna Olly muestra los comentarios asociados a una determinada línea de código. A veces, este contiene los nombres de las llamadas a la API (si Olly puede descifrarlos) como CreateWindow y GetDigItemX. Olly también trata de ayudarnos a entender el código nombrando a las llamadas que no forman parte de la API con nombres de la ayuda. Por supuesto, éstas no son tan útiles, pero Olly también nos permite convertirlos en nombres más significativos. También podemos poner nuestros propios comentarios en esta columna haciendo doble clic en una fila del código. Estos comentarios se pueden guardar para la próxima vez que se cargue la aplicación.

4.2.2 Los Registros

Cada CPU tiene una colección de registros, donde se guardan temporalmente valores, al igual que una variable en cualquier lenguaje de programación de alto nivel. A continuación se muestra una vista más detallada de los registros en Olly.



En la parte superior están los registros de la CPU. Los registros cambian de color según se vaya ejecutando el código de la aplicación lo que ayuda a ver los cambios. También se puede cambiar el contenido de un registro haciendo doble clic sobre el mismo.

En la sección central están las banderas o flags. La CPU utiliza las banderas para marcar los cambios en el código (dos números son iguales, un número es mayor que otro, etc). Haciendo doble clic en una de las banderas cambia su valor de “True” a “False” y viceversa.

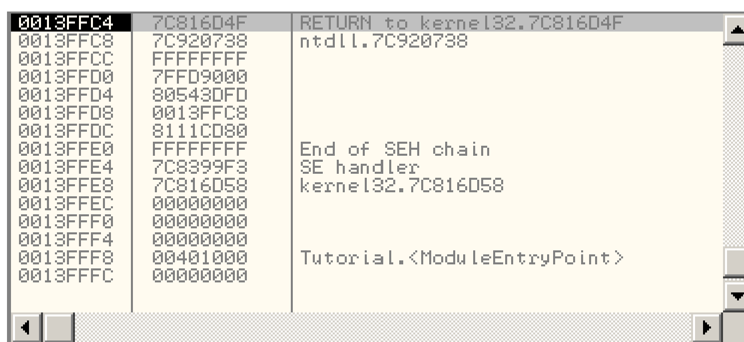
En la sección inferior están los FPU o Floating Point Registers. Estos se utilizan cada vez que la CPU realiza alguna operación aritmética con decimales. Cabe mencionar que dentro de la ingeniería inversa pocas veces se va a tener que recurrir a ellos, excepto cuando se utiliza el cifrado.

4.2.3 La Pila o Stack

La pila es una sección de memoria reservada para el binario como una lista "temporal" de los datos. Estos datos incluyen punteros a direcciones en la memoria, las cadenas, los marcadores, y lo más importante, las direcciones a las que tiene que regresar el código cuando hace una llamada a una función. Cuando un método en un programa llama a otro método, el controlador necesita estar desplazado hacia ese nuevo método para que pueda regresar. La CPU debe realizar un seguimiento del nuevo método de manera que al finalizarlo pueda regresar al lugar donde fue llamado y continuar así ejecutando el código después de la llamada. La pila es por tanto el lugar donde la CPU mantiene la dirección de retorno.

Una de las características de la pila es que sigue la estructura "First In, Last Out". Podemos compararlo con una pila de platos en una cafetería. Cuando se empuja (push) un plato en la parte superior, todas los platos debajo son empujados hacia abajo. Cuando se quita (pop) un plato de la parte superior, todos los platos que estaban debajo suben un nivel.

La siguiente imagen representa la pila en Olly. En la primera columna aparecen las direcciones de cada miembro de datos, la segunda columna es la representación hexadecimal de los datos de 32-bit, y en la última columna Olly muestra los comentarios de aquellos datos de los que posee alguna información. En la primera fila por ejemplo, se puede ver el comentario "RETURN to kernel ...". Esta dirección, que la CPU ha colocado en la pila, servirá como punto de referencia al cual habrá que regresar cuando la función haya finalizado.



Otras de las características de la pila es que haciendo clic con el botón derecho se puede 'modificar' el contenido de la misma.

4.2.4 Dump

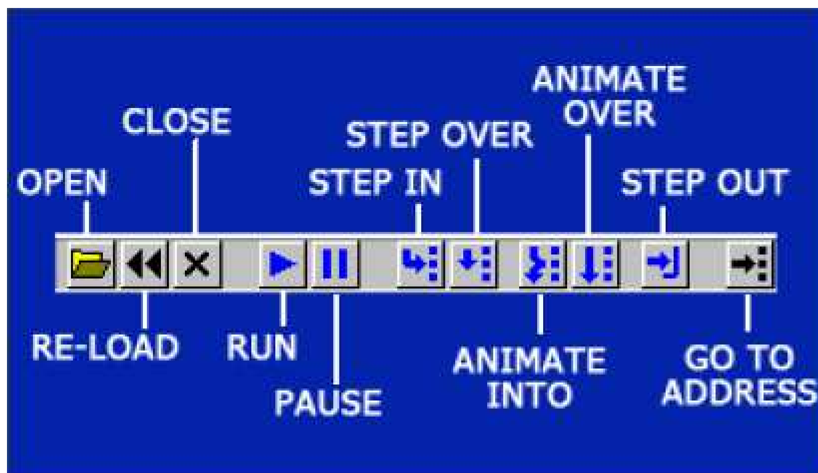
Al principio de este capítulo, cuando hablamos de los códigos de operación (opcodes) que la CPU lee en el interior de un binario, mencionamos que se podían ver estos datos en bruto con un visor hexadecimal. Pues bien, Olly incorpora un visor hexadecimal llamado "Dump ", que permite ver esos datos binarios en la memoria.

En la figura siguiente podemos observar como Olly muestra estos datos en hexadecimal y en ASCII. La primera columna muestra la dirección en la memoria de esos datos.

```
Address  Hex dump                                         ASCII
00403000 44 4C 47 43 4C 41 53 53 00 4D 79 4D 65 6E 75 00  DLGCLASS.MyMenu.
00403010 4D 79 44 69 61 6C 6F 67 00 44 69 61 6C 6F 67 20  MyDialog.Dialog
00403020 41 73 20 4D 61 69 6E 00 00 00 00 00 00 00 00 00  As Main.........
00403030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

4.2.5 La barra de herramientas

En la figura que se muestra a continuación aparecen marcados los iconos de la parte izquierda de la barra de herramientas. Todos ellos son accesibles a través del menú Debug.



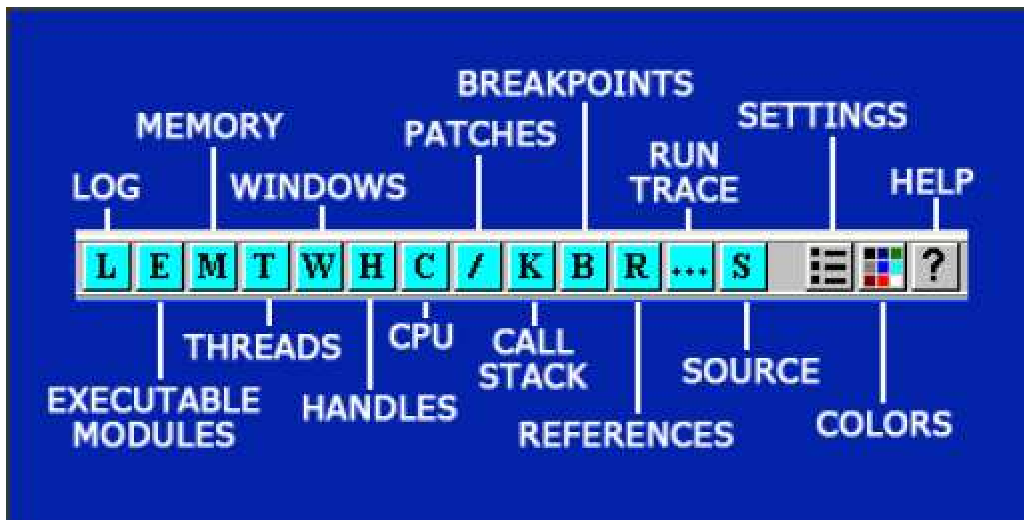
Los botones a tener en cuenta son el de reinicio (**RE-LOAD**), que reinicia la aplicación para pausarla en el 'Entry Point'. Durante el reinicio se borrarán todos los parches y algunos puntos de interrupción (breakpoints) pueden ser inhabilitados.

Con '**STEP IN**' se ejecuta una línea de código y después se detiene. En el caso de un call, hace una llamada a la función si es que existe.

'**STEP OVER**' hace lo mismo, pero saltando por encima de un call hacia otra función.

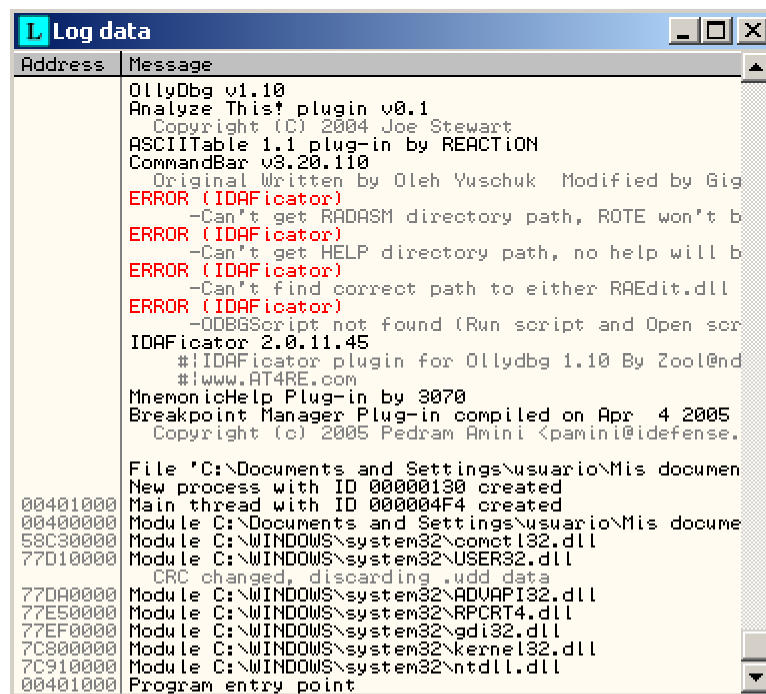
'**ANIMATE INTO**' y '**ANIMATE OVER**' hacen lo mismo que las ordenes anteriores, excepto que en estos casos el código se ejecuta lo suficientemente despacio para poder observar cómo se va ejecutando. Puede ser útil en los binomios polimórficos.

En la parte derecha de la barra de herramientas tenemos los siguientes botones:



Cada uno de estos botones abre una ventana nueva con el contenido que veremos a continuación. Podemos acceder a cualquier ventana de esta parte de la barra de herramientas a través del menú, haciendo clic en 'View'.

La ventana Log data no necesita mayor explicación. En este caso en particular se puede apreciar los errores que encontró Olly a la hora de ejecutar el plugin IDAFicator.



La siguiente ventana muestra los módulos ejecutables.

Executable modules

Base	Size	Entry	Name	File version	Path
00400000	00005000	00401000	Tutorial		C:\Documents and Settings\usuario\
58C30000	00097000	58C332DA	comctl32	5.82 (xpsp_sp2_	C:\WINDOWS\system32\comctl32.dll
77D10000	00090000	77D20EB9	USER32	5.1.2600.2180 (	C:\WINDOWS\system32\USER32.dll
77DA0000	000AC000	77DA70D4	ADVAPI32	5.1.2600.2180 (	C:\WINDOWS\system32\ADVAPI32.dll
77E50000	00091000	77E56284	RPCRT4	5.1.2600.2180 (	C:\WINDOWS\system32\RPCRT4.dll
77EF0000	00046000	77EF63CA	gdi32	5.1.2600.2180 (	C:\WINDOWS\system32\gdi32.dll
7C800000	00101000	7C80B436	kernel32	5.1.2600.2180 (	C:\WINDOWS\system32\kernel32.dll
7C910000	000B6000	7C923156	ntdll	5.1.2600.2180 (	C:\WINDOWS\system32\ntdll.dll

En la ventana 'Memory' se puede visualizar todos los bloques de la memoria que el programa tiene asignado. Incluye una sección principal de la aplicación que se está ejecutando. También podemos observar un listado de todos los DLL's que el programa ha cargado en la memoria para usarlos cuando sea necesario. Si hacemos clic en cualquiera de las lineas se abrirá la ventana Dump con todo el contenido de esa línea. Finalmente, podemos ver el tipo de bloque, los permisos de acceso, el tamaño y la dirección de la memoria donde cada sección ha sido cargada.

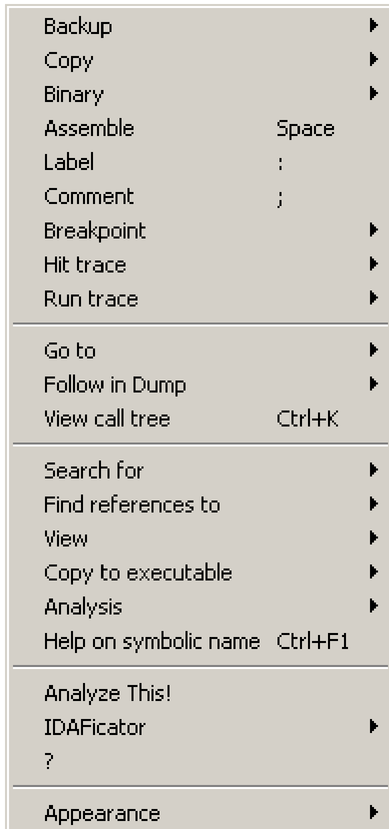
Memory map

Address	Size	Owner	Section	Contains	Type	Access	Initial access	Mapped as
00010000	00001000				Priv 00021004	RW	RW	
00020000	00001000				Priv 00021004	RW	RW	
00030000	00001000				Priv 00021004	RW	RW	
0013D000	00001000				Priv 00021104	RW Guarded	RW	
0013E000	00002000			stack of main thread	Priv 00021104	RW Guarded	RW	
00140000	00003000				Map 00041002	R	R	
00150000	00004000				Priv 00021004	RW	RW	
00250000	00006000				Priv 00021004	RW	RW	
00260000	00003000				Map 00041004	RW	RW	
00270000	00016000				Map 00041002	R	R	\Device\HarddiskVo
00290000	0003D000				Map 00041002	R	R	\Device\HarddiskVo
002D0000	00041000				Map 00041002	R	R	\Device\HarddiskVo
00320000	00006000				Map 00041002	R	R	\Device\HarddiskVo
00330000	00041000				Map 00041002	R	R	
00380000	00001000				Priv 00021004	RW	RW	
00390000	00004000				Priv 00021004	RW	RW	
003A0000	00003000				Map 00041002	R	R	\Device\HarddiskVo
00400000	00001000	Tutorial		PE header	Imag 01001002	R	RWE	
00401000	00001000	Tutorial	.text	code	Imag 01001002	R	RWE	
00402000	00001000	Tutorial	.rdata	imports	Imag 01001002	R	RWE	
00403000	00001000	Tutorial	.data	data	Imag 01001002	R	RWE	
00404000	00001000	Tutorial	.rsrc	resources	Imag 01001002	R	RWE	
00410000	00002000				Map 00041020	R E	R E	
004D0000	00002000				Map 00041020	R E	R E	
004E0000	00103000				Map 00041002	R	R	
005F0000	0003F000				Map 00041020	R E	R E	
58C30000	00001000	comctl32		PE header	Imag 01001002	R	RWE	
58C31000	00070000	comctl32	.text	code,imports,exports	Imag 01001002	R	RWE	
58CA1000	00003000	comctl32	.data	data	Imag 01001002	R	RWE	
58CA4000	0001F000	comctl32	.rsrc	resources	Imag 01001002	R	RWE	
58CC3000	00004000	comctl32	.reloc	relocations	Imag 01001002	R	RWE	
77D10000	00001000	USER32		PE header	Imag 01001002	R	RWE	
77D11000	0005F000	USER32	.text	code,imports,exports	Imag 01001002	R	RWE	
77D70000	00002000	USER32	.data	data	Imag 01001002	R	RWE	
77D72000	0002B000	USER32	.rsrc	resources	Imag 01001002	R	RWE	
77D9D000	00003000	USER32	.reloc	relocations	Imag 01001002	R	RWE	
77DA0000	00001000	ADVAPI32		PE header	Imag 01001002	R	RWE	
77DA1000	00075000	ADVAPI32	.text	code,imports,exports	Imag 01001002	R	RWE	
77E16000	00005000	ADVAPI32	.data	data	Imag 01001002	R	RWE	
77E1B000	0002C000	ADVAPI32	.rsrc	resources	Imag 01001002	R	RWE	
77E47000	00005000	ADVAPI32	.reloc	relocations	Imag 01001002	R	RWE	
77E50000	00001000	RPCRT4		PE header	Imag 01001002	R	RWE	
77E51000	00082000	RPCRT4	.text	code,imports,exports	Imag 01001002	R	RWE	
77ED3000	00007000	RPCRT4	.orpc	code	Imag 01001002	R	RWE	
77EDA000	00001000	RPCRT4	.data	data	Imag 01001002	R	RWE	

El contenido de las ventanas restantes se irá viendo según vayamos progresando en la resolución de los casos prácticos.

4.2.6 El menú contextual

Para acceder al menú contextual, hacemos clic con el botón derecho en el área del desensamblador. Veremos el significado de la mayoría de los comandos en la sección de casos prácticos. Señalar aquí que el contenido del menú contextual puede variar dependiendo de los plugins que hayamos instalado.



Capítulo 5

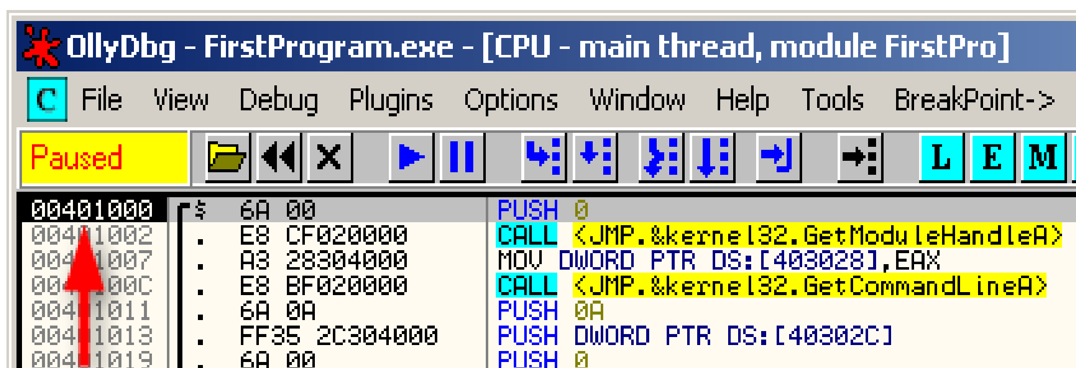
OLLYDBG (I)

5.1 Cargando una aplicación

Para cargar un binario en Olly basta con hacer clic en la carpeta, seleccionar el destino y hacer doble clic en el ejecutable.

Olly hará un análisis del binario y parará en el 'Entry Point' (EP).

De la primera columna podemos observar que el EP está en la dirección 401000. Para ejecutables que no hayan sido empaquetados u ofuscados, esta será el punto de partida estándar.



Cargando el binario FirstProgram.exe

Lo primero que haremos es mirar el espacio que el programa ocupa en la memoria. Para ello pulsamos sobre el icono ‘M’.

De la primera columna podemos observar que la dirección 401000 es de 1000 bytes; el nombre del propietario es “FirstPro”; el nombre de la sección que contiene el código es “.text”.

Memory map

Address	Size	Owner	Section	Contains	Type	Access	Initial access	Mapped as
00010000	00001000				Priv 00021004	RW	RW	
00020000	00001000				Priv 00021004	RW	RW	
00030000	00001000				Priv 00021004	RW	RW	
0013D000	00001000				Priv 00021104	RW Guarded	RW	
0013E000	00002000			stack of main thread	Priv 00021104	RW Guarded	RW	
00140000	00003000				Map 00041002	R	R	
00150000	00004000				Priv 00021004	RW	RW	
00250000	00006000				Priv 00021004	RW	RW	
00260000	00003000				Map 00041004	RW	RW	
00270000	00016000				Map 00041002	R	R	\Device\HarddiskV
00290000	0003D000				Map 00041002	R	R	\Device\HarddiskV
002D0000	00041000				Map 00041002	R	R	\Device\HarddiskV
00320000	00006000				Map 00041002	R	R	\Device\HarddiskV
00330000	00041000				Map 00041002	R	R	
00380000	00001000				Priv 00021004	RW	RW	
00390000	00004000				Priv 00021004	RW	RW	
003A0000	00003000				Map 00041002	R	R	\Device\HarddiskV
00400000	00001000	FirstPro		PE header	Imag 01001002	R	RWE	
00401000	00001000	FirstPro	.text	code	Imag 01001002	R	RWE	
00402000	00001000	FirstPro	.rdata	imports	Imag 01001002	R	RWE	
00403000	00001000	FirstPro	.data	data	Imag 01001002	R	RWE	
00404000	00001000	FirstPro	.rsrc	resources	Imag 01001002	R	RWE	
00410000	00002000				Map 00041020	R E	R E	
004D0000	00002000				Map 00041020	R E	R E	
004E0000	00103000				Map 00041002	R	R	
005F0000	0005E000				Map 00041020	R E	R E	
58C30000	00001000	comctl32		PE header	Imag 01001002	R	RWE	
58C31000	00070000	comctl32	.text	code,imports,exports	Imag 01001002	R	RWE	
58CA1000	00003000	comctl32	.data	data	Imag 01001002	R	RWE	
58CA4000	0001F000	comctl32	.rsrc	resources	Imag 01001002	R	RWE	
58CC3000	00004000	comctl32	.reloc	relocations	Imag 01001002	R	RWE	
77D10000	00001000	USER32		PE header	Imag 01001002	R	RWE	
77D11000	0005F000	USER32	.text	code,imports,exports	Imag 01001002	R	RWE	
77D70000	00002000	USER32	.data	data	Imag 01001002	R	RWE	
77D72000	0002B000	USER32	.rsrc	resources	Imag 01001002	R	RWE	
77D9D000	00003000	USER32	.reloc	relocations	Imag 01001002	R	RWE	
77DA0000	00001000	ADVAPI32		PE header	Imag 01001002	R	RWE	
77DA1000	00075000	ADVAPI32	.text	code,imports,exports	Imag 01001002	R	RWE	
77E16000	00005000	ADVAPI32	.data	data	Imag 01001002	R	RWE	
77E1B000	0002C000	ADVAPI32	.rsrc	resources	Imag 01001002	R	RWE	
77E47000	00005000	ADVAPI32	.reloc	relocations	Imag 01001002	R	RWE	
77E50000	00001000	RPCRT4		PE header	Imag 01001002	R	RWE	
77E51000	00082000	RPCRT4	.text	code,imports,exports	Imag 01001002	R	RWE	
77ED3000	00007000	RPCRT4	.orpc	code	Imag 01001002	R	RWE	

Una de las secciones más importantes es "PE Header". Podemos compararlo con un manual de instrucciones para Windows en el que se especifican los pasos necesarios para cargar el programa en la memoria, la cantidad de espacio que necesita para funcionar, etc. Se sitúa en la cabecera de casi cualquier ejecutable.

Más abajo aparecen archivos distintos al de nuestra aplicación FirstPro, como por ejemplo comctl32, gdi32, kernel32 etc. Estos son los archivos DLL que nuestra aplicación necesita para funcionar. Un archivo DLL (Dynamic-Link Library) es una colección de funciones que nuestro programa puede llamar y que han sido proporcionados por Windows (u otro programa). Su función es abrir los cuadros de texto, la comparación de secuencias, la creación de ventanas y similares. El conjunto de estos archivos se conoce como API (Application Programming Interface) de Windows. La razón por la cual los programas utilizan estas funciones es para facilitar al programador la tarea de escribir miles de líneas de código cada vez que quiera mostrar por ejemplo una ventana. En su lugar, Windows proporciona una función como CreateWindow.

Cuando Windows carga nuestro ejecutable en la memoria, comprueba la cabecera “PE Header”, lee los nombres de las DLLs y sus funciones asociadas, luego carga estas funciones en el espacio que la memoria tiene asignada para ellos. Si queremos ver exactamente que funciones son llamadas por nuestro programa, hacemos clic con el botón derecho en el desensamblador de Olly y seleccionamos "Search for" -> "All intermodular calls”.

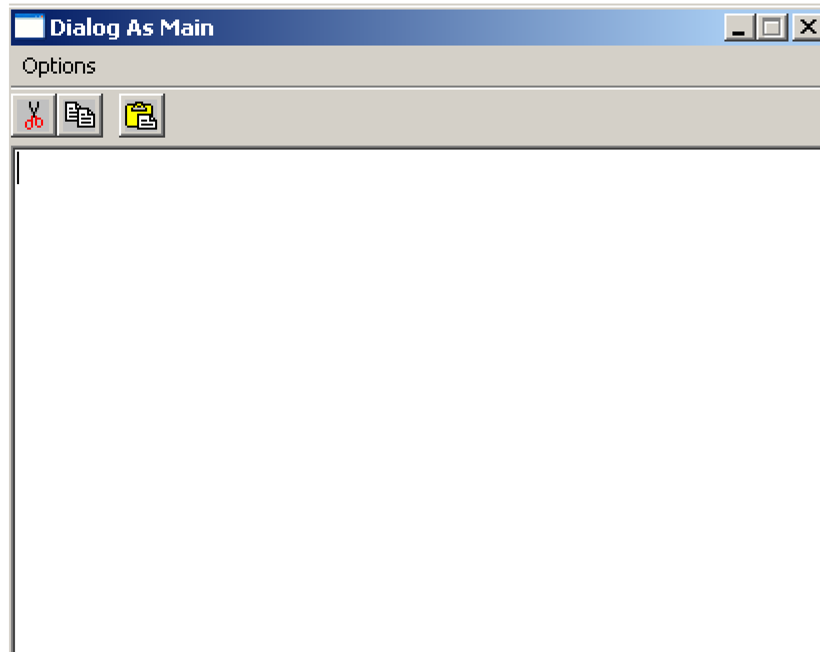
Found intermodular calls

Address	Disassembly	Destination
00401002	CALL <JMP.&kernel32.GetModuleHanc	kernel32.GetModuleHandleA
0040100C	CALL <JMP.&kernel32.GetCommandLin	kernel32.GetCommandLineA
00401027	CALL <JMP.&kernel32.ExitProcess>	kernel32.ExitProcess
00401032	MOV [LOCAL.12],30	(Initial CPU selection)
00401077	CALL <JMP.&user32.LoadIconA>	USER32.LoadIconA
00401089	CALL <JMP.&user32.LoadCursorA>	USER32.LoadCursorA
00401095	CALL <JMP.&user32.RegisterClassEx	USER32.RegisterClassExA
0040109A	CALL <JMP.&comctl32.InitCommonCon	comctl32.InitCommonControls
004010B0	CALL <JMP.&user32.CreateDialogPar	USER32.CreateDialogParamA
004010C9	CALL <JMP.&user32.SendDlgItemMess	USER32.SendDlgItemMessageA
004010E2	CALL <JMP.&user32.SendDlgItemMess	USER32.SendDlgItemMessageA
004010F1	CALL <JMP.&comctl32.ImageList_Cre	comctl32.ImageList_Create
0040110C	CALL <JMP.&user32.LoadImageA>	USER32.LoadImageA
00401118	CALL <JMP.&comctl32.ImageList_Add	comctl32.ImageList_Add
0040111F	CALL <JMP.&gdi32.DeleteObject>	gdi32.DeleteObject
00401136	CALL <JMP.&user32.SendDlgItemMess	USER32.SendDlgItemMessageA
00401143	CALL <JMP.&user32.GetDlgItem>	USER32.GetDlgItem
00401149	CALL <JMP.&user32.SetFocus>	USER32.SetFocus
00401153	CALL <JMP.&user32.ShowWindow>	USER32.ShowWindow

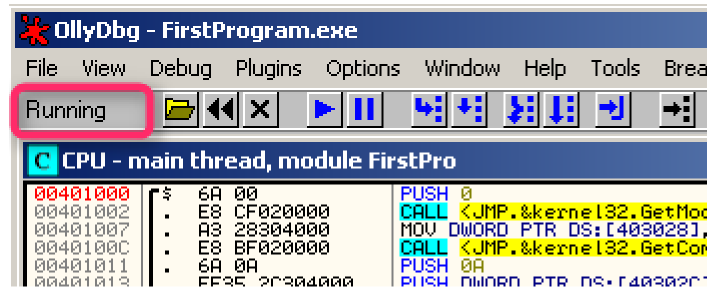
Esta ventana muestra el nombre de la DLL seguido por el nombre de la función. Por ejemplo, USER32.LoadIconA está en la DLL USER32 y el nombre de la función es LoadIconA. Esta función es la encargada de carga el icono en la esquina superior izquierda de la ventana.

5.2 ejecutando una aplicación

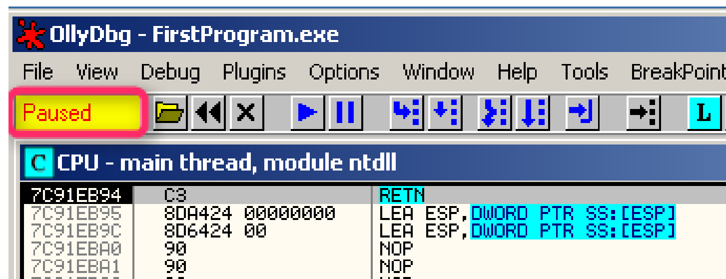
Para ejecutar una aplicación dentro de Olly haremos clic en Run desde el menú Debug o pulsamos F9. A los pocos segundos aparecerá un cuadro de texto.



En la casilla de la esquina superior izquierda podemos ver como Olly cambio de “Paused” a “Running”. Esto quiere decir que el programa se está ejecutando, pero dentro de Olly.

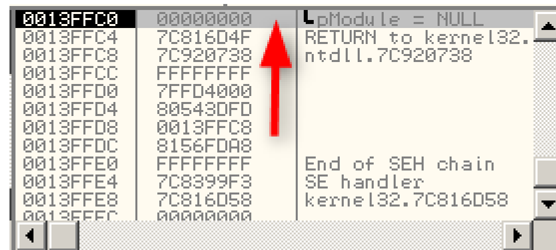


Si ahora hacemos clic en el icono de pausa (pulsando F12, o seleccionando Debug -> Pause) se detendrá nuestro programa dondequiera que esté ejecutándose en la memoria.



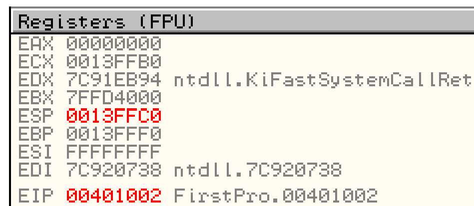
5.3 Ejecutando paso a paso

Para ejecutar el programa paso a paso hay que pulsar F8 (Step Over). El selector de línea se desplaza una línea hacia abajo y Olly ejecuta esa línea de instrucciones para detenerse en la siguiente línea. En la pila podemos ver una nueva entrada en la parte superior:



Esto es debido a la instrucción PUSH 0, que ha puesto un cero encima de la pila. Esta información nos viene dado por la expresión pModule = NULL.

Si nos fijamos en la ventana de registros podemos observar como los registros ESP y EIP cambiaron de color gris a rojo.



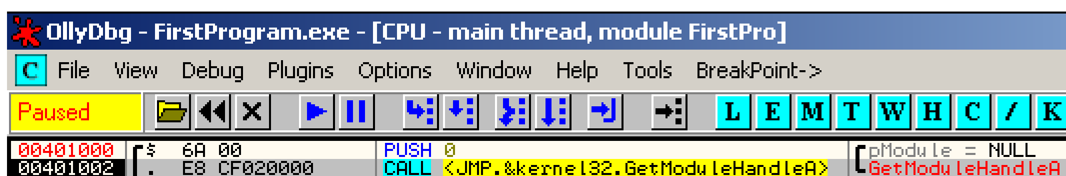
Cuando un registro se convierte en rojo, significa que la última ejecución de instrucciones cambió el valor de ese registro. En este caso, el registro ESP, que apunta a la dirección de la parte superior de la pila, se ha incrementa en uno ya que 'empujamos' un nuevo valor a la pila. El registro EIP, que apunta a la instrucción en curso, también aumentó en dos. Esto se debe a que ya no estamos en la dirección 401000, sino en la dirección 401002.

La instrucción en la que Olly se detuvo es ahora un CALL. Una instrucción CALL significa que queremos detenernos temporalmente en la función actual para ejecutar otra función. Esto es análogo a llamar a un método dentro de un lenguaje de alto nivel, por ejemplo:

```
int main()

{

int x = 1;

call doSomething();

x = x + 1;

}
```

En este código, lo primero que hacemos es igualar x a 1, después detenemos esta expresión lógica y en su lugar llamamos a la función doSomething(). Cuando doSomething() haya acabado, volveremos a reanudar nuestra expresión lógica original aumentando la x por 1.

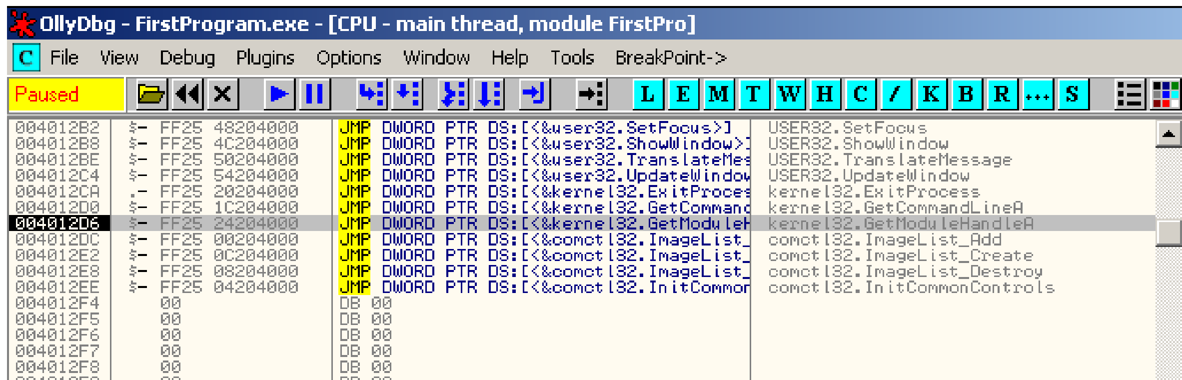
El mismo razonamiento podemos emplearlo en el lenguaje de ensamblado. Primero empujamos un cero encima de la pila y después queremos llamar una función, que en este caso es una función situada dentro del Kernel32.dll y con nombre GetModuleHandleA().



Volvemos a pulsar F8. El indicador de línea se moverá una posición hacia abajo, el registro EIP permanecerá rojo y aumentará su valor en 5 bytes, que coincide con la longitud de la instrucción. La pila regresará de vuelta a su valor original.

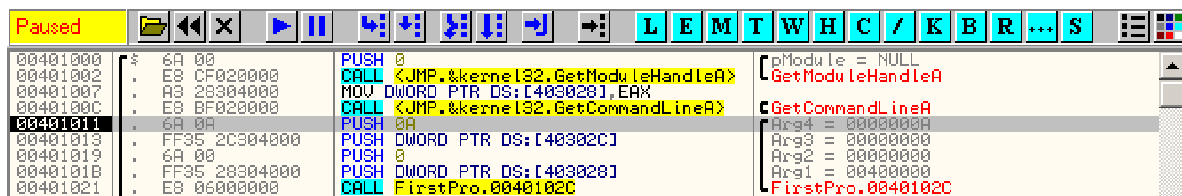
Lo que aquí ocurrió es que al presionar F8, que significa "Step-Over", se ejecutó el código dentro de la llamada call y Olly se detuvo en la siguiente línea. Ahora, en el interior de esta llamada el programa podría haber hecho cualquier cosa, pero nosotros hemos pasado por encima de la misma.

Para ver el funcionamiento de la otra opción, reiniciamos el programa, y pulsamos F8 para saltar por encima de la primera instrucción y F7 cuando llegamos a la instrucción CALL. En esta ocasión la ventana tiene el siguiente aspecto:



Al pulsar F7 (Step-In), Olly ejecutó la instrucción CALL y se detuvo en la primera línea de la nueva función. Vemos como la aplicación saltó a una nueva área dentro de la memoria (EIP = 4012D6). Teóricamente, si seguimos paso a paso a través de estas líneas de código, tendríamos finalmente que volver a la declaración que nos trajo hasta aquí. Por ahora, vamos a reiniciar el programa y empezamos de nuevo.

Pulsamos F8 cuatro veces y nos paramos en la siguiente línea:



Una vez situados sobre el PUSH de la dirección 401011, pulsaremos cuatro veces F8 para observar como la pila va creciendo hacia abajo.

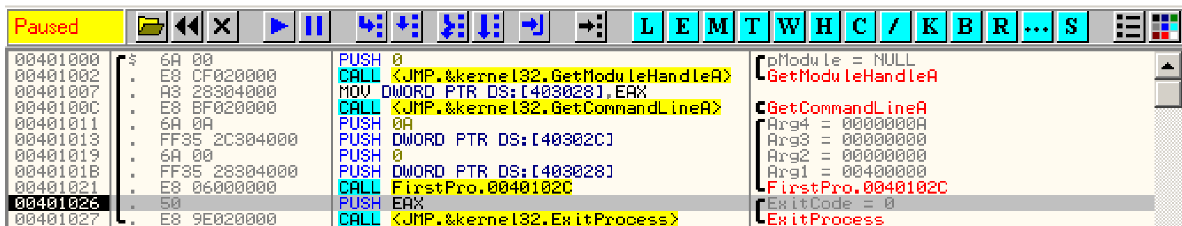
Estos cuatro números son empujados encima de la pila para ser pasados como parámetros a una función (la función que estamos a punto de llamar en la dirección 401021). Podemos verlo de forma más clara si lo transformamos a un lenguaje de alto nivel.

```
int main()

{

int x = 1;

int y = 0;

call doSomething( x, y );

x = x + 1;

}
```

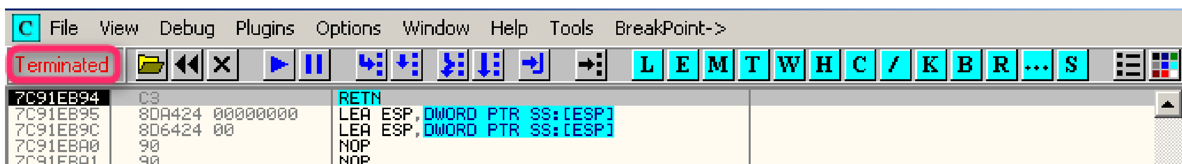
Declaramos dos variables X e Y, y las pasamos a la función doSomething. La función doSomething hará algo con estas variables, y después devuelve el control al programa que hizo la llamada. La pila es una de las principales formas en que las variables pueden ser pasadas a una función: cada variable es insertada en la pila, la función se llama para acceder a estas variables, por lo general utilizando la instrucción inversa de PUSH que es POP.

Otra forma de hacer lo mismo es poniendo las variables en los registros para ser llamados por la función que se sitúa dentro de la instrucción CALL, pero en este caso, el compilador de nuestro programa optó por ponerlos en la pila.

Ahora, si pulsamos F8 una vez más, vemos que Olly se pone en modo "Running" y vuelve aparecer el cuadro de texto. Esto se debe a que saltamos por encima de la llamada, que contiene la mayor parte del código. Dentro de este CALL el código entra en un bucle a la espera que hagamos algo, de modo que nunca avanzaremos para situarnos después de la llamada. Para arreglar esto cerramos el cuadro de dialogo. Olly se detendrá inmediatamente en la siguiente línea después de la llamada:

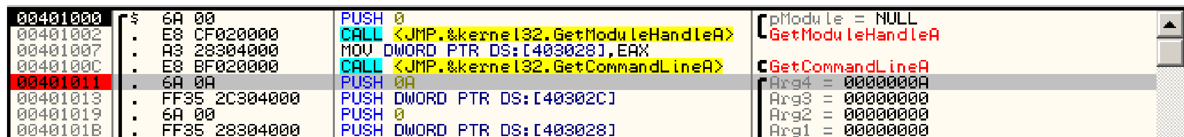


Si nos fijamos ahora en la siguiente línea, veremos que estamos a punto de llamar a kernel32.dll -> ExitProcess. Esta es la API de Windows que finaliza una aplicación. Vemos que Olly ha detenido nuestra programa después de haber cerrado la ventana de dialogo, sin embargo, ¡ la aplicación sigue ejecutándose ! Si ahora pulsamos F9, el programa sí terminará, y en la barra activa de Olly podemos leer "Terminated".



5.4 puntos de interrupción (breakpoints)

Los puntos de interrupción o “Breakpoints” obligan a Olly a detenerse cuando son alcanzados. Para poner un Breakpoint basta con hacer doble clic en el opcode. Una vez puesto el Breakpoint la dirección en memoria cambia al color rojo.



Breakpoint en 00401011

A la hora de detener la ejecución de una aplicación, contamos con varios tipos de Breakpoints.

5.4.1 Software Breakpoint

Un Software Breakpoint reemplaza el byte en la dirección del Breakpoint por el opcode 0xCC, que es un int3. Se trata de una interrupción especial que indica al sistema operativo que un depurador desea detenerse en ese punto para hacerse con el control del programa antes de continuar ejecutando la instrucción. El cambio de la instrucción a 0xCC, no será visible, pero cuando Olly llega al Breakpoint se produce una excepción y Olly atrapará la excepción para permitir al usuario hacer lo que él / ella desean cambiar. Si permitimos al programa que continúe ejecutándose (ya sea pasando o saltando), el código de operación 0xCC se sustituirá de nuevo por su valor original.

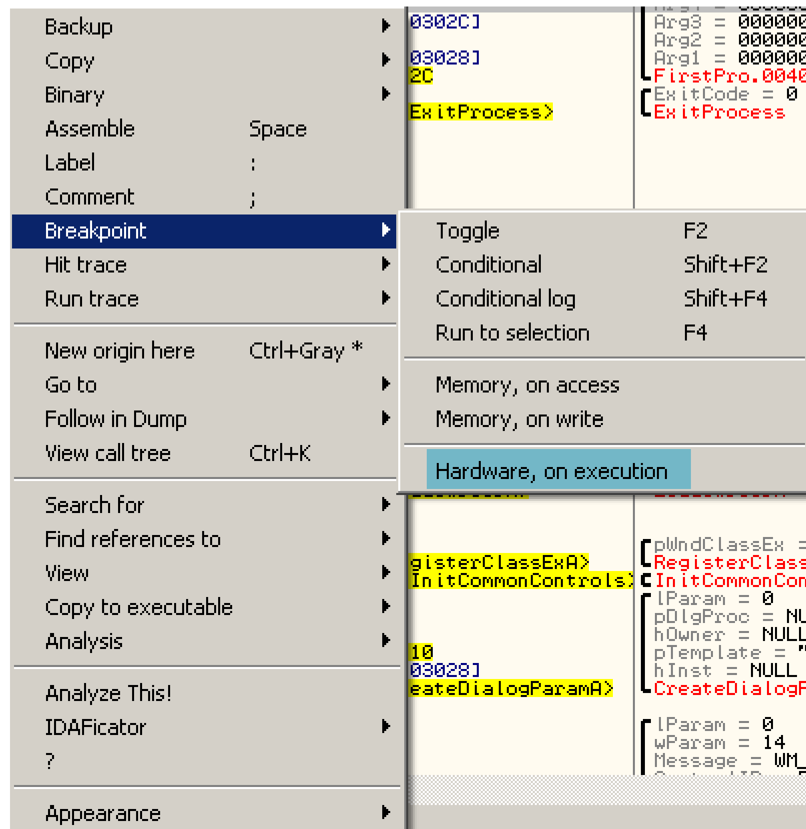
Para poner un Software Breakpoint, podemos hacer doble clic en el opcode de la segunda columna, o también clic con el botón derecho sobre la línea y seleccionar Breakpoints -> Toggle (F2). Para quitar un Breakpoint haremos doble clic sobre la misma línea, o clic con el botón derecho y selecciondar Breakpoint -> Remove Software Breakpoint (F2).

Reiniciamos de nuevo el programa y ponemos un Breakpoint en la dirección 401011. Desde el “Entry Point” pulsamos F9 o run, y el programa empezará a ejecutarse hasta llegar al Breakpoint, donde se detendrá.

Si queremos visualizar todos los Breakpoints que hemos puesto, pulsamos obre el botón “Br” en la barra de herramientas o seleccionamos “View” -> “Breakpoints”.

5.4.2 Hardware Breakpoints

Un hardware Breakpoint utiliza los registros depuradores de la CPU. Hay 8 de estos registros incorporados en la CPU, R0-R7. A pesar de que hay 8 integrados en el chip, sólo podemos usar cuatro de ellos. Estos pueden ser usados para detener la lectura, escritura o ejecución de una sección en la memoria. La diferencia entre el hardware y el software Breakpoint es que el hardware BP no cambia la memoria del procesador, de forma que es más fiable, especialmente en los programas que están empaquetados o protegidos. Los hardware Breakpoints se ponen haciendo clic con el botón derecho sobre la línea deseada y seleccionando "Breakpoint" -> "Hardware, on execution":



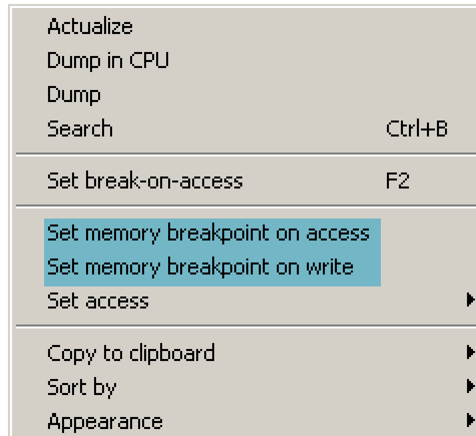
5.4.3 Memory Breakpoints

A veces se puede encontrar una cadena o una constante en la memoria del programa, sin saber desde que parte del programa se puede acceder a ello. Estableciendo un memory Breakpoint le decimos a Olly que se detenga cada vez que una instrucción lee o escribe en esa dirección de memoria (o grupos de direcciones). Hay tres formas de seleccionar un memory Breakpoint:

- Para una instrucción determinada, haciendo clic en la línea deseada y seleccionando "Breakpoint" -> "Memory, On Access" o "Memory, On Write".

- Para establecer un Breakpoint en una dirección situada en la memoria dump, resaltamos uno o más bytes en el dump, hacemos clic con el botón derecho y seleccionamos la misma opción que en el caso anterior.

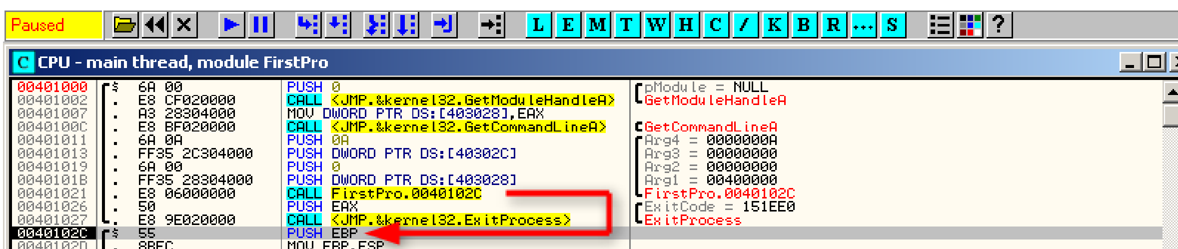
➢ También se puede establecer un Breakpoint para toda una sección de la memoria. Desde la ventana de memoria (icono "M" o “View” -> “Memory”), haciendo clic con el botón derecho en la sección de la memoria deseada, y seleccionando con el botón derecho "Set memory break on access” o “Set memory break on write”.



5.5 Utilizando el panel Dump

El panel Dump se puede utilizar para inspeccionar el contenido de cualquier posición en memoria dentro del espacio de memoria del depurador. Si una instrucción en la ventana de desensamblaje, un registro, o cualquier elemento de la pila contiene una referencia a una posición en la memoria, podemos hacer clic con el botón derecho en la referencia y seleccionar "Follow in Dump" y el panel Dump mostrará la sección que corresponde a esa dirección. También podemos hacer clic con el botón derecho en cualquier lugar del panel Dump y seleccionar "Go To" para introducir la dirección que deseamos ver.

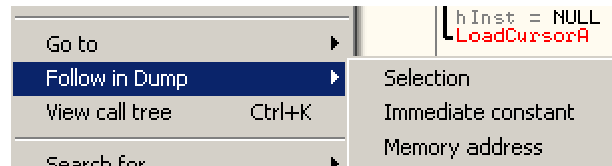
Cargamos nuestro FirstProgram.exe y comprobamos que se ha detenido en el “Entry Point”. A continuación presionamos F8 ocho veces hasta llegar a la instrucción correspondiente a la dirección 401021 que dice CALL FirstPro.40102C. Si nos fijamos en esta línea, nos daremos cuenta de que este CALL va a saltar a la dirección 40102C, que se sitúa 3 líneas por debajo de donde estamos actualmente. Pulsamos F7 para coger el salto y situarnos en 40102C. Recordemos que al tratarse de una instrucción CALL, volveremos a regresar a la dirección 401021 (o al menos a la instrucción siguiente).



Seguimos pulsando (F8) hasta llegar a la dirección 401062.

```
0040105B  .  C745 F0 10000000 MOV [LOCAL.4],10
00401062  .  C745 F4 09304000 MOV [LOCAL.3],FirstPro.00403009     ASCII "MyMenu"
00401069  .  C745 F8 00304000 MOV [LOCAL.2],FirstPro.00403000     ASCII "DLGCLASS"
```

En esta línea la instrucción es: MOV [LOCAL.3], FirstPro.00403009. De acuerdo al lenguaje ensamblador, esta instrucción mueve el contenido de la dirección 00403009 a la pila (lo que Olly identifica como LOCAL.3). En la columna de los comentarios se puede ver lo que Olly ha descubierto en esta dirección: la cadena ASCII "MyMenu". A continuación hacemos clic con el botón derecho en la instrucción y seleccionamos "Follow in Dump".



Seleccionamos "Immediate constant". Esto cargará cualquier dirección afectada por la instrucción. Si hubiéramos elegido "Selection", la ventana Dump habría mostrado la dirección de la línea resaltada, en este caso 401062 (la línea en la que nos detuvimos). Estaríamos viendo en el dump lo que ya hemos visto en la ventana de desensamblado. Por último, si habríamos elegido "Memory address", la ventana Dump mostraría la memoria para LOCAL.3. En efecto, mostraría la memoria en la cual se encuentran las variables locales dentro de la pila.

```
Address  Hex dump                                         ASCII
00403009 4D 79 4D 65 6E 75 00 4D 79 44 69 61 6C 6F 67 00  MyMenu.MyDialog.
00403019 44 69 61 6C 6F 67 20 41 73 20 4D 61 69 6E 00 00  Dialog As Main..
00403029 00 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .@..............
00403039 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403049 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403059 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403069 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403079 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403089 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403099 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030A9 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030B9 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Como se puede ver, el dump muestra ahora la memoria a partir de la dirección 403009, lugar en el que Olly cargó la cadena ASCII. A la derecha, se puede ver la cadena, "MyMenu". A la izquierda tenemos la interpretación hexadecimal de cada carácter que conforma la cadena ASCII. Las cadenas siguientes se utilizaran en otras partes del programa.

Capítulo 6

OLLYDBG(II)

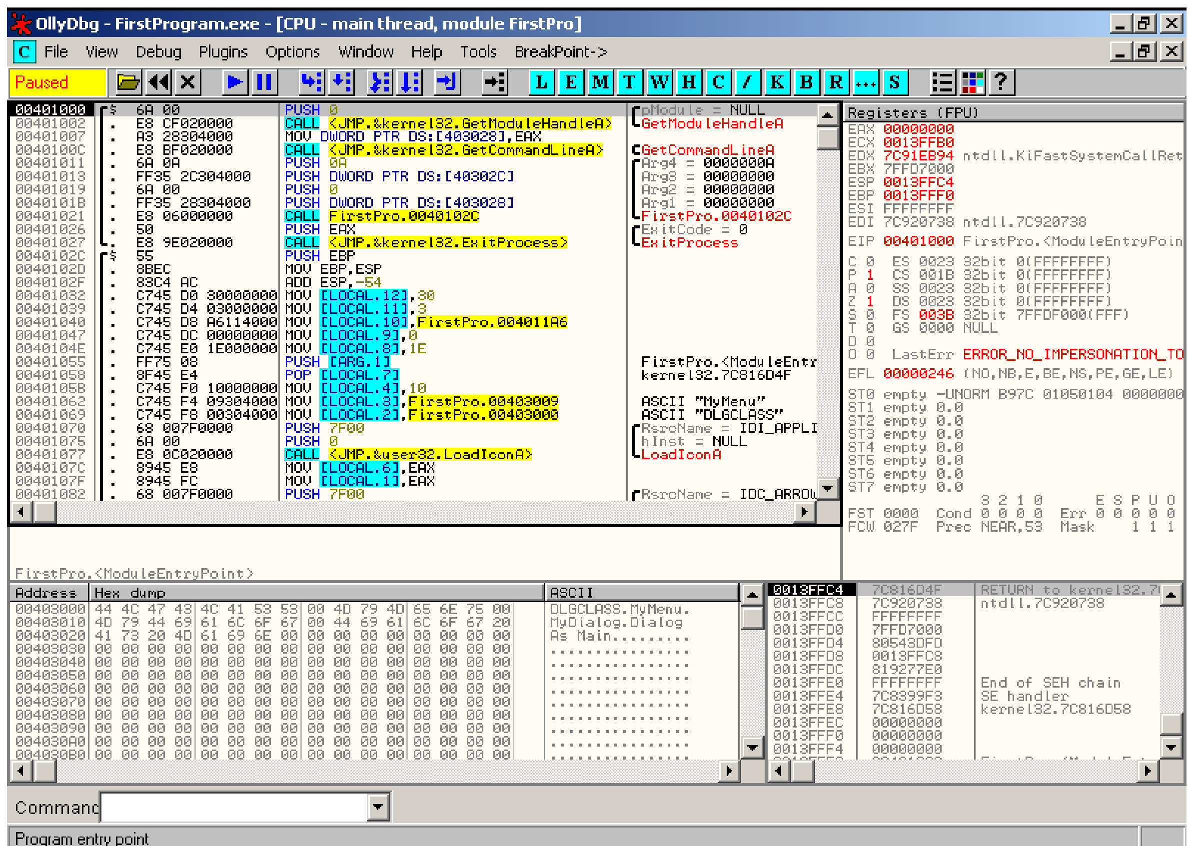
6.1 ¿QUÉ son los DLL?

Los DLL (Dynamic Link Libraries) son colecciones de funciones, por lo general proporcionados por Windows, que contienen las funciones que se utilizan de forma recurrente en los programas de Windows. Estas funciones hacen que sea más fácil para los programadores realizar lo que de otro modo serían, tediosas tareas repetitivas.

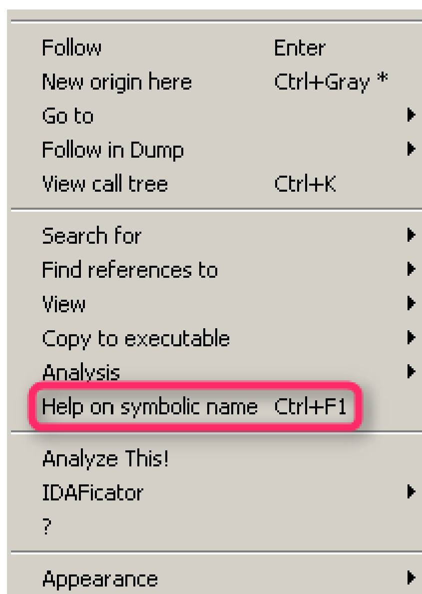
6.2 ¿CÓMO se usan los DLL?

La primera vez que ejecutamos una aplicación, el cargador de Windows comprueba una sección especial del PE Header para ver qué funciones son llamadas por la aplicación y cuál es su lugar dentro de los DLL. Después de cargar la aplicación en la memoria, itera a través de estos archivos DLL y carga cada uno en el espacio de la memoria. A continuación, pasará a través de todo el código de la aplicación inyectando la dirección correcta de las funciones DLL. Por ejemplo, si una de las primeras llamadas es convertir un buffer de letras a mayúsculas, llamando StrToUpper en la DLL kernel32, el cargador encontrará el lugar donde la DLL kernel32 está cargada, buscará la dirección de la función StrToUpper, e inyectará esa dirección en las líneas de código de la aplicación que llamó a esa función. La aplicación llamará a continuación al espacio en la memoria donde reside la DLL kernel32 para ejecutar la función StrToUpper, y luego regresará de nuevo al programa.

Veámoslo en un ejemplo práctico. Cargamos el programa FirstProgram.exe en Olly. La aplicación se detendrá en la primera línea de código, el Entry Point.



En la segunda línea del código, aparece una llamada a la función kernel32.GetModuleHandleA. En primer lugar, veamos el significado de esta función. Hacemos clic con el botón derecho sobre GetModuleHandleA y seleccionamos "Help on Symbolic Name". Olly abre una hoja de ayuda sobre esta API:



Win32 Programmer's Reference

Archivo Edición Marcador Opciones Ayuda

Contenido Índice Atrás << >>

GetModuleHandle

Quick Info Overview Group

The **GetModuleHandle** function returns a module handle for the specified module if the file has been mapped into the address space of the calling process.

```
HMODULE GetModuleHandle(
  LPCTSTR lpModuleName     // address of module name to return handle for
 );
```

Parameters

lpModuleName

Points to a null-terminated string that names a Win32 module (either a .DLL or .EXE file). If the filename extension is omitted, the default library extension .DLL is appended. The filename string can include a trailing point character (.) to indicate that the module name has no extension. The string does not have to specify a path. The name is compared (case independently) to the names of modules currently mapped into the address space of the calling process.

If this parameter is NULL, **GetModuleHandle** returns a handle of the file used to create the calling process.

Return Values

If the function succeeds, the return value is a handle to the specified module.

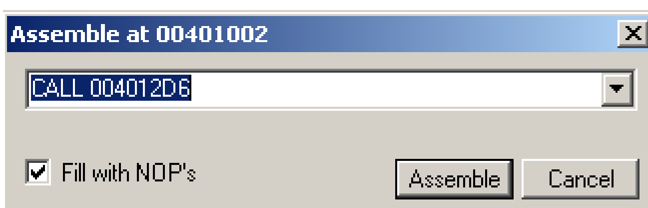
If the function fails, the return value is NULL. To get extended error information, call GetLastError .

Remarks

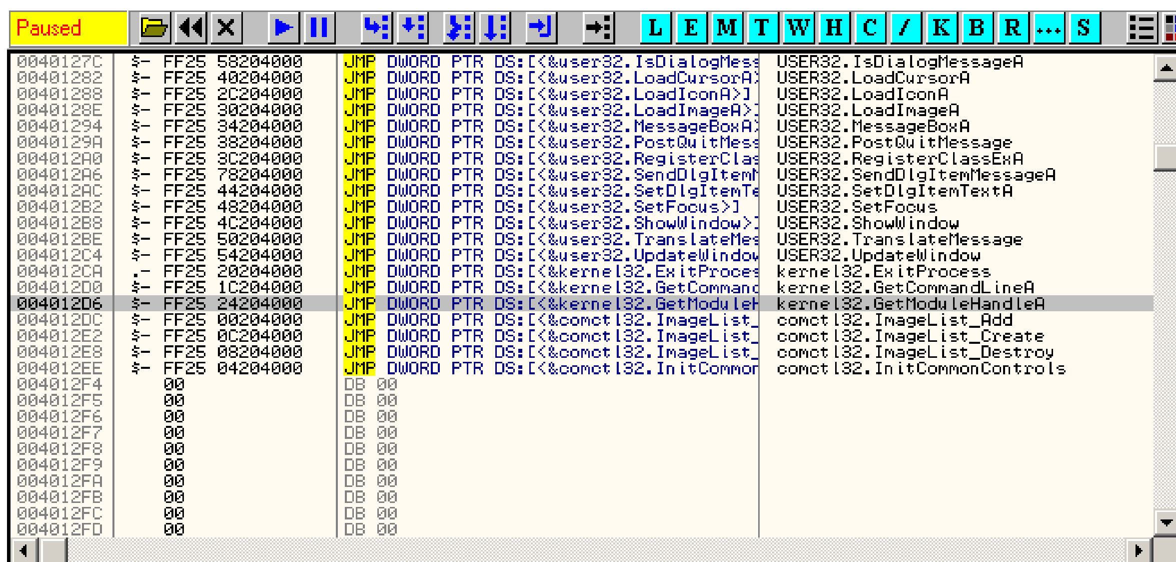
The returned handle is not global, inheritable, or duplicative, and it cannot be used by another process.

Vemos que esta función recibirá un identificador de la ventana que vamos a modificar. En Windows, si deseamos manipular una ventana debemos obtener un identificador que ayude a encontrar el objeto al que nos estamos refiriendo.

Cerramos la ventana de ayuda y vamos a ver exactamente hacia donde apunta esta llamada. Para ver la dirección de la función GetModuleHandleA, hacemos clic en la línea de CALL GetModuleHandleA y pulsamos la barra espaciadora. Se abrirá la siguiente ventana de ensamblaje:



Esta ventana tiene dos propósitos: en primer lugar muestra las instrucciones exactas del lenguaje ensamblador y en segundo lugar nos permite editar nuevas instrucciones. Existen dos formas para saltar a la dirección 4012D6 (sin llegar a ejecutar el código). Podemos resaltar la línea "CALL GetModuleHandleA " y pulsa "Enter" o pulsar Ctrl+G e introducir la dirección manualmente:



6.3 La tabla de saltos

Lo primero que debemos saber es que los DLL no siempre se cargan en el mismo lugar dentro de la memoria. El cargador de Windows puede cambiar el lugar de los archivos DLL. La razón se debe a que una de las primeras DLL que se cargan en Windows tiene asignada la dirección 80000000.

Si nuestra aplicación incluye un archivo DLL que va a ser cargado en esa misma dirección, el cargador debe mover uno de estos archivos DLL a otra dirección. Esto sucede todo el tiempo, y se llama reubicación.

La primera vez que escribimos el código para una aplicación y añadimos una instrucción que llama GetModuleHandleA, el compilador va a saber exactamente dónde está el DLL adecuado, por lo que asigna una dirección a esa instrucción, algo así como "CALL 800000000". Después, cuando el programa se cargue en memoria, todavía sigue teniendo este CALL a 800000000, pero ¿qué sucedería si el cargador haya decidido trasladar esa DLL a 80000E300? ¡La instrucción CALL llamaría a la función equivocada!

La forma en que el archivo PE, y por lo tanto el archivo de Windows, evita este problema es mediante la creación de una tabla de saltos. Es decir, cuando se compila por primera vez el código, cada llamada a GetModuleHandleA va a señalar una ubicación única en la aplicación, y esta ubicación única inmediatamente saltará a una dirección arbitraria (lo que eventualmente se convertirá en la dirección correcta). De hecho, todas las llamadas a funciones DLL utilizan esta misma técnica; llaman a una dirección específica que inmediatamente saltará a una dirección arbitraria. Cuando el cargador carga todos los archivos DLL, pasará a través de esta "tabla de saltos" y reemplazará todas las direcciones arbitrarias por las direcciones reales que la función tiene en la memoria.

Volviendo a nuestra aplicación FisrtProgram.exe, cuando se escribió el código para dicho programa, todas las funciones que vemos en la tabla de saltos han sido llamadas por varios DLL, pero el compilador no sabe dónde situarlos en la memoria una vez que el programa se ejecuta. Por lo tanto se podría haber creado una tabla como la siguiente:

40124C JMP XXXXX // gdi32.DeleteObject

401252 JMP XXXXX // user32.CreateDialogParamA

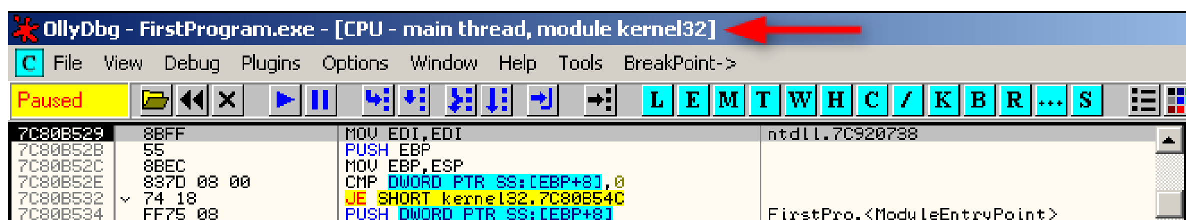
401258 JMP XXXXX // user32.DefWindowProcA

40125E JMP XXXXX // user32.DestroyWindow

…

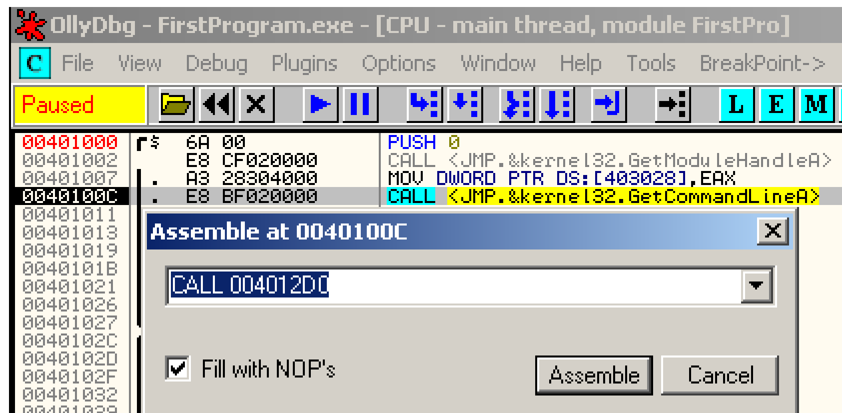
Una vez que el cargador haya cargado nuestra aplicación junto con los DLL, y encontrado las direcciones de esas funciones, recorrerá todas ellas, una por una para reemplazarlas por la dirección real en la que residen ahora esas funciones. Si no se hubiese hecho de esta forma, el cargador se vería forzado a recorrer toda nuestra aplicación y reemplazar **todos** los CALL a **todas** las funciones en **todos** los DLL con la dirección real. De esta otra forma el cargador solo reemplazará la dirección en un sitio por cada función CALL, es decir aquellas líneas de funciones en nuestra tabla de saltos.

Para ver esto mejor reiniciamos la aplicación en Olly y pulsamos F7. Llegados a la instrucción 401002 pulsamos la barra espaciadora y vemos que la dirección apunta a 4012D6 (ver apartado 12.2). Ahora pulsamos F7 para entrar dentro del CALL y veremos cómo nos encontramos en la dirección 4012D6. Si ahora volvemos a pulsar F7 llegaremos a la dirección **real** de GetlModuleHandleA, en nuestro caso 7C80B529. Existen dos formas de ver que nos encontramos en el módulo kernel32; la primera es fijándonos en el título de la ventana de la CPU de Olly:

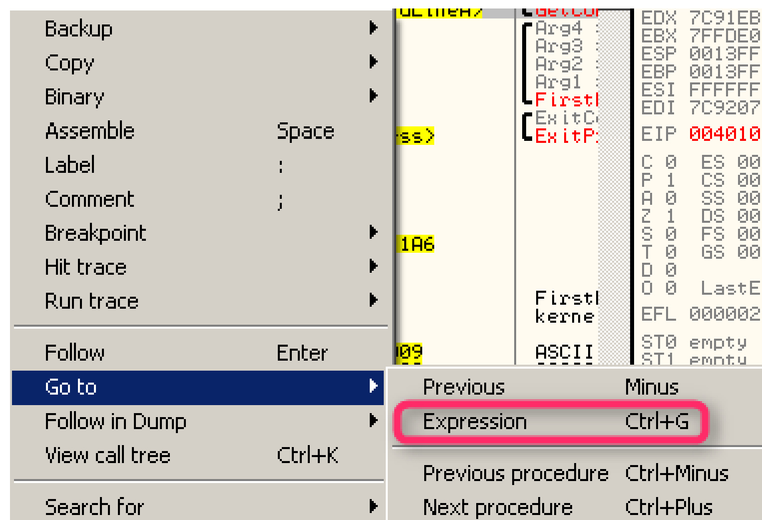


La segunda forma es abriendo la ventana de la memoria y buscando la dirección.

Continuemos estudiando más funciones. Reiniciamos la aplicación y pulsamos F8 hasta llegar a la dirección 40100C. En esta línea vemos un CALL a la función GetCommandLineA. Hacemos clic en la línea y pulsamos sobre la barra espaciadora para ver a qué dirección apunta.



Vemos que apunta a la dirección 4012D0. A continuación utilizaremos el método manual para acceder a dicha dirección. Para ello pulsamos Ctrl+G, o haciendo clic con el botón derecho seleccionamos Go to -> Expression.



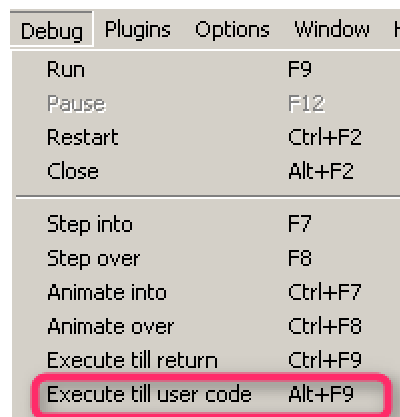
Hacemos clic en OK y saltaremos a la tabla de saltos para la función GetCommandLineA.

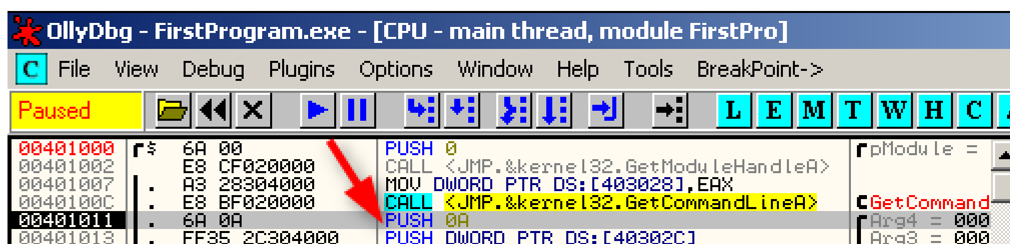
Pulsamos F7 para tomar el salto hacia el principio de la función GetCommanLineA del kernel32.dll. Vemos que la función empieza en 7C812C8D:

6.4 Saltar dentro y fuera de los DLL's

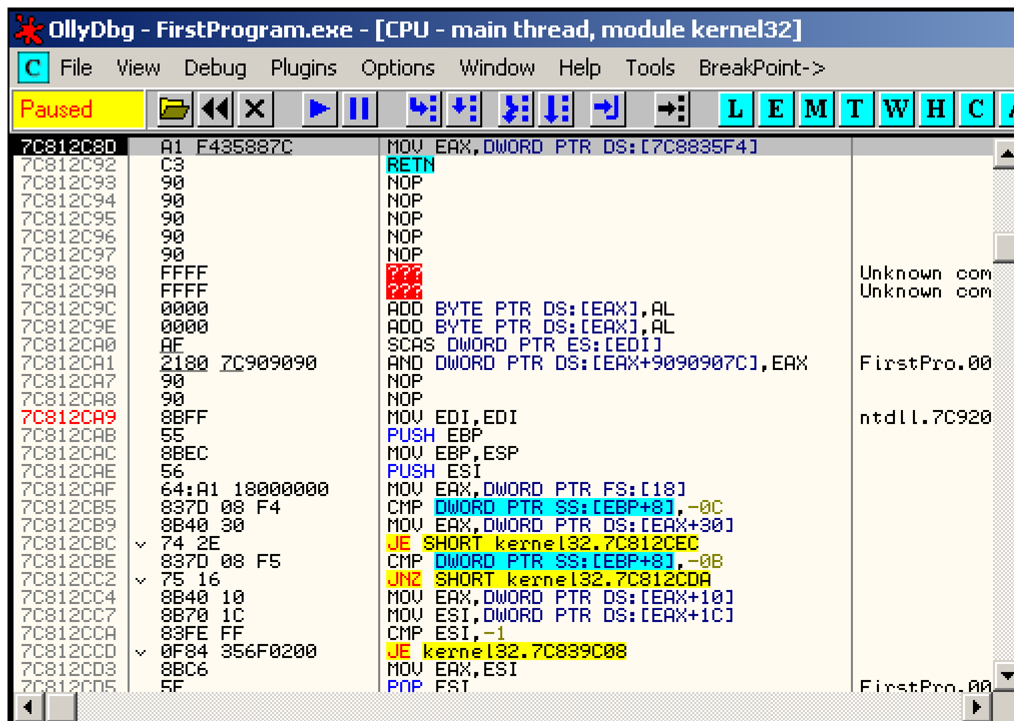
A medida que analizamos un programa, no serán pocas las veces en los que tengamos que confrontarnos con los archivos DLL. Sin embargo, a la hora de superar algún tipo de protección, estos archivos DLL de Windows no nos van aportar gran cosa. La única salvedad lo constituyen aquellos programas que vienen con sus propios archivos DLL (o en los que el esquema de protección está dentro de en una DLL). Hay un par de maneras de regresar a nuestro código a partir de un archivo DLL. Una forma es simplemente pasar a través de la totalidad del código de una función perteneciente a un DLL, para finalmente regresar al programa. Esta opción es poco recomendable debido a la excesiva cantidad de tiempo que tendríamos que emplear para realizar dicha tarea. La segunda opción es ir al menú de Olly y seleccionar "Debug" -> "Execute till user code" o presionar Alt+F9. A través de esta opción ejecutaríamos el código dentro de los DLL hasta volver a nuestro propio programa. Esta opción sin embargo no funcionará en los casos en que el DLL tiene acceso a un buffer o variable que está en el espacio de trabajo de nuestro programa. Olly se detendría ahí, por lo que podríamos terminar llamando Alt+F9 varias veces antes de que finalmente regresáramos al código del programa.

Veamos un ejemplo en Olly. Partimos de la dirección 7C812C8D en la que nos hemos detenido en el paso anterior, y que supone el comienzo de la función GetCommandLineA. Presionando Alt+F9 nos sitúa en la siguiente instrucción del programa:





Intentemos ahora otra manera de regresar al código de nuestro programa. Reiniciamos el programa, pulsamos F8 hasta llegar al CALL GetCommandLineA (40100C), pulsamos F7 para entrar en el CALL y volvemos a pulsar F7 para saltar dentro de la tabla de saltos. Volvemos a estar al comienzo de GetCommandLineA:

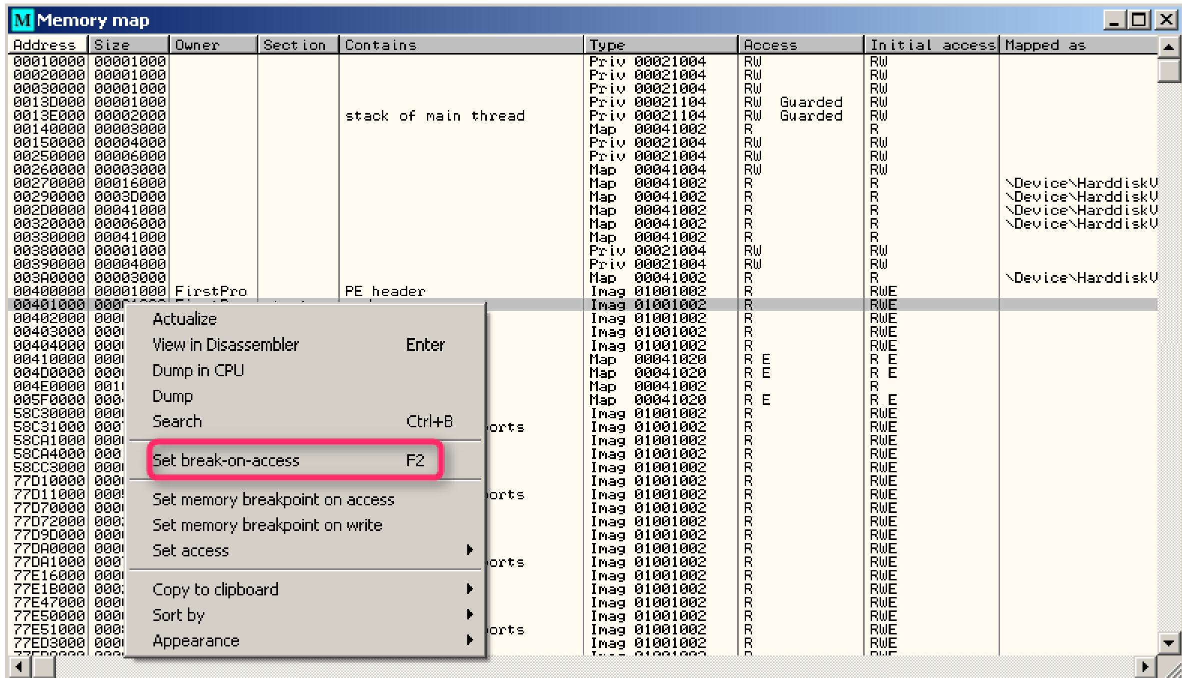


A continuación vamos abrir la ventana de la memoria y buscamos la sección que contiene el código del programa (a partir de la dirección 400000 con el PE Header).

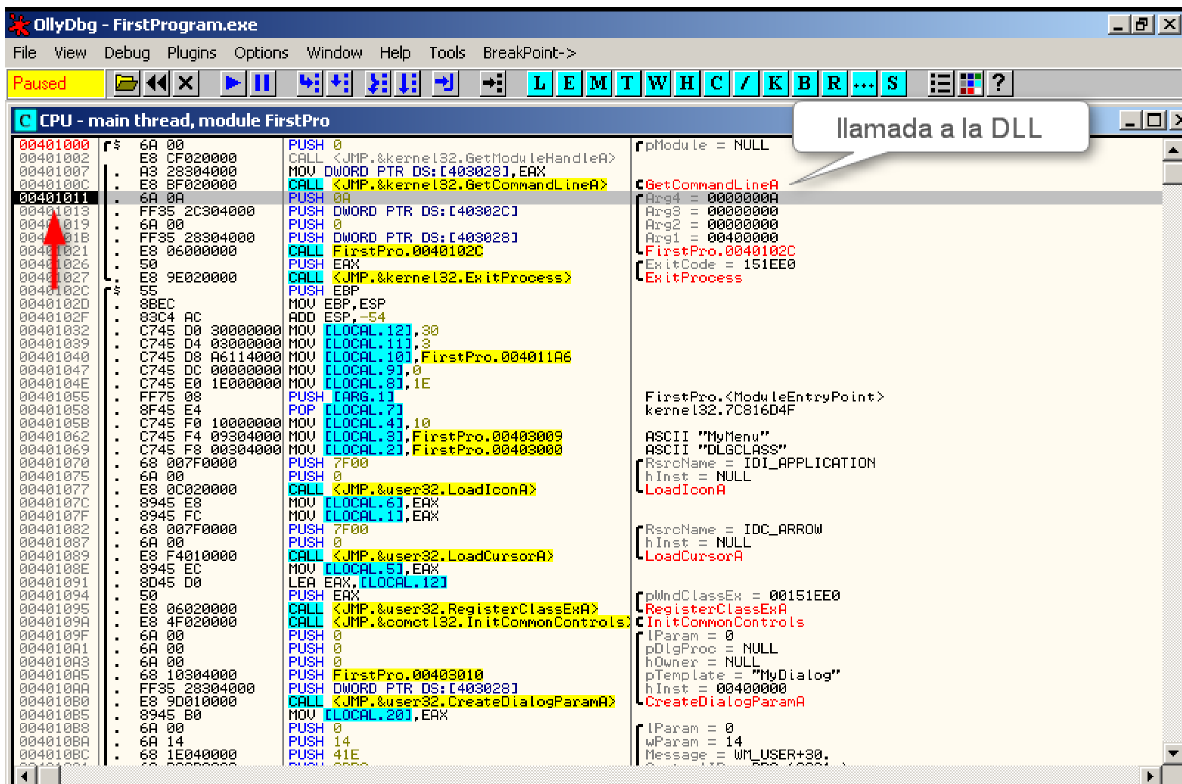
Memory map

Address	Size	Owner	Section	Contains	Type	Access	Initial access	Mapped as
00010000	00001000				Priv 00021004	RW	RW	
00020000	00001000				Priv 00021004	RW	RW	
00030000	00001000				Priv 00021004	RW	RW	
0013D000	00001000				Priv 00021104	RW Guarded	RW	
0013E000	00002000			stack of main thread	Priv 00021104	RW Guarded	RW	
00140000	00003000				Map 00041002	R	R	
00150000	00004000				Priv 00021004	RW	RW	
00250000	00006000				Priv 00021004	RW	RW	
00260000	00003000				Map 00041004	RW	RW	
00270000	00016000				Map 00041002	R	R	\Device\HarddiskV
00290000	0003D000				Map 00041002	R	R	\Device\HarddiskV
002D0000	00041000				Map 00041002	R	R	\Device\HarddiskV
00320000	00006000				Map 00041002	R	R	\Device\HarddiskV
00330000	00041000				Map 00041002	R	R	
00380000	00001000				Priv 00021004	RW	RW	
00390000	00004000				Priv 00021004	RW	RW	
003A0000	00003000				Map 00041002	R	R	\Device\HarddiskV
00400000	00001000	FirstPro		PE header	Imag 01001002	R	RWE	
00401000	00001000	FirstPro	.text	code	Imag 01001002	R	RWE	
00402000	00001000	FirstPro	.rdata	imports	Imag 01001002	R	RWE	
00403000	00001000	FirstPro	.data	data	Imag 01001002	R	RWE	
00404000	00001000	FirstPro	.rsrc	resources	Imag 01001002	R	RWE	
00410000	00002000				Map 00041020	R E	R E	
004D0000	00002000				Map 00041020	R E	R E	
004E0000	00103000				Map 00041002	R	R	
005F0000	00042000				Map 00041020	R E	R E	
58C30000	00001000	comctl32		PE header	Imag 01001002	R	RWE	
58C31000	00070000	comctl32	.text	code,imports,exports	Imag 01001002	R	RWE	
58CA1000	00003000	comctl32	.data	data	Imag 01001002	R	RWE	
58CA4000	0001F000	comctl32	.rsrc	resources	Imag 01001002	R	RWE	
58CC3000	00004000	comctl32	.reloc	relocations	Imag 01001002	R	RWE	
77D10000	00001000	USER32		PE header	Imag 01001002	R	RWE	
77D11000	0005F000	USER32	.text	code,imports,exports	Imag 01001002	R	RWE	
77D70000	00002000	USER32	.data	data	Imag 01001002	R	RWE	
77D72000	0002B000	USER32	.rsrc	resources	Imag 01001002	R	RWE	
77D9D000	00003000	USER32	.reloc	relocations	Imag 01001002	R	RWE	
77DA0000	00001000	ADVAPI32		PE header	Imag 01001002	R	RWE	
77DA1000	00075000	ADVAPI32	.text	code,imports,exports	Imag 01001002	R	RWE	
77E16000	00005000	ADVAPI32	.data	data	Imag 01001002	R	RWE	
77E1B000	0002C000	ADVAPI32	.rsrc	resources	Imag 01001002	R	RWE	
77E47000	00005000	ADVAPI32	.reloc	relocations	Imag 01001002	R	RWE	
77E50000	00001000	RPCRT4		PE header	Imag 01001002	R	RWE	
77E51000	00082000	RPCRT4	.text	code,imports,exports	Imag 01001002	R	RWE	
77ED3000	00007000	RPCRT4	.orpc	code	Imag 01001002	R	RWE	

Hacemos clic en la dirección 401000, que es la línea que contiene la sección .text. Pulsamos F2 para poner un Breakpoint on access en esta sección de la memoria.



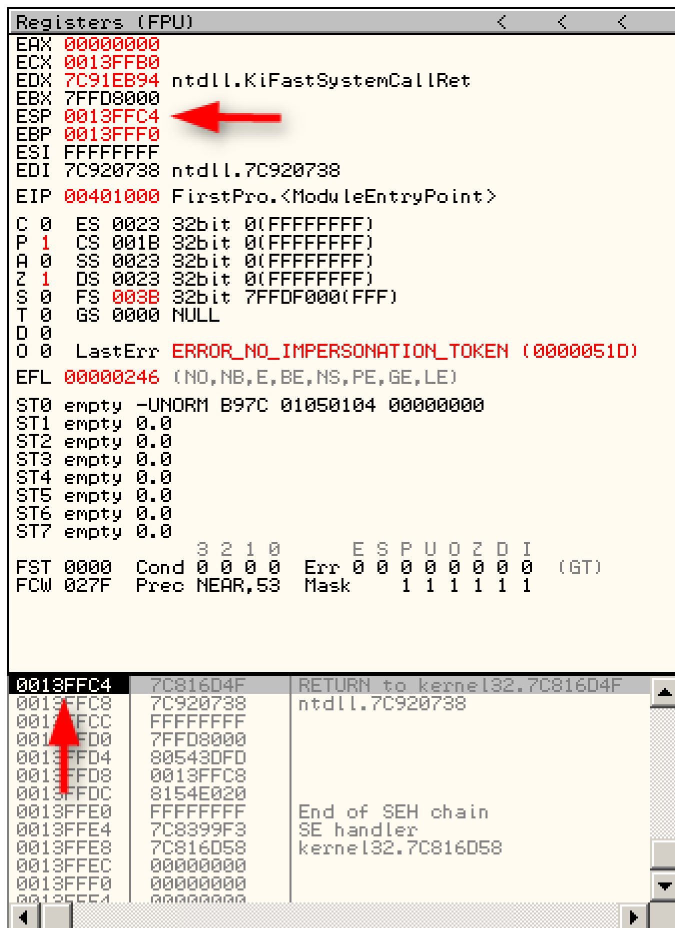
Pulsamos F9 para ejecutar el programa, y vemos que Olly se detiene en 401011, una línea despúes de nuestro CALL a la DLL !



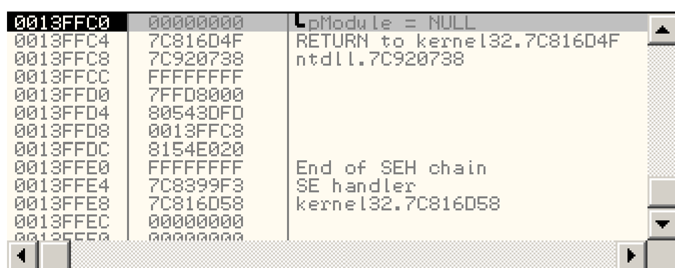
6.5 La Pila (II)

La pila es una parte muy importante en ingeniería inversa, y sin una comprensión exhaustiva de la misma no lograremos alcanzar nuestros objetivos.

Comenzaremos por visualizar la ventana de registros, (después de reiniciar la aplicación) y especialmente el registro ESP. Este registro apunta a la dirección situada en la parte superior de la pila. En nuestro caso ESP es 13FFC4, valor que coincide con la dirección que muestra la pila en primer lugar:



Pulsamos F8 u F7 para colocar el valor cero en la pila.



Si nos fijamos en la ventana de los registros vemos que ESP ha cambiado a 13FFC0, porque después de colocar un byte en la pila, esta es ahora la nueva cabecera de la pila.

Pulsamos F8, saltamos por encima del CALL GetModuleHandleA, y volvemos a la ventana de la pila:

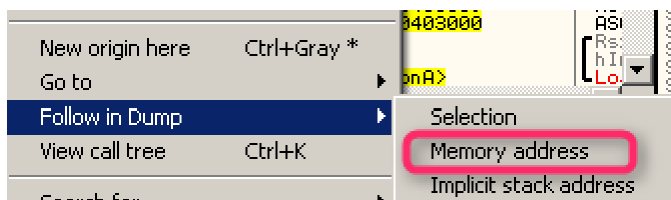
Vemos que la pila se ha incrementado en uno hacia abajo. La razón de ello es la función GetModuleHandleA, que utilizó como argumento el cero que fue empujado en la pila y luego lo quitó de la pila, ya que no era necesario. Como se vio en el capítulo anterior, esta es una manera de pasar argumentos a las funciones: son empujados a la pila, la función CALL los saca de la pila, los usa, y luego regresa, por lo general con alguna información en los registros.

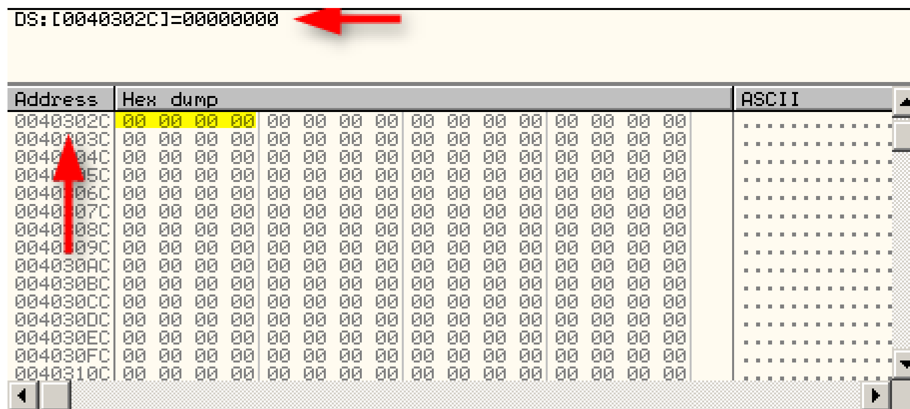
Si presionamos ahora F8 dos veces para pasar por encima de la llamada a GetCommandLineA observaremos que la pila no ha cambiado. Eso es porque no empujamos en la pila algo que pudiese ser utilizado por esa función. A continuación, se llega a una instrucción PUSH 0A. Este es el primer argumento que vamos a pasar a la siguiente función CALL. Pulsamos F8 y vemos que 0A se ha situado en la parte superior de la pila y el registro ESP se ha reducido en 2 bytes (cuando se empuja un valor en la pila, el registro ESP disminuye a medida que la pila 'crece' hacia abajo en la memoria). Pulsamos F8 de nuevo, y el registro ESP baja de nuevo 4 bytes, ya que hemos empujado un valor de 4 bytes en la pila. Si nos fijamos en la parte superior de la pila vemos que hemos empujado 00000000 en la pila. Veamos de donde sale ese valor.

La instrucción en 401013 es:

PUSH DWORD PTR DS:[40302C]

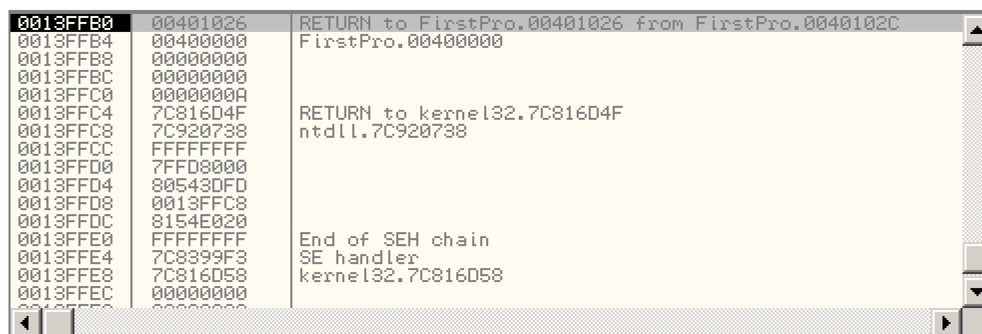
Lo que viene a decir esta instrucción es que tomemos cuatro bytes del contenido en la dirección 40302C para empujarlos en la pila. ¿Que hay en 40302C? Hacemos clic con el botón derecho sobre la instrucción en la dirección 401013 y seleccionamos "Follow in Dump" -> "Memory Address". Esto cargará la ventana dump con el contenido de la memoria a partir de 40302C:





Si presionemos F8 una vez más nos encontraremos con otro PUSH pero esta vez en la dirección 403028. Si nos desplazamos hacia arriba en la ventana dump se puede ver que en esta dirección hay todavía más ceros. Así que lo que esta sección está haciendo es empujar punteros a direcciones de memoria, actualmente fijados en cero, y que nuestro código usará como variables. Pasamos (F8) sobre el último PUSH y entramos (F7) en el CALL que apunta a la dirección 40102C. Podemos observar que algo nuevo ha sido empujado en la pila. En nuestro caso es la dirección de regreso correspondiente a nuestra llamada, 401026.

Cuando cualquier código utiliza una instrucción CALL, la dirección de la siguiente instrucción que se ejecutaría si no hubiésemos hecho la llamada es empujada automáticamente en la pila. Después de hacer todo lo que tenga que hacer la función llamada, ha de saber dónde regresar. Esta dirección que es empujada automáticamente en la pila es la dirección de regreso. Podemos verlo en la parte superior de la ventana de pila:



Vemos que Olly descubrió que se trata de una dirección de regreso y que apunta de vuelta a nuestro programa FirstPro. La dirección de regreso es 40102C.

Ahora, al final de esta función, se utilizará una instrucción RETN "Retorno". Esta instrucción de retorno lo que realmente significa es: saca (POP) la dirección que se encuentra en la parte superior de la pila y señala hacia esa dirección el código que estamos ejecutando (básicamente sustituye el registro EIP -el registro de la actual línea de código que estamos ejecutando- con el valor que ha sido sacado de la pila). Así que ahora, la función que ha sido llamada sabe exactamente donde tiene que regresar cuando haya acabado! De hecho, si nos desplazamos un poco hacia abajo veremos la instrucción RETN en la dirección 4011A3, sacando la dirección de la pila para seguir la ejecución de código a partir de esa dirección:

El diez significa dame la dirección de retorno y borra 10 bytes de la pila.

6.6 Mostrando argumentos y variables locales

Haciendo doble clic en el registro EIP regresaremos a la línea de código donde paramos la ejecución, en la dirección 40102C. Un poco más abajo podemos apreciar varias etiquetas azules que dicen LOCAL y una que dice ARG:

Un argumento, como hemos comentado anteriormente, son variables que se pasan a una función y posteriormente puede que a la pila. Una variable local es una variable que la función CALL 'crea' para sostener algo de forma temporal. He aquí un ejemplo de un pequeño programa que contiene los dos conceptos:

```
main()

{

        sayHello( "Manuel");

}

sayHello( String name)

{

        int numTimes  = 3;

        String hello = "Hello, ";



        for (int x = 0; x < numTimes; x++)

                print (hello + name);



        }
```

En este programa, la cadena "Manuel" es un argumento pasado a la función sayHello. En lenguaje desensamblado, esta cadena (o al menos la dirección de esta cadena) sería empujado en la pila de manera que la función sayHello pueda hacer referencia a ella. Una vez que el control se transfiere a la función sayHello, esta necesita establecer un par de variables locales, es decir, variables que se utilizarán por la función, pero que no serán necesarias una vez la función haya finalizado. Los ejemplos de variables locales son los numTimes enteros, la cadena hola, y el número entero x. Tanto los argumentos como las variables locales son almacenados en la pila con la ayuda del registro ESP que normalmente apunta a la parte superior de la pila.

Imaginemos que entramos en la función sayHello y la pila tiene los siguientes datos sobre ella:

1. La dirección de la cadena ‘manuel’
2. La dirección de regreso del CALL.

Si queremos crear una variable local, todo lo que tenemos que hacer es restar una cantidad ‘x’ del registro ESP lo que creará un espacio en la pila! Digamos que restamos 4 bytes de ESP. La pila tendría entonces el siguiente aspecto:

1. Un número vacío de 32-bit
2. La dirección de la cadena ‘manuel’
3. La dirección de regreso del CALL.

Ahora podríamos poner todo lo que quisiéramos en esta dirección, por ejemplo, podríamos pasar la variable numTimes a nuestra función sayHello. Ya que nuestra función utiliza tres variables (todos de 32 bits de longitud) realmente restaría 12 bytes (o 0xC en

hexadecimal) de ESP y entonces tendríamos tres variables locales que podríamos usar. La pila quedaría entonces de la siguiente manera:

1. Una dirección vacía de 32-bits que apunta a la cadena 'hello'
2. Un número vacío de 32-bits para la variable 'x'
3. Un número vacío de 32-bits para la variable 'numTimes'
4. La dirección de regreso del CALL.

Ahora, la función sayHello puede rellenar, cambiar y volver a utilizar estas direcciones para jugar con nuestras variables, y todavía tiene el argumento pasado en el primer lugar (la cadena "manuel"). Cuando sayHello termine, tiene dos maneras de eliminar estas variables y argumentos locales (ya que no serán necesarios después de que la función haya sido ejecutada) y restablecer la pila de nuevo a la forma en la que estaba: 1) puede cambiar el registro ESP de nuevo a su estado original o 2) utilizar una instrucción especial RETN con un número después de ella. En el primer caso, para que el programa pueda recordar cuál era el valor original de ESP, utilizará otro registro, EBP, cuyo propósito es hacer un seguimiento de la ubicación original la pila cuando entramos en la función sayHello por primera vez. Cuando esté listo para regresar, simplemente copia el valor original de EBP nuevamente a ESP y las variables desaparecen. La dirección de regreso está ahora en la parte superior de la pila, y cuando la instrucción RETN se ejecuta, utilizará este valor para volver a nuestro programa principal.

En el segundo caso, podemos decirle a la CPU la cantidad de bytes que no son necesarios en la pila para eliminarlos de la parte superior. En nuestro caso, usaríamos RETN 16 (0xF en hexadecimal) y esto desharía los primeros 16 bytes (o 4 números de 32 bits) de la parte superior de la pila, dejando la nueva parte superior de la pila con la dirección de retorno para volver a nuestro programa principal. El tipo de retorno utilizado generalmente depende del compilador.

Volviendo a nuestro programa FirstPro.exe podemos ver que Olly ha descifrado un argumento y 12 variables locales. Estas variables locales son usadas en nuestro programa para realizar un seguimiento de cosas como la dirección de nuestro icono, la dirección del buffer para nuestra cadena de texto, la longitud de la cadena de texto etc, Cuando termine, sacará estos valores de la pila, cambiará el registro ESP de nuevo a EBP o RETN con un número.

6.7 Ollydbg Cheatsheet

Program	
Program reset	Ctrl + F2
Close program	Alt + F2
Open EXE file	F3
Bring Olly to top	Alt + F5
Step into	F7
Animate into	Ctrl + F7
Step over	F8
Animate over	Ctrl + F8
Run program	F9
Execute till return	Ctrl + F9
Execute till user code	Alt + F9
Run trace into	Ctrl + F11
Stop execution	F12
Run trace over	Ctrl + F12
Stop anim. or tracing	Esc

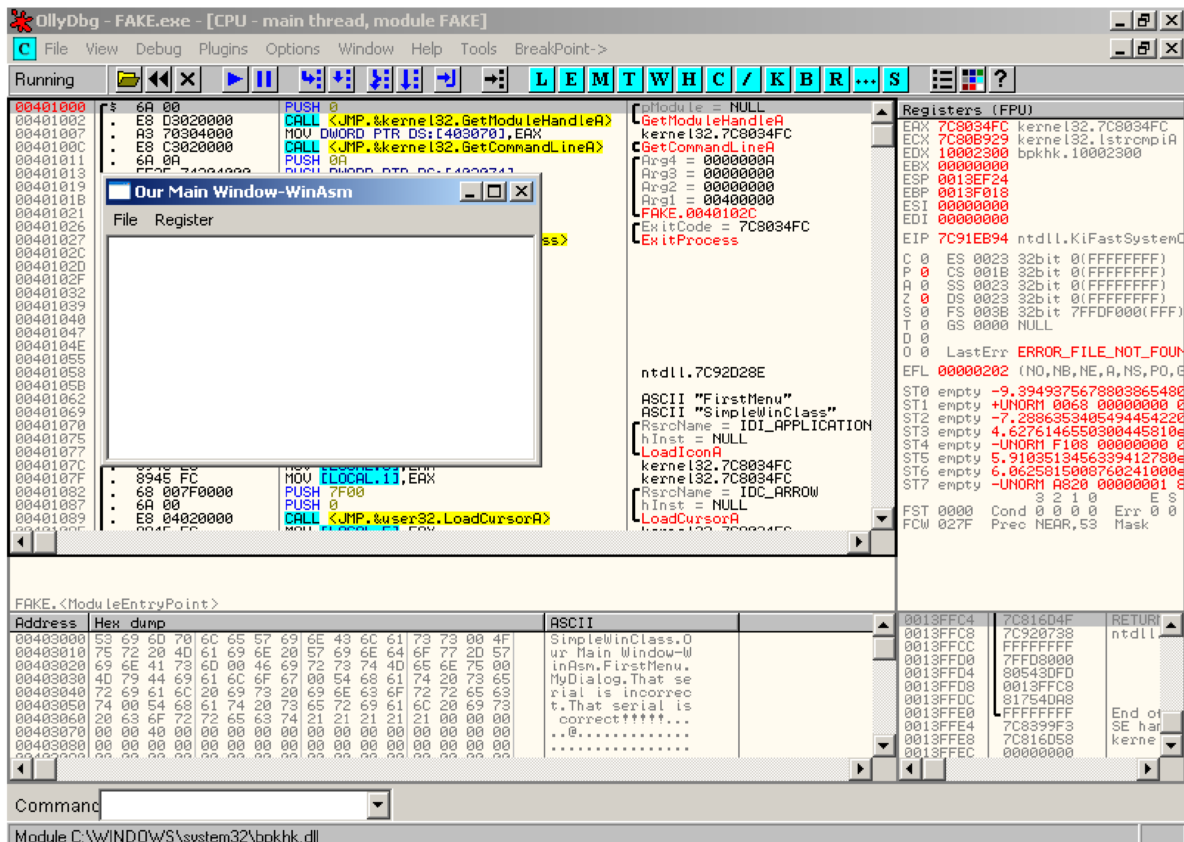
Windows	
Breakpoint window	Alt + B
CPU window	Alt + C
Modules window	Alt + E
Call stack window	Alt + K
Log window	Alt + L
Memory window	Alt + M
Patches window	Alt + P
Run Trace window	Alt + T
Quit Olly	Alt + X
Maximize window	F5

Capítulo 7

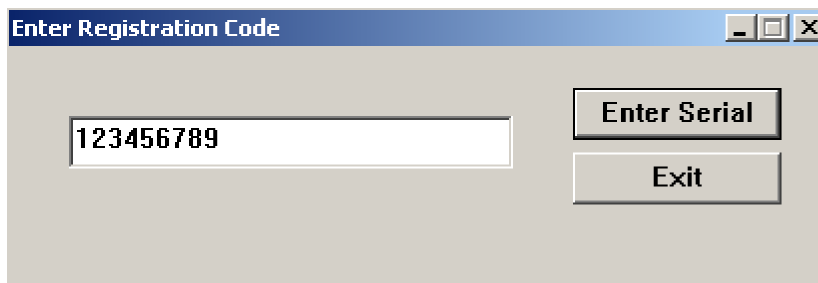
EJERCICIOS RESUELTOS

7.1 Caso práctico 1: buscando cadenas de texto

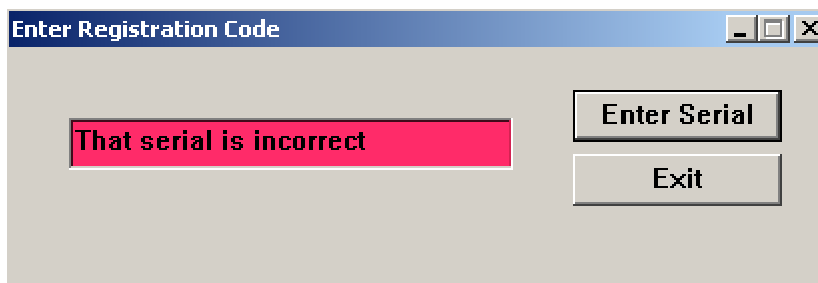
Cargamos fake.exe en Olly, y lo primero que haremos es estudiar el comportamiento del programa pulsando F9.



Vemos que aparece una ventana para registrarnos. Hacemos clic en “Register” e introducimos un serial cualquiera.



Pulsamos Enter Serial y obtenemos la siguiente respuesta:



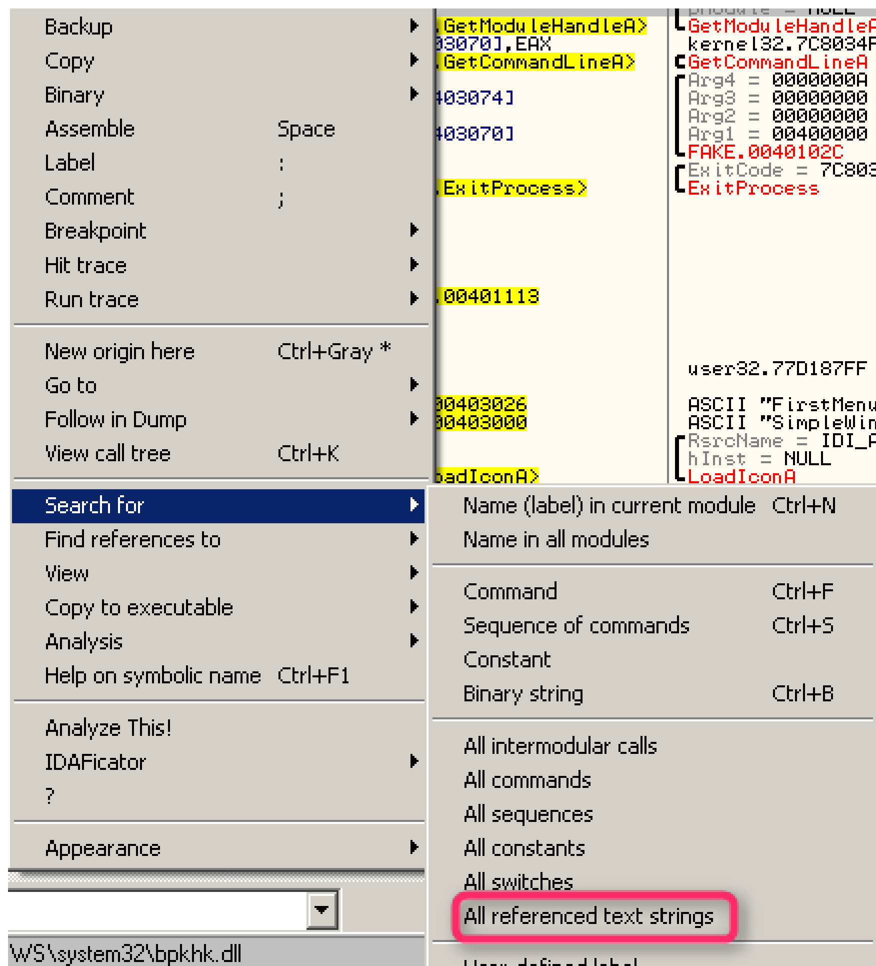
El primer método que utilizaremos para seguir la rutina de comprobar un serial se denomina:

Searching for all text strings

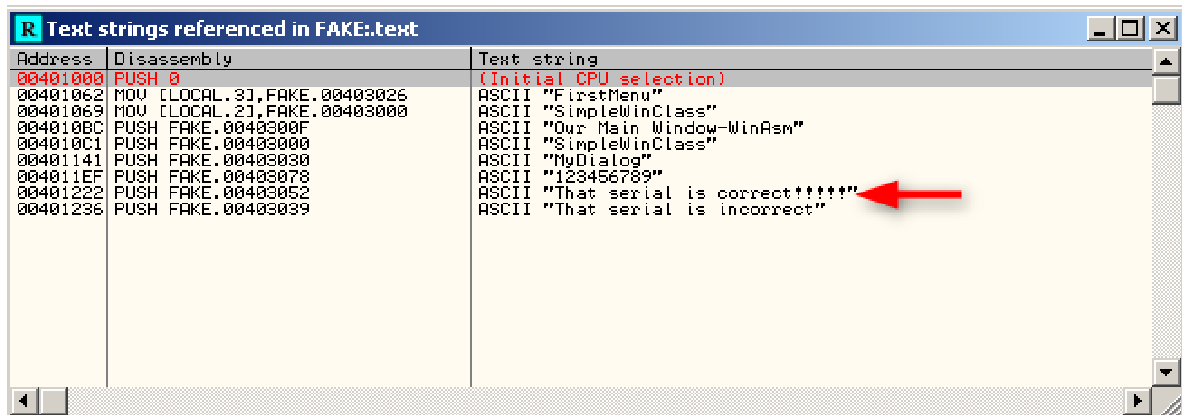
A través de este método Olly se verá invocado a buscar dentro del espacio de la memoria de nuestro programa cualquier cosa que se parezca a una cadena ASCII o Unicode.

Comprobar las cadenas de texto nos puede dar información valiosa sobre la estructura de un binario. De hecho podemos averiguar si el binario se ha empaquetado o protegido de alguna manera, o si se trata de un binario malicioso.

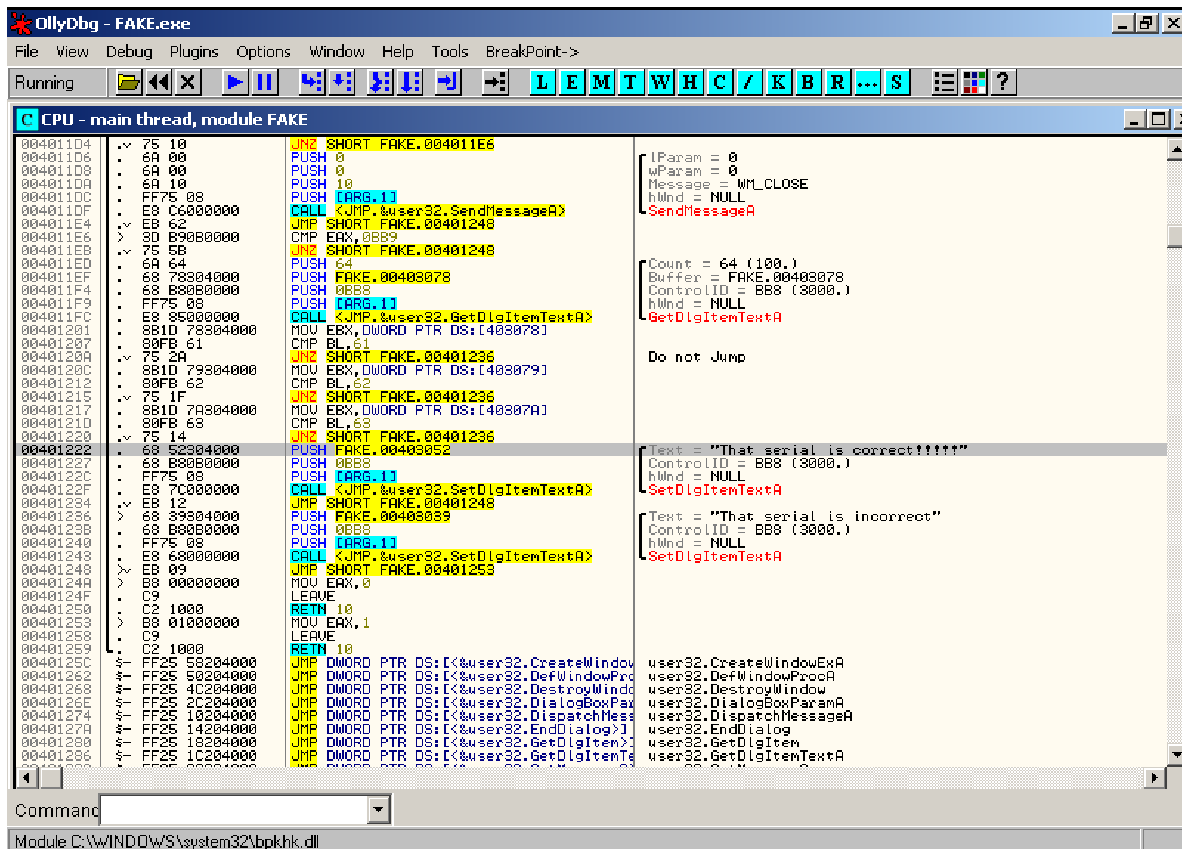
Hacemos clic con el botón derecho dentro de la ventana de desensamblado y seleccionamos “Search for” -> “All referenced text strings”.



Se abre una ventana con todas las cadenas de texto que Olly haya encontrado en la memoria del programa.



Hacemos doble clic en “That serial is correct” y Olly nos lleva al código desensamblado dentro de la ventana de desensamblaje.



Cabe mencionar en este punto que todo mensaje (bueno o malo), va estar precedido por una comparación, o una instrucción CALL cuyo resultado nos llevará a coger un salto hacia el mensaje bueno o malo.

En nuestro caso el primer salto es un JNZ en la dirección 401220.

```
00401220  .v 75 14        JNZ SHORT FAKE.00401236
00401222  .  68 52304000  PUSH FAKE.00403052                     ┌Text = "That serial is correct!!!!!"
00401227  .  68 B80B0000  PUSH 0BB8                              │ControlID = BB8 (3000.)
0040122C  .  FF75 08      PUSH [ARG.1]                           │hWnd = NULL
0040122F  .  E8 7C000000  CALL <JMP.&user32.SetDlgItemTextA>     └SetDlgItemTextA
00401234  .v EB 12        JMP SHORT FAKE.00401248
00401236  >  68 39304000  PUSH FAKE.00403039                     ┌Text = "That serial is incorrect"
0040123B  .  68 B80B0000  PUSH 0BB8                              │ControlID = BB8 (3000.)
00401240  .  FF75 08      PUSH [ARG.1]                           │hWnd = NULL
00401243  .  E8 68000000  CALL <JMP.&user32.SetDlgItemTextA>     └SetDlgItemTextA
```

Vemos que el salto nos lleva directamente al mensaje malo "That serial is incorrect". Fijémonos pues en la instrucción justamente anterior al salto. En este caso estamos ante una instrucción de comparación (CMP) cuyo resultado podría determinar si vamos a saltar hacia el mensaje bueno o malo.

En realidad hay tres combinaciones de CMP/JNZ, la primera en la dirección 40120A, la segunda en la dirección 401215 y la tercera, como acabamos de ver en la dirección 401220. Estos tres saltos nos van llevar directamente al mensaje malo (401236). Si queremos podemos anotar un comentario al lado de cada instrucción de salto.

```
MOV EBX,DWORD PTR DS:[403078]
CMP BL,61
JNZ SHORT FAKE.00401236                  No saltar
MOV EBX,DWORD PTR DS:[403079]
CMP BL,62
JNZ SHORT FAKE.00401236                  No saltar
MOV EBX,DWORD PTR DS:[40307A]
CMP BL,63
JNZ SHORT FAKE.00401236                  No saltar
PUSH FAKE.00403052                       ┌Text = "That serial is correct!!!!!"
PUSH 0BB8                                │ControlID = BB8 (3000.)
PUSH [ARG.1]                             │hWnd = NULL
CALL <JMP.&user32.SetDlgItemTextA>       └SetDlgItemTextA
JMP SHORT FAKE.00401248
PUSH FAKE.00403039                       ┌Text = "That serial is incorrect"
PUSH 0BB8                                │ControlID = BB8 (3000.)
PUSH [ARG.1]                             │hWnd = NULL
CALL <JMP.&user32.SetDlgItemTextA>       └SetDlgItemTextA
```

A continuación ponemos un Breakpoint en la dirección 401201, pulsamos F9 y nos vuelve a salir la ventanilla, introducimos un serial y le damos ok.

Olly se detendrá en nuestro Breakpoint.

```
00401201  .  8B1D 78304000  MOV EBX,DWORD PTR DS:[403078]
00401207  .  80FB 61        CMP BL,61
0040120A  .v 75 2A          JNZ SHORT FAKE.00401236              No saltar
0040120C  .  8B1D 79304000  MOV EBX,DWORD PTR DS:[403079]
00401212  .  80FB 62        CMP BL,62
00401215  .v 75 1F          JNZ SHORT FAKE.00401236              No saltar
00401217  .  8B1D 7A304000  MOV EBX,DWORD PTR DS:[40307A]
0040121D  .  80FB 63        CMP BL,63
00401220  .v 75 14          JNZ SHORT FAKE.00401236              No saltar
00401222  .  68 52304000    PUSH FAKE.00403052                   ┌Text = "That serial is correct!!!!!"
00401227  .  68 B80B0000    PUSH 0BB8                            │ControlID = BB8 (3000.)
0040122C  .  FF75 08        PUSH [ARG.1]                         │hWnd = 000900F8 ('Enter Registration Code',class='#32770')
0040122F  .  E8 7C000000    CALL <JMP.&user32.SetDlgItemTextA>   └SetDlgItemTextA
00401234  .v EB 12          JMP SHORT FAKE.00401248
00401236  >  68 39304000    PUSH FAKE.00403039                   ┌Text = "That serial is incorrect"
0040123B  .  68 B80B0000    PUSH 0BB8                            │ControlID = BB8 (3000.)
00401240  .  FF75 08        PUSH [ARG.1]                         │hWnd = 000900F8 ('Enter Registration Code',class='#32770')
00401243  .  E8 68000000    CALL <JMP.&user32.SetDlgItemTextA>   └SetDlgItemTextA
```

Analizemos esta instrucción: MOV EBX, DWORD PTR DS:[403078]

Para ver el contenido de la dirección 403078 en la memoria, hacemos clic con el botón derecho sobre la instrucción y seleccionamos "Follow in Dump" -> "Memory address"

En la ventana Dump podemos ver el resultado de nuestra busqueda:

```
Address   Hex dump                                          ASCII
00403078  31 32 33 34 35 36 37 38 39 00 00 00 00 00 00 00  123456789.......
00403088  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403098  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030A8  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030B8  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030C8  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030D8  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030E8  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030F8  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

¡En la dirección 403078 se encuentra el serial que hemos introducido!

De la instrucción anterior podemos entonces concluir que se han cargado los cuatro primero bytes de la dirección 403078 en EBX (31 32 33 34 en hexadecimal y 1234 en ASCII).

A continuación pulsamos F8 y comprobamos el valor de EBX:

```
Registers (FPU)
EAX 00000009
ECX 77D3F88F user32.77D3F88F
EDX 7C91EB94 ntdll.KiFastSystemCallRet
EBX 34333231
ESP 0013F9F8
EBP 0013F9F8
ESI 00401178 FAKE.00401178
EDI 0013FA60

EIP 00401207 FAKE.00401207

C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 1  FS 003B 32bit 7FFDF000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00000286 (NO,NB,NE,A,S,PE,L,LE)

ST0 empty -4.4112880553140798840e-1178
ST1 empty 0.0000000000000023290e-4933
ST2 empty +UNORM 1F80 000700DE 00000000
ST3 empty 6.3171253928737695410e-4932
ST4 empty -UNORM 8628 0000003B F0807C38
ST5 empty +UNORM 003B 0013FCD4 00000000
ST6 empty -UNORM FCCC 00000202 0000001B
ST7 empty 0.5263546642222927530e-4933
               3 2 1 0      E S P U O Z D I
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0 0  (GT)
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1
```

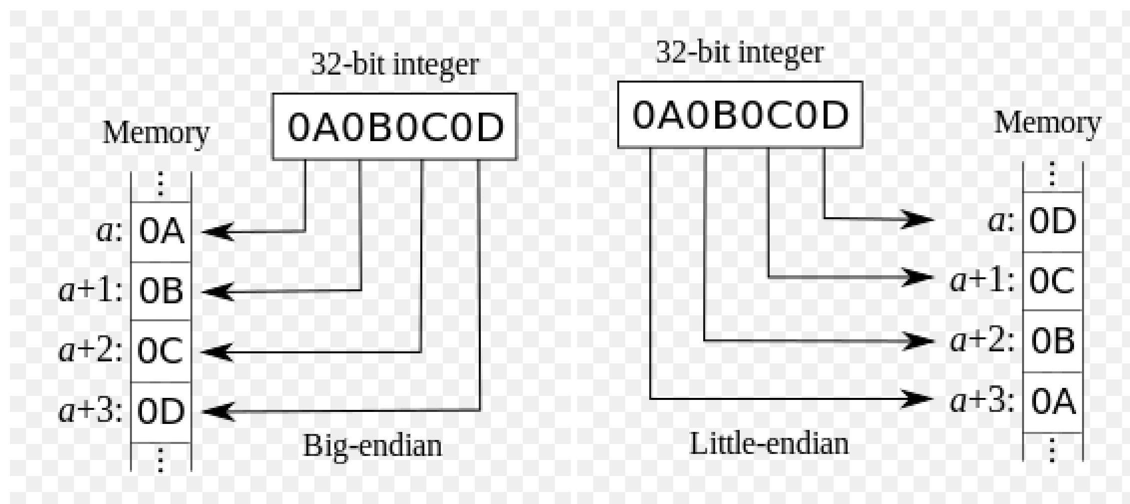
Para ver el valor de EBX en ASCII basta con hacer doble clic sobre el registro:

```
Modify EBX
Hexadecimal   34333231
Signed        875770417
Unsigned      875770417
Char   4   3   2   1
OK   Cancel
```

*Nota: El término inglés **endianness** ("extremidad") designa el formato en el que se almacenan los datos **de más de un** byte en un ordenador. El problema es similar a los idiomas en los que se escriben de derecha a izquierda, como el árabe, o el hebreo, frente a los que se escriben de izquierda a derecha, pero trasladado de la escritura al almacenamiento en memoria de los bytes.*

No se debe confundir trivialmente el orden de escritura textual en este artículo con el orden de escritura en memoria, por ello establecemos que lo que escribimos primero lleva índices de memoria más bajos, y lo que escribimos a continuación lleva índices más elevados, que lo que lleva índices bajos es previo en memoria, y así sucesivamente, siguiendo la ordenación natural de menor a mayor, por ejemplo la secuencia {0,1,2} indicaría, -algo más allá de la intuición- que 0 es previo y contiguo en el espacio de memoria a 1, etc.

*Usando este criterio el sistema **big-endian** adoptado por Motorola entre otros, consiste en representar los bytes en el orden "natural": así el valor hexadecimal 0x4A3B2C1D se codificaría en memoria en la secuencia {4A, 3B, 2C, 1D}. En el sistema **little-endian** adoptado por Intel, entre otros, el mismo valor se codificaría como {1D, 2C, 3B, 4A}, de manera que de este modo se hace más intuitivo el acceso a datos, porque se efectúa fácilmente de manera incremental de menos relevante a más relevante (siempre se opera con incrementos de contador en la memoria), en un paralelismo a "lo importante no es como empiezan las cosas, sino como acaban."*



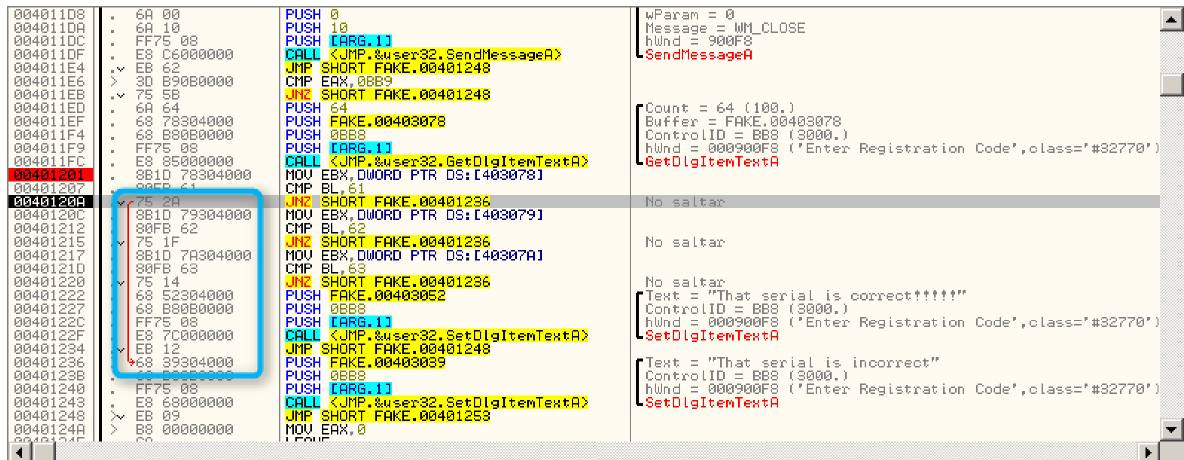
https://es.wikipedia.org/wiki/Endianness

De vuelta en la ventana del registro EBX, podemos observar que la representación hexadecimal está en Little-endian.

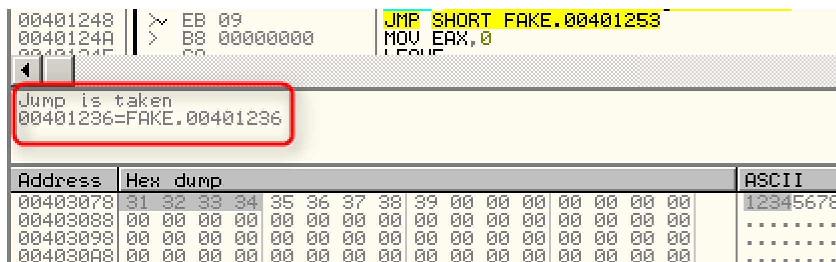
En la siguiente instrucción comparamos BL que es el primer byte del registro EBX con el valor hexadecimal 61. Si el contenido de BL no es igual a 61, saltaremos al mensaje malo.

Volviendo al registro EBX podemos observar que el último byte (BL) no es 61 sino 31, así que saltaremos al mensaje malo.

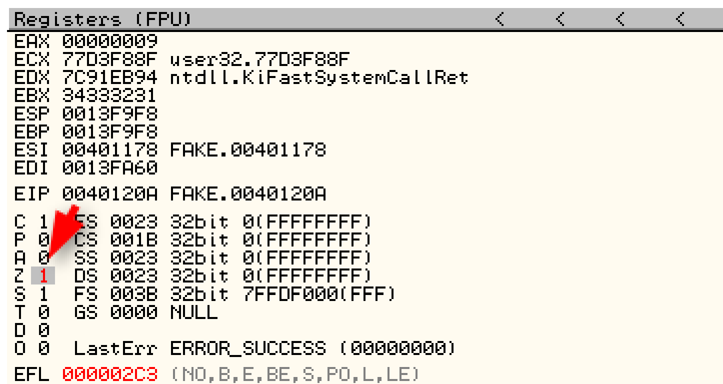
Pulsamos F8 y nos situamos en la instrucción JNZ SHORT FAKE.401236. Vemos que aparece una flecha roja en la columna de los opcodes. Esta flecha nos indica dos cosas: 1) si es roja (como en este caso), entonces se va a tomar el salto (si la flecha fuese gris, **no** tomaríamos el salto) y 2) nos muestra el destino de dicho salto.



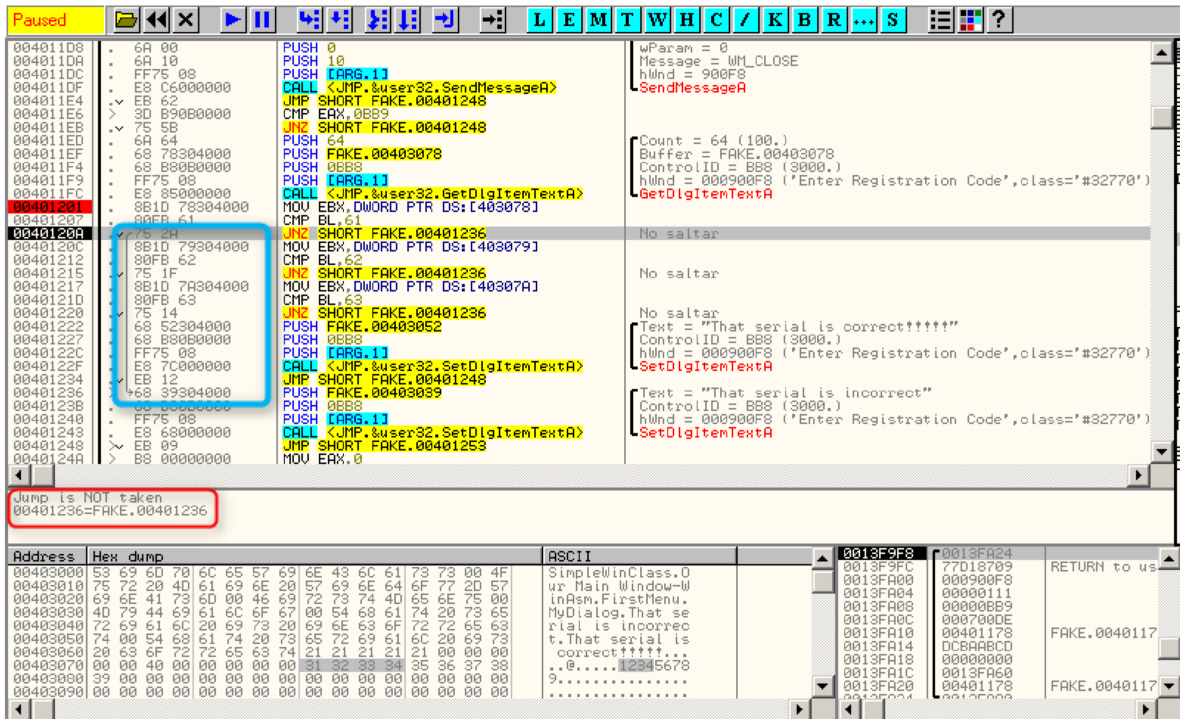
La misma información obtendríamos si miramos en el espacio que hay entre la ventana de desensamblaje y la ventana dump.



El siguiente paso será decirle a Olly que no salte. Para ello, dentro de la ventana de los registros fijémonos en la bandera Z. En estos momentos el valor de Z es 0, o lo que es lo mismo, después de comparar el valor 61h con 31h, Olly dedujo (correctamente) que ambos valores no son cero y por tanto la comparación es FALSE. Para convertir la comparación en TRUE basta con hacer doble clic sobre el cero:



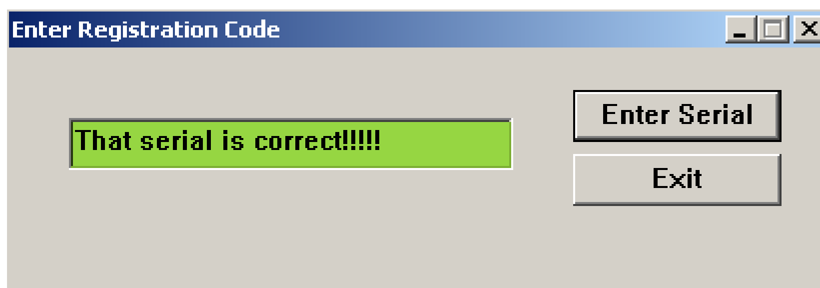
El valor de Z es ahora 1, y la flecha roja cambio a gris. Sabemos entonces que este salto no va a ser tomado.



Pulsamos F8, y llegamos a la siguiente instrucción. Aquí EBX va a ser cargado con el segundo carácter de nuestro serial para posteriormente ser comparado con el valor 62h. Sabemos que el segundo carácter de nuestro serial no es 62h, por lo tanto saltaremos al mensaje malo a no ser que tomemos las precauciones necesarias. ¡Hacemos doble clic sobre el valor de la bandera Z para sustituir el cero por un uno!

Antes del último salto se comparan el tercer carácter de nuestro serial con el valor 63h. Y como sabemos que no son iguales cambiamos el valor de la bandera Z por última vez.

Pulsamos F8 para situarnos en la dirección 401222. A partir de aquí no hay ningún salto más, por lo que o bien seguimos pulsando F8 o pulsamos F9 para ejecutar el programa y llegar a la conclusión de que:



Reflexiones finales

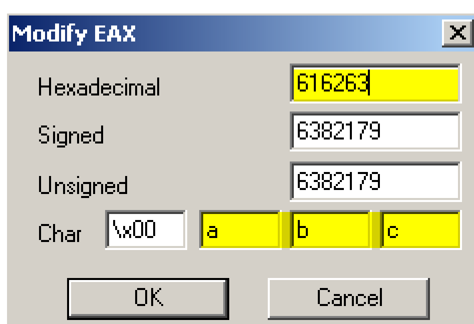
¿Qué ocurriría si tomo los tres valores de las comparaciones, los convierto de hexadecimales a ASCII e introduzco el resultado en la ventana de registrar?

Cargo el programa en Olly y busco los valores de las comparaciones:

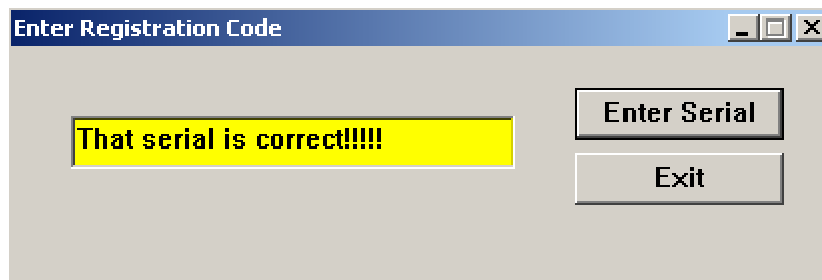
```
MOV EBX,DWORD PTR DS:[403078]
CMP BL,61
JNZ SHORT FAKE.00401236
MOV EBX,DWORD PTR DS:[403079]
CMP BL,62
JNZ SHORT FAKE.00401236
MOV EBX,DWORD PTR DS:[40307A]
CMP BL,63
JNZ SHORT FAKE.00401236
```

Tenemos por lo tanto en hexadecimal: 61 62 63

Hacemos doble clic sobre un registro cualquiera para convertir el valor en ASCII.



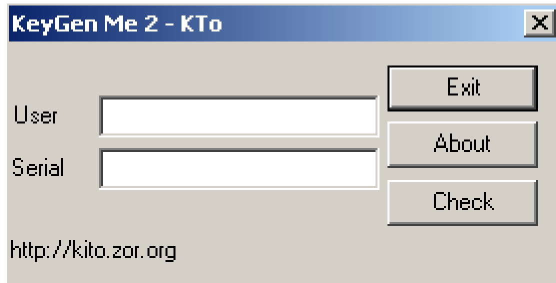
Pulsamos F9 para ejecutar el programa, y nos registramos con el serial: abc. Hacemos clic en “Enter Serial”



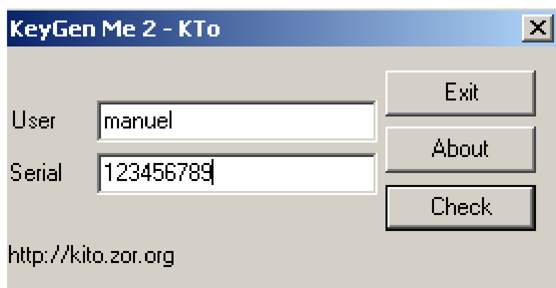
Conclusión: No solo hemos llegado al mensaje bueno, sino que fuimos capaz de sacar el serial autentico y todo ello sin hacer modificaciones en el código del programa.

7.2 Caso práctico 2: Parches

Abrimos Olly y cargamos Crackme2.exe. Pulsamos F9 para estudiar el comportamiento del programa.

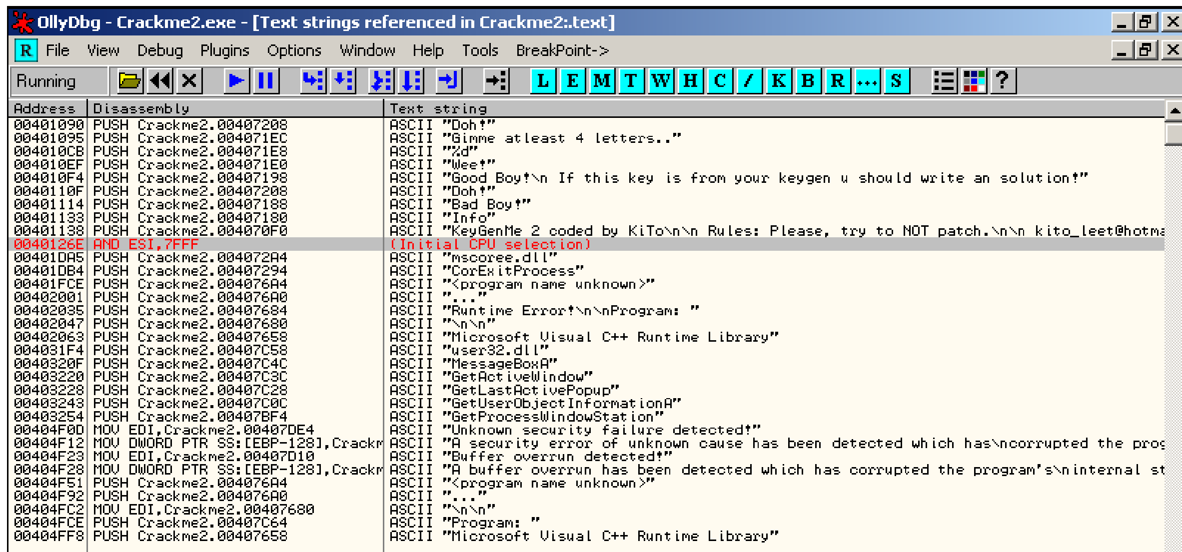


Como podemos ver se trata de un programa estándar. Rellenamos los campos en blanco.



Hacemos clic en “Check”.

Empezaremos a utilizar nuestra primera e única herramienta (que conocemos hasta ahora), para analizar el código del programa: dentro de la ventana de desensamblado hacemos clic con el botón derecho y seleccionamos “Search for” -> “All referenced rext strings”.



De la ventana anterior podemos extraer la siguiente información:

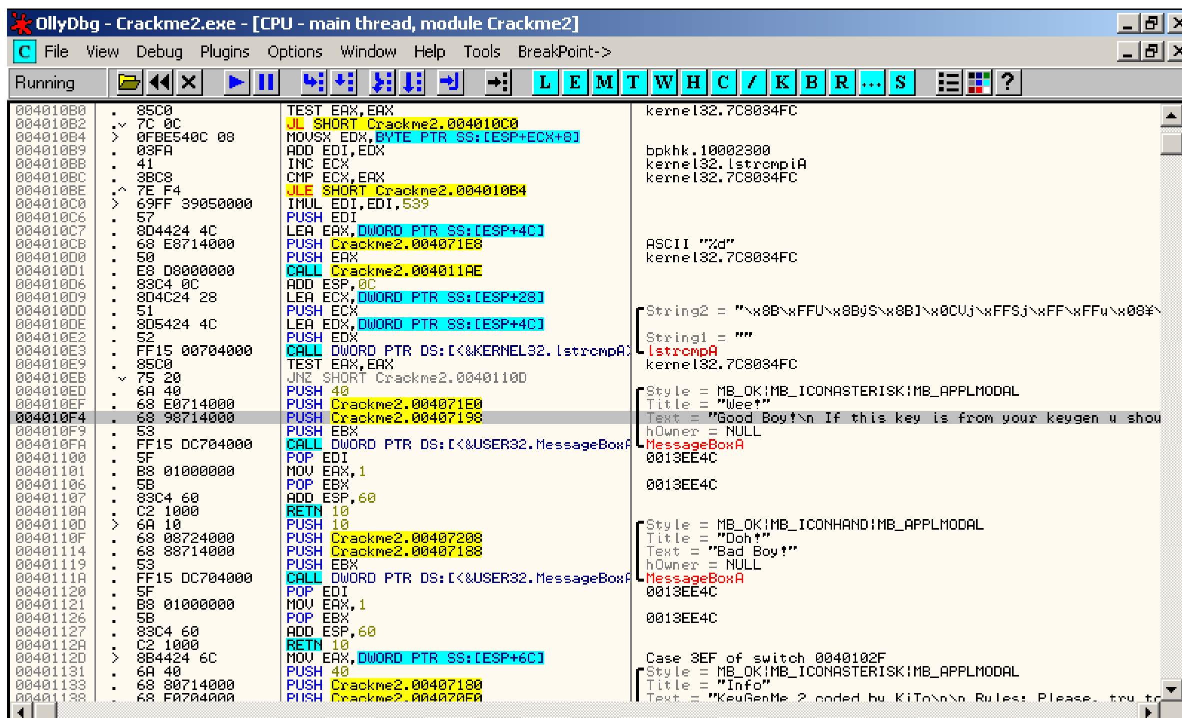
- El serial tiene que tener un mínimo de 4 caracteres.

```
00401095 PUSH Crackme2.004071EC          ASCII "Gimme atleast 4 letters.."
```

- Sabemos dónde localizar el "good boy" y el "bad boy".

```
004010F4 PUSH Crackme2.00407198          ASCII "Good Boy!\n If this key is from your keygen u should write an solution!"
0040110F PUSH Crackme2.00407208          ASCII "Doh!"
00401114 PUSH Crackme2.00407188          ASCII "Bad Boy!"
```

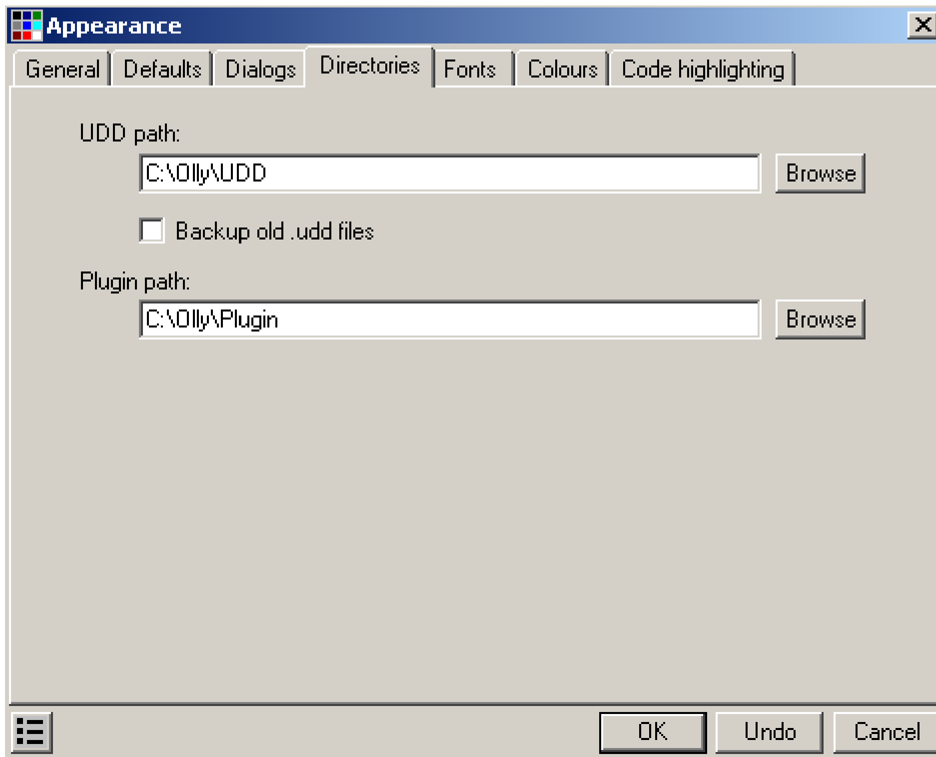
Hacemos doble clic en el "good boy", lo que nos lleva a la dirección 4010F4 en la ventana de desensamblado.

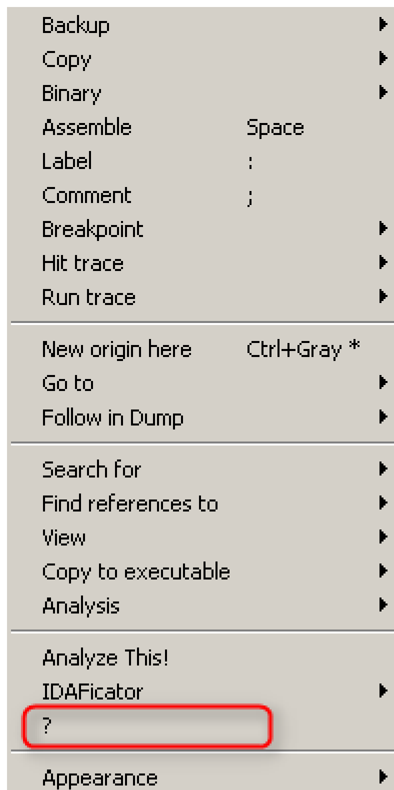


Nos situamos sobre la primera instrucción de salto (JNZ), en la dirección 4010EB, para averiguar hasta donde nos lleva.

Podemos ver como la instrucción nos lleva directamente al “bad boy”. Sabemos también que delante de una instrucción de salto debe haber una instrucción de comparación, cuyo resultado determinará si finalmente tomaremos el salto o no. La instrucción que precede al salto es: TEST EAX,EAX. Si queremos averiguar el significado de la instrucción TEST, basta con hacer clic con el botón derecho sobre TEST y seleccionar el signo de interrogación lo que abrirá el plugin MnemonicHelp.

Nota: Para instalar un plugin en Olly debemos crear primero una carpeta que nos permitirá guardar todos los plugins que vayamos descargando. Seleccionamos Options de la barra de menú, y hacemos clic en Appearance. A continuación hacemos clic sobre la pestaña Directories y escribimos la ruta tanto para la carpeta UDD como para la carpeta Plugin. Dentro de la carpeta UDD se irán guardando todos los archivos de las aplicaciones que vayamos manipulando en Olly. Cualquier breakpoint, comentario,… será almacenado en estos archivos lo que nos permitirá volver a cargar la aplicación más adelante conservando las últimas modificaciones hechas en el programa.


Appearance
General
Defaults
Dialogs
Directories
Fonts
Colours
Code highlighting
UDD path:
C:\Olly\UDD
Browse
Backup old .udd files
Plugin path:
C:\Olly\Plugin
Browse
OK
Undo
Cancel


Backup
Copy
Binary
Assemble Space
Label :
Comment ;
Breakpoint
Hit trace
Run trace
New origin here Ctrl+Gray *
Go to
Follow in Dump
Search for
Find references to
View
Copy to executable
Analysis
Analyze This!
IDAFicator
?
Appearance

Intel x86 Instructions

Archivo Edición Marcador Opciones Ayuda

Contenido Índice Atrás Imprimir

TEST—Logical Compare

See also

Opcode	Instruction	Description
A8 *ib*	TEST AL,*imm8*	AND *imm8* with AL; set SF, ZF, PF according to result
A9 *iw*	TEST AX,*imm16*	AND *imm16* with AX; set SF, ZF, PF according to result
A9 *id*	TEST EAX,*imm32*	AND *imm32* with EAX; set SF, ZF, PF according to result
F6 /0 *ib*	TEST *r/m8*,*imm8*	AND *imm8* with *r/m8*; set SF, ZF, PF according to result
F7 /0 *iw*	TEST *r/m16*,*imm16*	AND *imm16* with *r/m16*; set SF, ZF, PF according to result
F7 /0 *id*	TEST *r/m32*,*imm32*	AND *imm32* with *r/m32*; set SF, ZF, PF according to result
84 /*r*	TEST *r/m8*,*r8*	AND *r8* with *r/m8*; set SF, ZF, PF according to result
85 /*r*	TEST *r/m16*,*r16*	AND *r16* with *r/m16*; set SF, ZF, PF according to result
85 /*r*	TEST *r/m32*,*r32*	AND *r32* with *r/m32*; set SF, ZF, PF according to result

Description

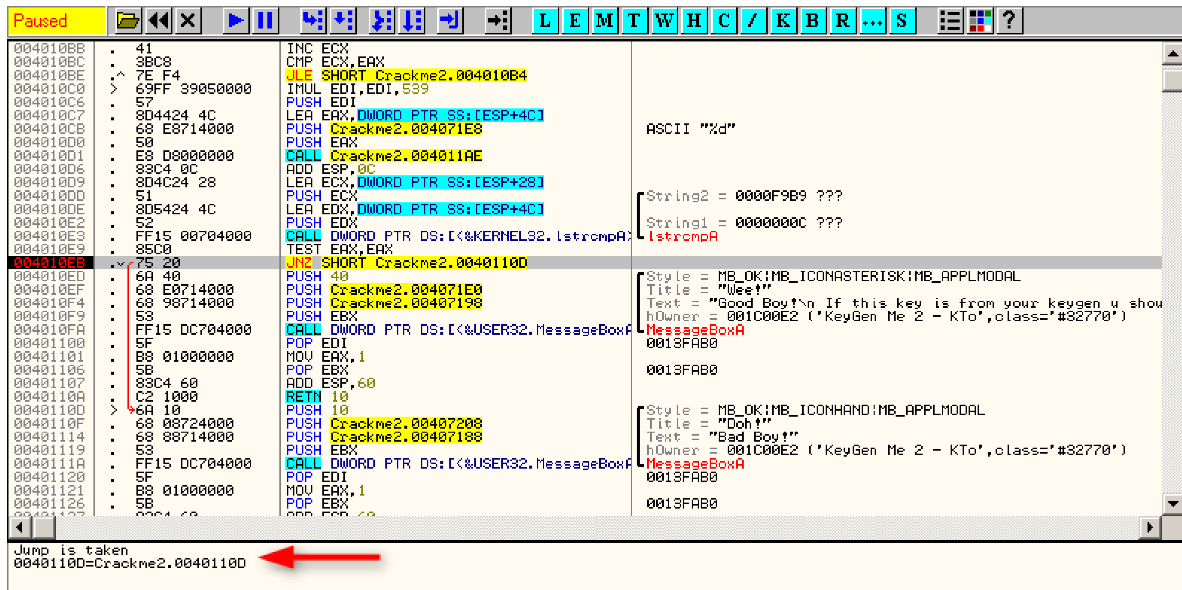
Computes the bit-wise logical AND of first operand (source 1 operand) and the second operand (source 2 operand) and sets the SF, ZF, and PF status flags according to the result. The result is then discarded.

Operation

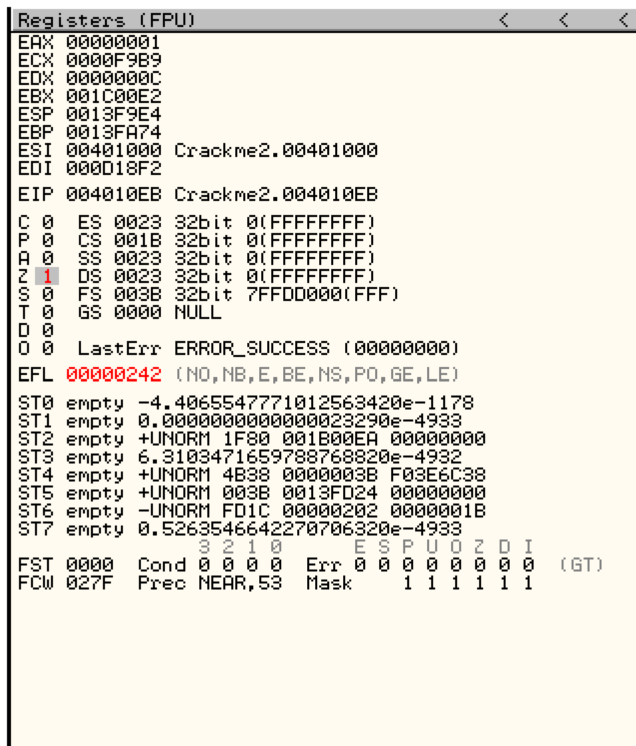
```
TEMP ← SRC1 AND SRC2;
SF ← MSB(TEMP);
IF TEMP = 0
   THEN ZF ← 0;
   ELSE ZF ← 1;
```

De la definición podemos concluir que la instrucción TEST lo que hace es comparar si los dos registros son idénticos, o lo que es lo mismo comprobará si cualquiera de ellos vale cero. Si EAX no vale cero entonces saltaremos a la dirección 40110D (nuestro “bad boy”).

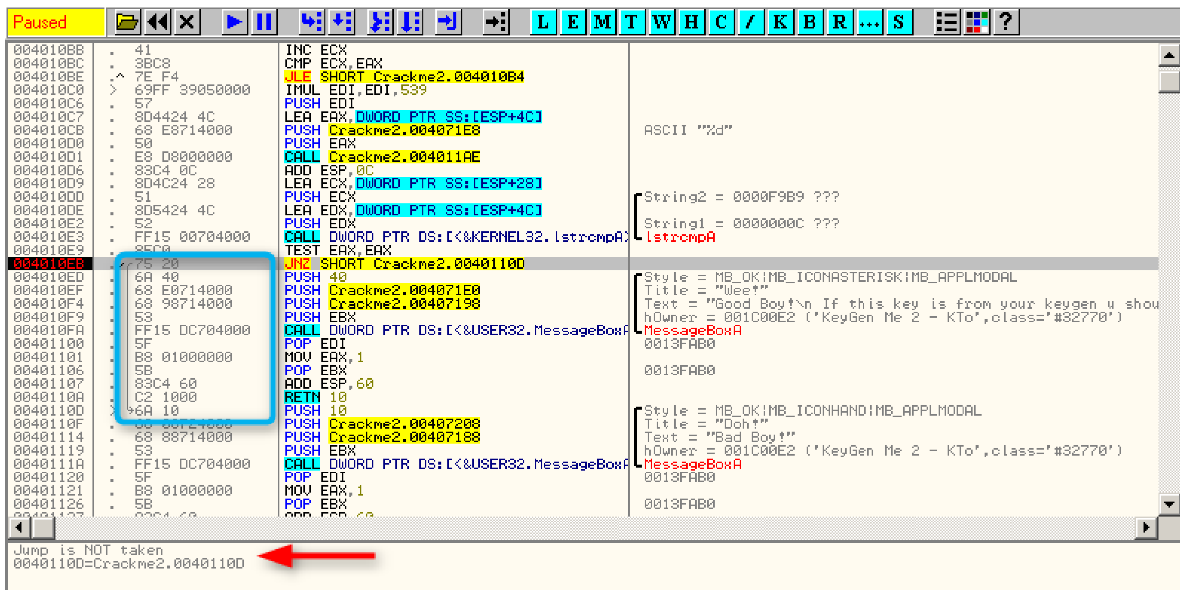
Como no queremos tomar el salto, pondremos un Breakpoint en la instrucción JNZ y reiniciamos la aplicación. Introducimos nuestro usuario y el serial (con un mínimo de cuatro caracteres), hacemos clic en “Check” y Olly se detendrá en nuestro Breakpoint.

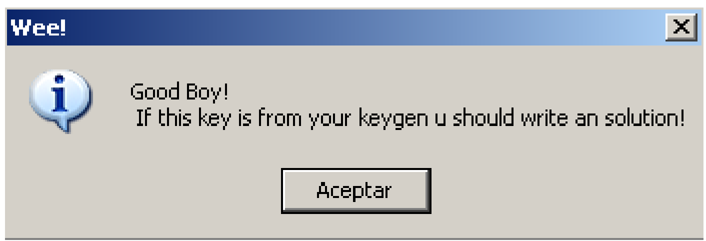


Vemos que tomaremos el salto al “bad boy” a no ser que hagamos algo para impedir a Olly de tomar el salto. Vamos pues a la ventana de registros y cambiamos el valor de la bandera Z haciendo doble clic sobre el cero.



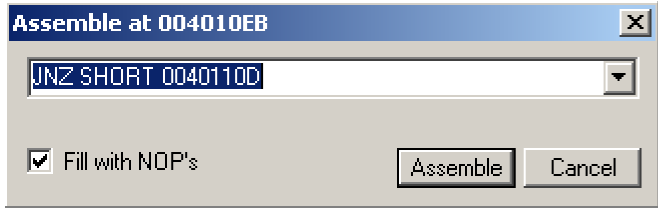
En la ventana de desensamblado la flecha roja cambio a gris, por lo que podemos estar seguros de que no vamos a tomar el salto. Comprobemoslo reiniciando la aplicación.



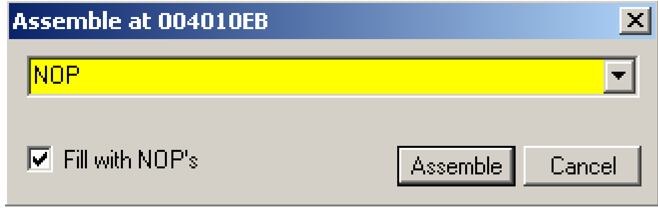


A continuación veremos como parchear esta aplicación. Para ello reiniciamos el programa, introducimos el usuario y el serial hacemos clic en "Check" y vemos como Olly se detiene en nuestro breakpoint.

Ahora en lugar de cambiar el valor de la bandera Z vamos a modificar el código del binario. Para ello seleccionamos el Breakpoint en la dirección 4010EB y hacemos clic en la instrucción JNZ SHORT Crackm2.0040110D. Pulsamos la barra espaciadora. Y se abre una nueva ventana con la instrucción que habíamos seleccionado.



Sustituimos la instrucción por NOP (No Operation), con lo cual nos aseguramos de no saltar nunca, ya que estamos prescindiendo por completo de la instrucción JNZ.



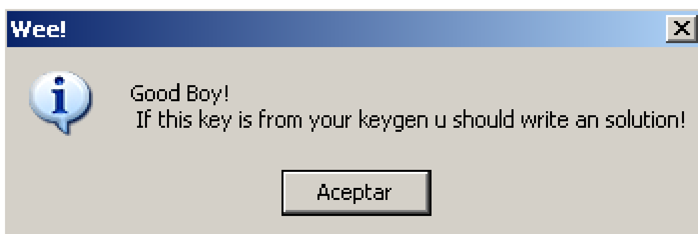
Hacemos clic en ”Assemble” **y** en “Cancel”.

Nota: Si no hacemos clic en “Cancel” y continuamos haciendo clic en “Assemble”, procederemos a ensamblar línea por línea todo el código del programa.

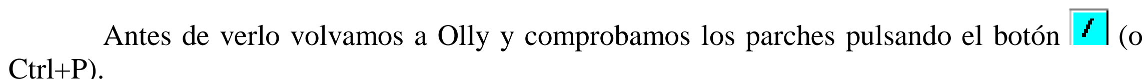
Volviendo a la ventana de desensamblaje podemos observar como ha desaparecido la instrucción de saltar y en su lugar se han añadido dos intrucciónes NOP.

La razón por la que Olly pone dos NOP’s es que el opcode NOP es de solo un byte y la instrucción que hemos reemplazado (JNZ) es de dos bytes. También podemos observar que la flecha que nos mostraba el salto hacia el “bad boy” ha desaparecido ya que ahora no va haber ningún salto en esta línea.

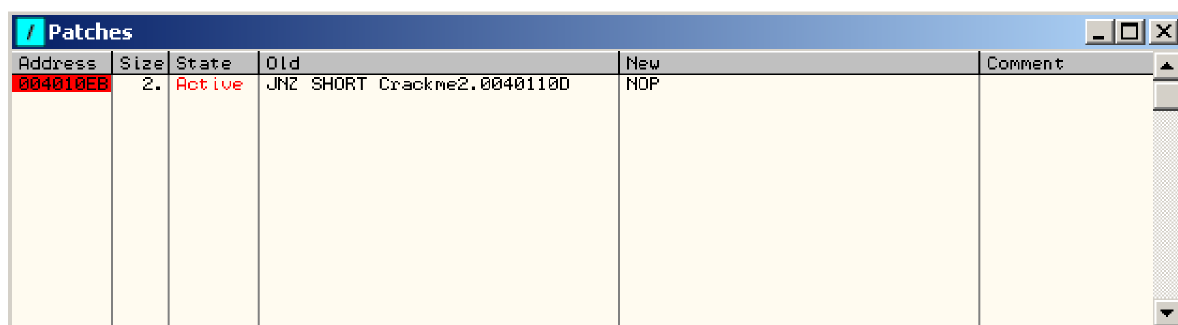
Pulsamos F8 hasta llegar a la dirección 4010FA donde nos espera el “good boy”.



Nota: Cuando cambiamos el código del binario para parchear una aplicación, debemos activarlo siempre que reiniciemos el programa. De lo contrario el programa se ejecutará sin tener en cuenta las modificaciones hechas.



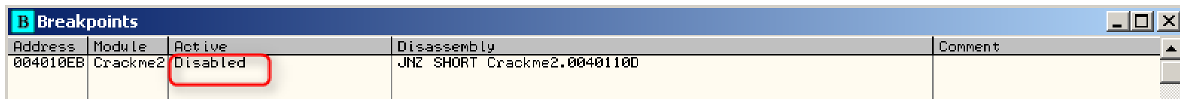
Antes de verlo volvamos a Olly y comprobamos los parches pulsando el botón (o Ctrl+P).



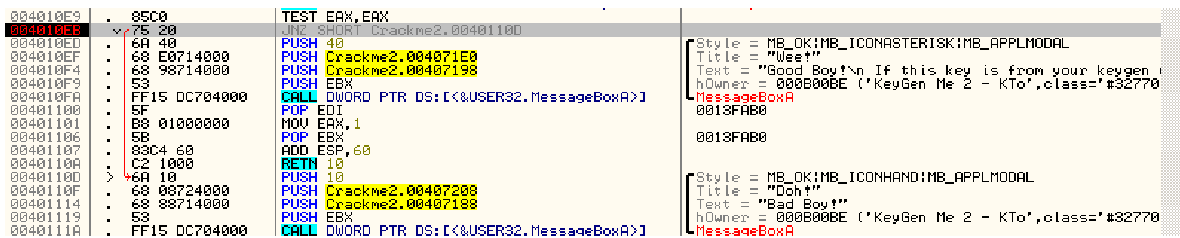
La ventana nos muestra el parche que hemos creado, la dirección, el tamaño, el estado y las instrucciónes intercambiadas. En este momento nuestra aplicación se está ejecutando, lo que significa que nuestro parche va a ser tenido en cuenta por la CPU.

Si reiniciamos ahora la aplicación, lo primero que nos muestra Olly es el siguiente error:

Hacemos caso omiso (por ahora) y pulsamos “Aceptar”. Abrimos la ventana de los Breakpoints y observamos como el Breakpoint que hemos puesto anteriormente está inhabilitado:

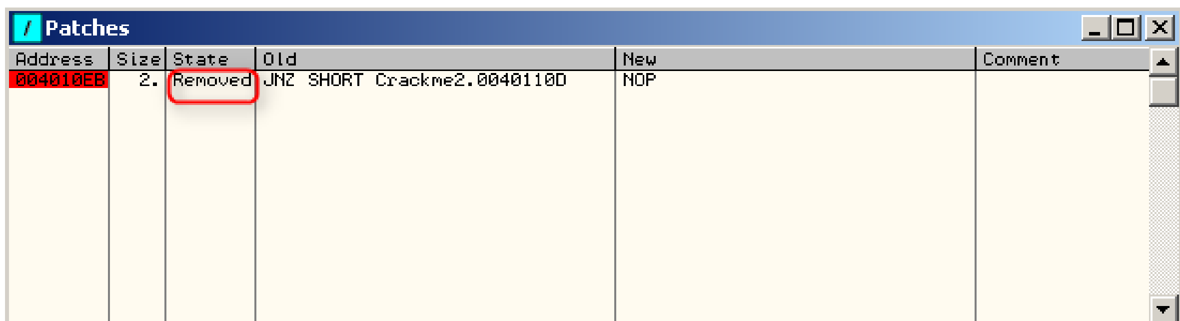


Para habilitarlo de nuevo pulsamos sobre la barra espaciadora y volvemos a ejecutar el programa. Introducimos nuestro nombre y el serial y Olly volverá a detenerse en nuestro Breakpoint.

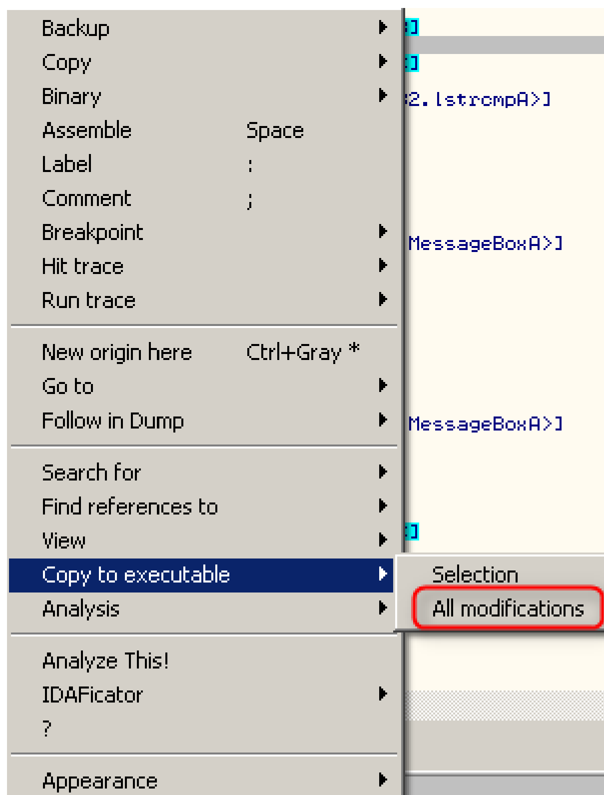


Vemos como las intrucciones NOP han desaparecido, volviendo la instrucción del salto pero esta vez en gris.

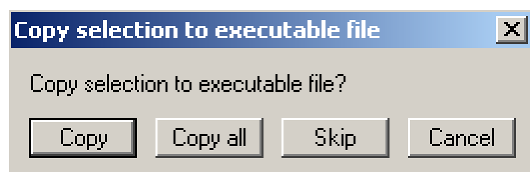
Abrimos la ventana de los parches:



Vemos que Olly desactivo el parche. Para volver activarlo pulsaremos la barra espaciadora. Si no queremos activar el parche cada vez que reiniciemos nuestra aplicación debemos guardar los cambios hechos en el código binario e asi garantizar su permanencia. Para ello hacemos clic con el botón derecho en cualquier parte de la ventana de desensamblaje y seleccionamos “Copy to executable ” -> “All modifications”.



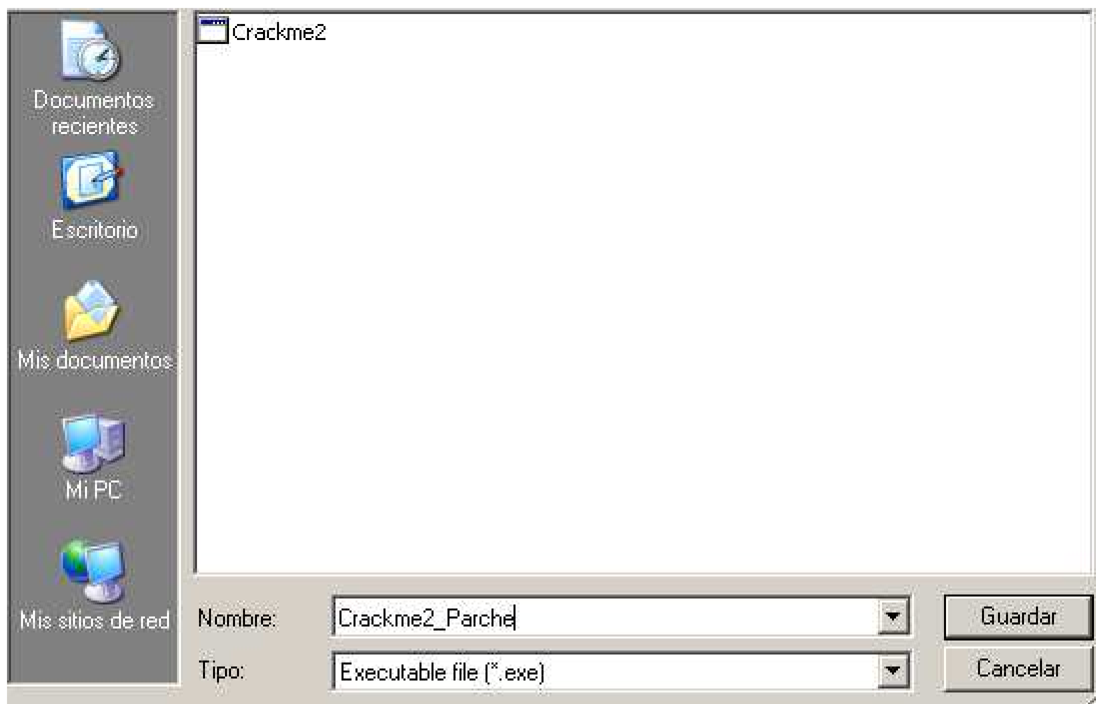
Seleccionamos “Copy all”.



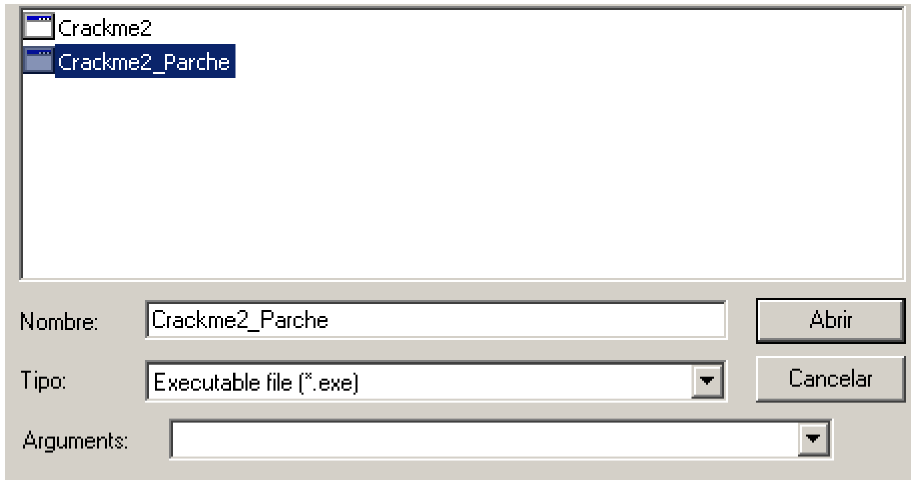
Se abre una nueva ventana Dump, con todo el código binario, incluido nuestro parche.

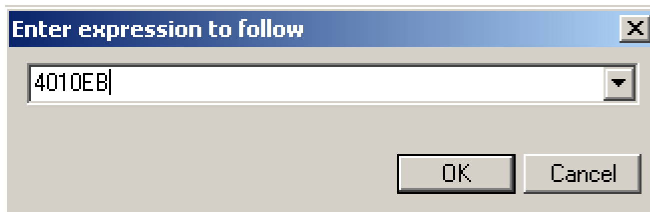
```
File C:\Documents and Settings\usuario\Mis documentos\...
000010EB  90              NOP
000010EC  90              NOP
000010ED  6A 40           PUSH 40
000010EF  68 E0714000     PUSH 4071E0
000010F4  68 98714000     PUSH 407198
000010F9  53              PUSH EBX
000010FA  FF15 DC704000   CALL DWORD PTR DS:[4070DC]
00001100  5F              POP EDI
00001101  B8 01000000     MOV EAX,1
00001106  5B              POP EBX
00001107  83C4 60         ADD ESP,60
0000110A  C2 1000         RETN 10
0000110D  6A 10           PUSH 10
0000110F  68 08724000     PUSH 407208
00001114  68 88714000     PUSH 407188
00001119  53              PUSH EBX
0000111A  FF15 DC704000   CALL DWORD PTR DS:[4070DC]
00001120  5F              POP EDI
00001121  B8 01000000     MOV EAX,1
00001126  5B              POP EBX
00001127  83C4 60         ADD ESP,60
0000112A  C2 1000         RETN 10
0000112D  8B4424 6C       MOV EAX,DWORD PTR SS:[ESP+6C]
00001131  6A 40           PUSH 40
00001133  68 80714000     PUSH 407180
00001138  68 F0704000     PUSH 4070F0
0000113D  50              PUSH EAX
0000113E  FF15 DC704000   CALL DWORD PTR DS:[4070DC]
00001144  5F              POP EDI
00001145  B8 01000000     MOV EAX,1
0000114A  5B              POP EBX
0000114B  83C4 60         ADD ESP,60
0000114E  C2 1000         RETN 10
00001151  8B4C24 6C       MOV ECX,DWORD PTR SS:[ESP+6C]
00001155  6A 01           PUSH 1
00001157  51              PUSH ECX
```

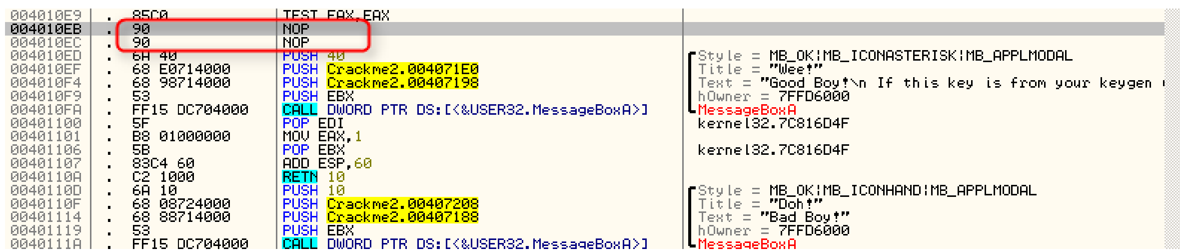
A continuación hacemos clic con el botón derecho sobre la ventana Dump, seleccionamos "Save file" y le damos un nombre:



Comprobamos que el nuevo ejecutable tenga el parche. Abrimos Crackme2_Parche.exe en Olly, pulsamos Ctrl+G e introducimos la dirección del parche (4010EB).

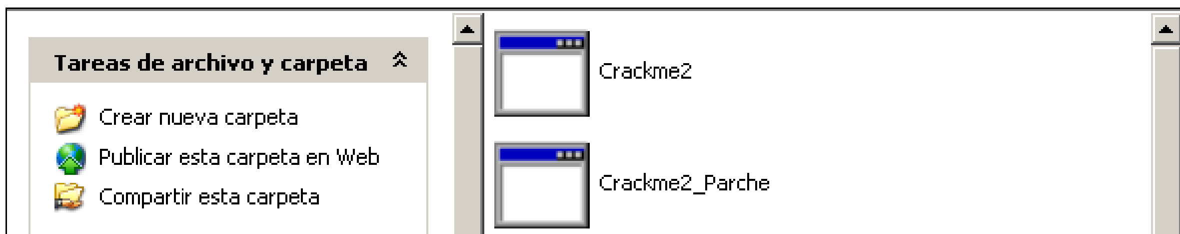




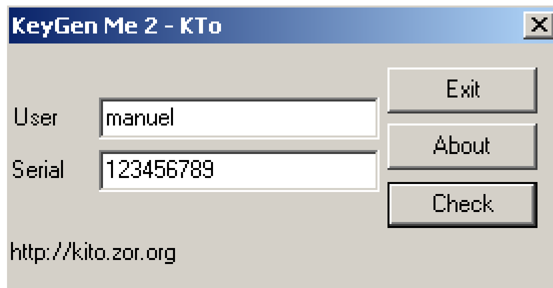


¡Aparece el parche!

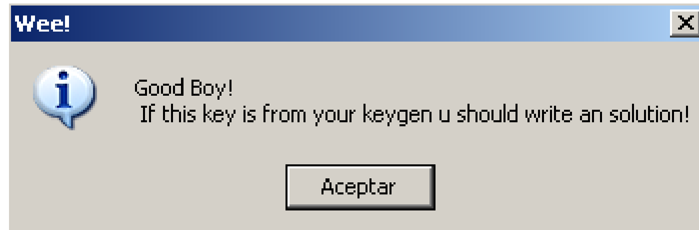
Ahora comprobamos el nuevo ejecutable haciendo doble clic en el:



Introducimos el nombre y el serial:



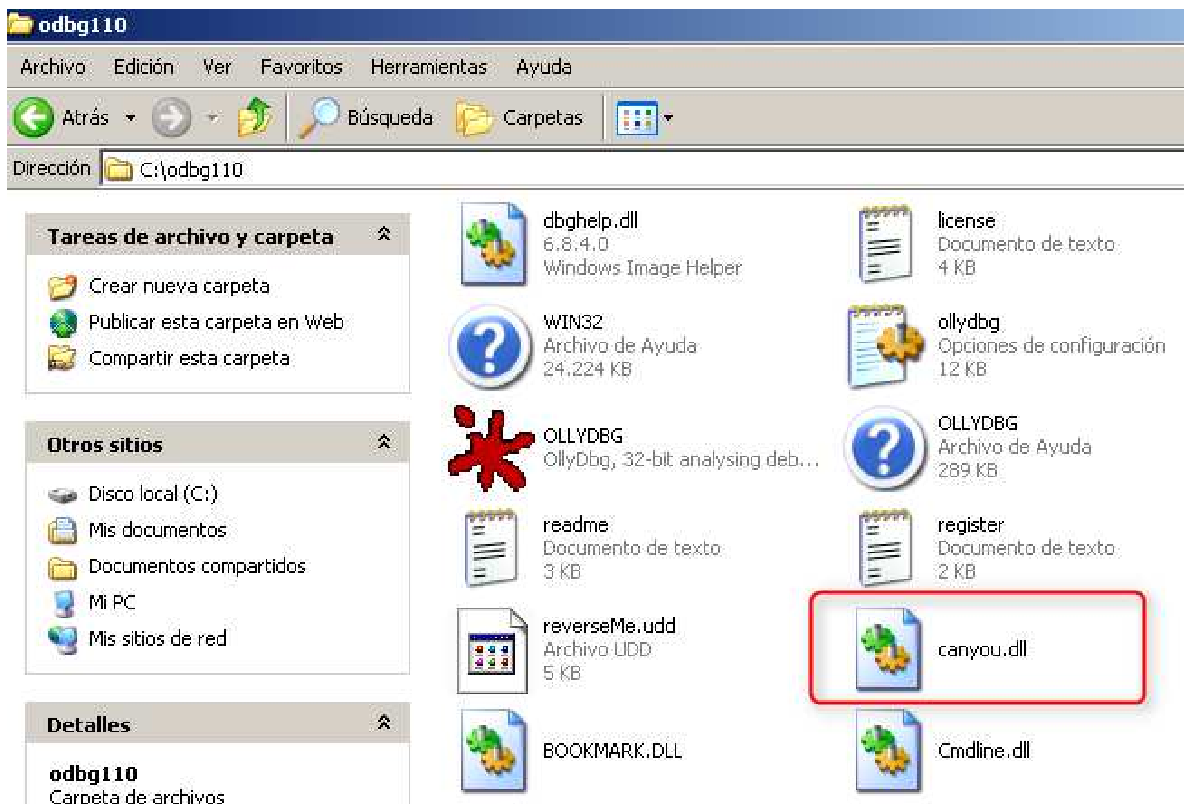
Hacemos clic en “Check”:

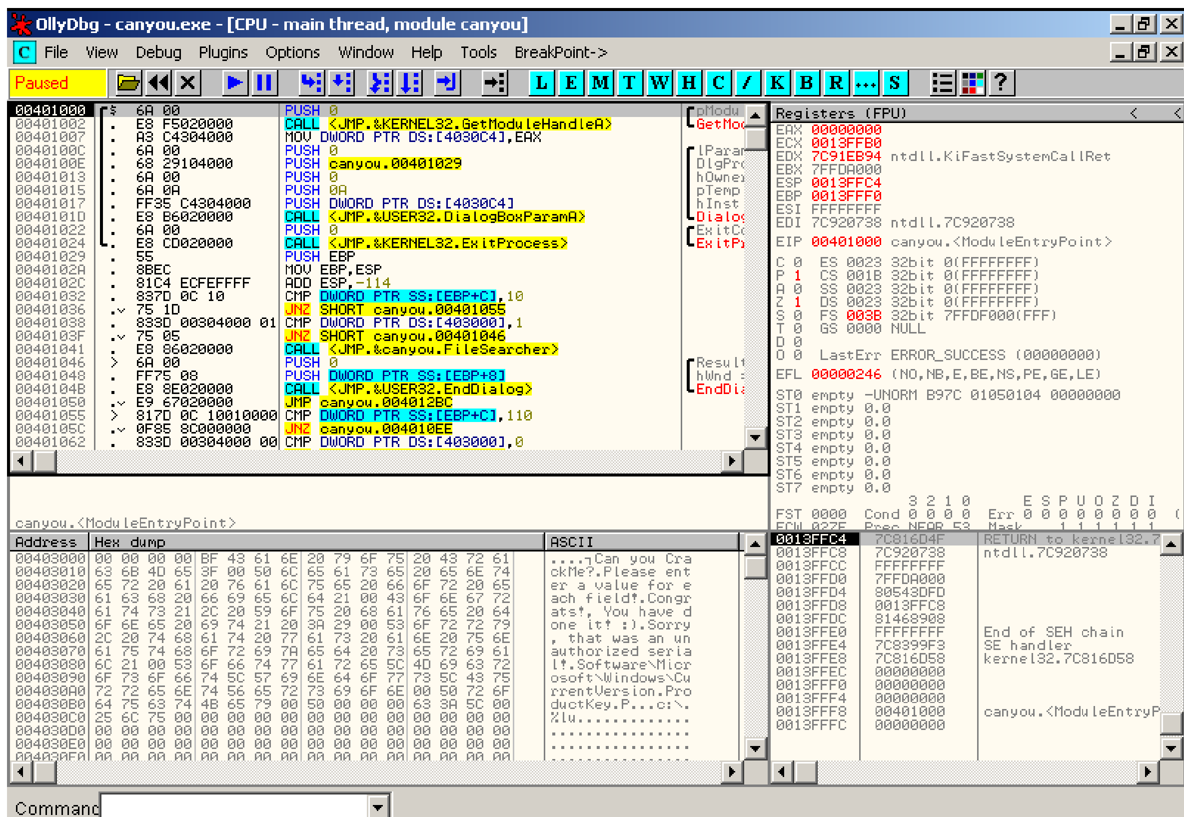


¡La aplicación ha sido parcheada de forma exitosa!

7.3 Caso práctico 3: Repasando conceptos

Abrimos Olly y cargamos la aplicación canyou.exe (para este ejercicio necesitaremos introducir el fichero canyou.dll en Olly).

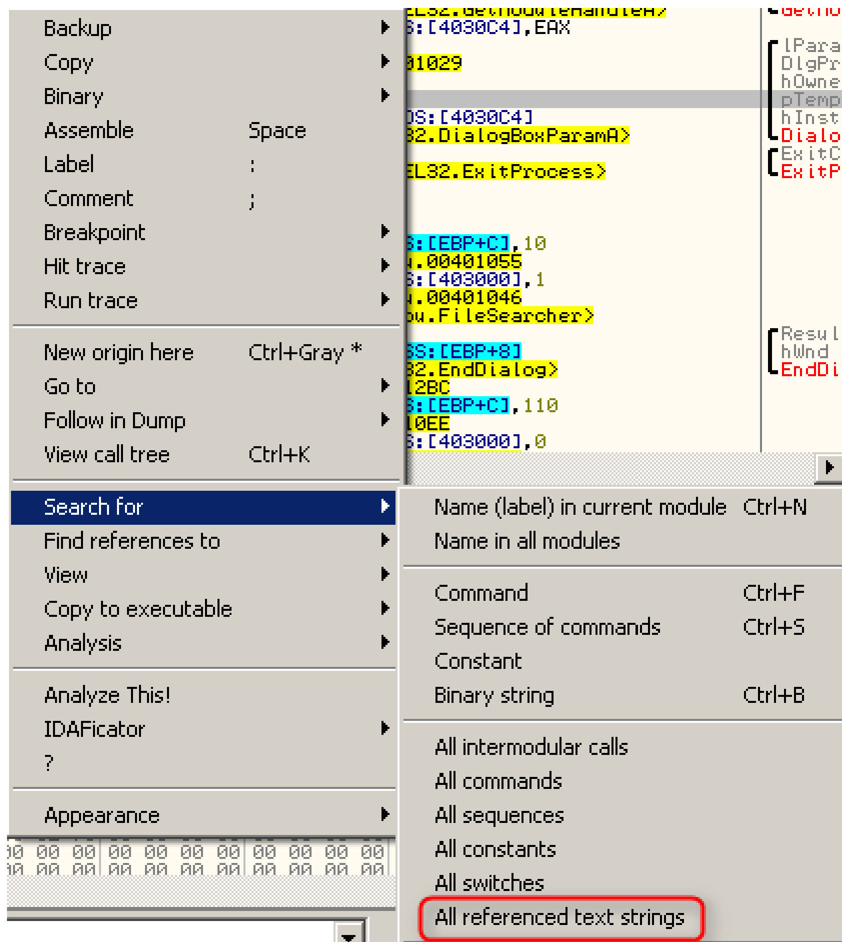




Como se vio en los ejercicios anteriores, una de las cosas más importantes que podemos hacer una vez cargada la aplicación en Olly, es averiguar el funcionamiento del programa. Para ello ejecutamos la aplicación pulsando F9.

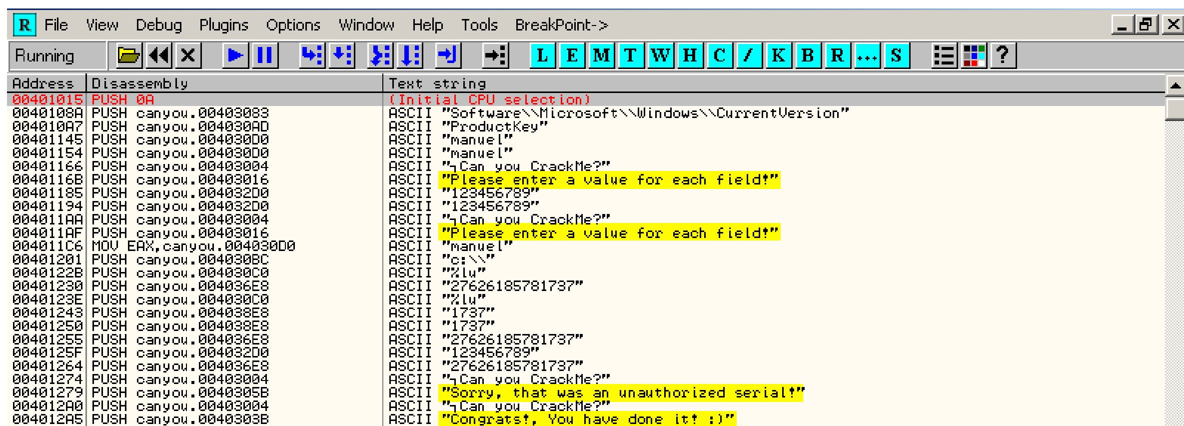
Rellenamos los campos de texto y pulsamos sobre “Gain Access”.

Siguiendo con el procedimiento habitual, buscaremos a continuación cadenas de texto que nos puedan dar alguna pista.

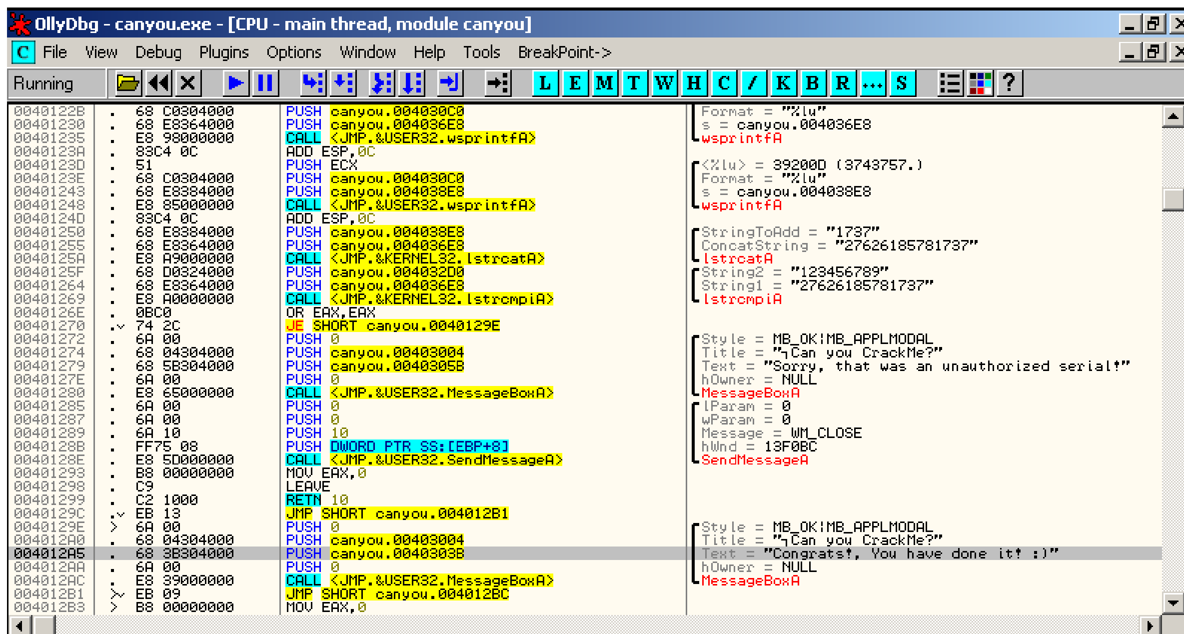


De la ventana que se abre a continuación podemos extraer algunas informaciones muy valiosas:

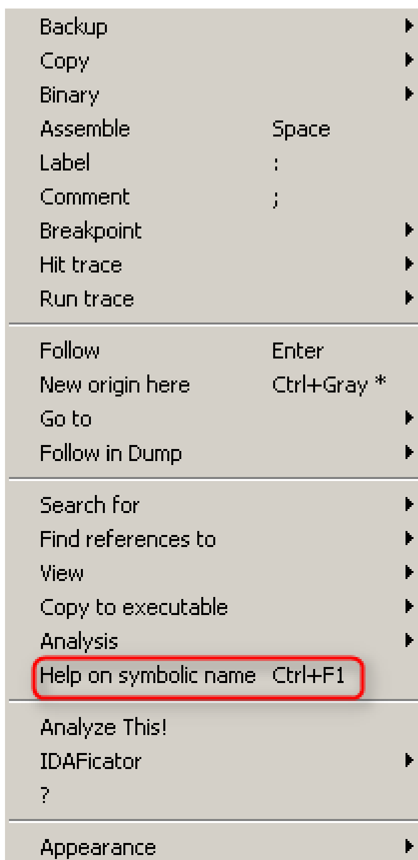
- Necesitamos introducir “algo” en cada campo de texto.
- Vamos a tener acceso tanto al “bad boy” como al “good boy”.

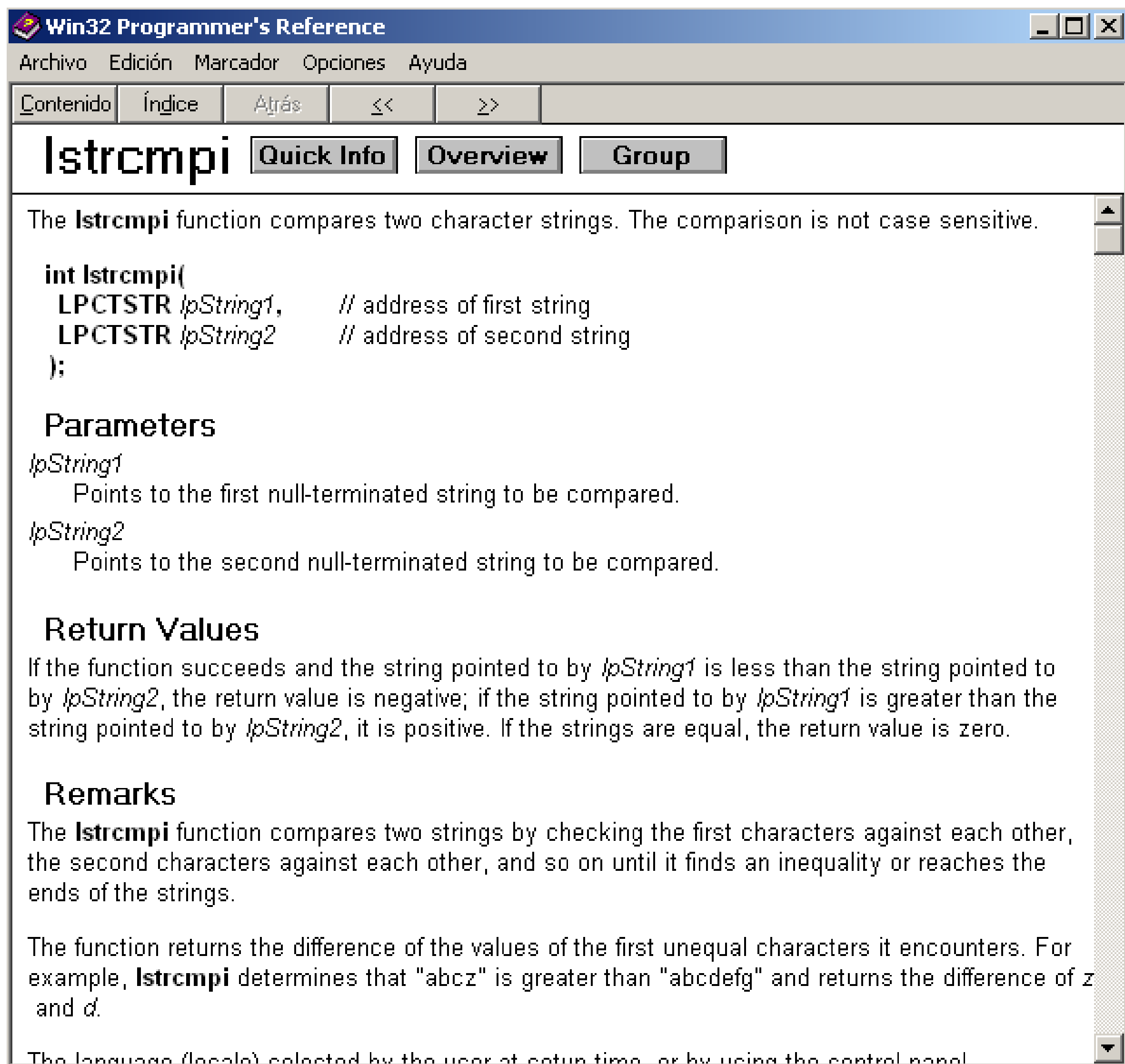


Hacemos doble clic en cualquiera de los “boys” lo que nos llevará al área de la ventana de desensamblaje que nos interesa estudiar.



A simple vista podemos identificar el “goog boy” precedido por el “bad boy” y que a su vez está precedido por una instrucción de salto y otra de comparación. Antes de analizar estas secciones, fijemonos en la instrucción anterior, donde aparece un CALL a la API de Windows lstrcmpiA. Para averiguar el significado de esta API nos situamos encima, hacemos clic con el botón derecho y seleccionamos “Help on symbolic name”.




Win32 Programmer's Reference

Archivo Edición Marcador Opciones Ayuda

Contenido Índice Atrás << >>

lstrcmpi Quick Info Overview Group

The **lstrcmpi** function compares two character strings. The comparison is not case sensitive.

```
int lstrcmpi(
  LPCTSTR lpString1,    // address of first string
  LPCTSTR lpString2     // address of second string
 );
```

Parameters

lpString1
Points to the first null-terminated string to be compared.

lpString2
Points to the second null-terminated string to be compared.

Return Values

If the function succeeds and the string pointed to by *lpString1* is less than the string pointed to by *lpString2*, the return value is negative; if the string pointed to by *lpString1* is greater than the string pointed to by *lpString2*, it is positive. If the strings are equal, the return value is zero.

Remarks

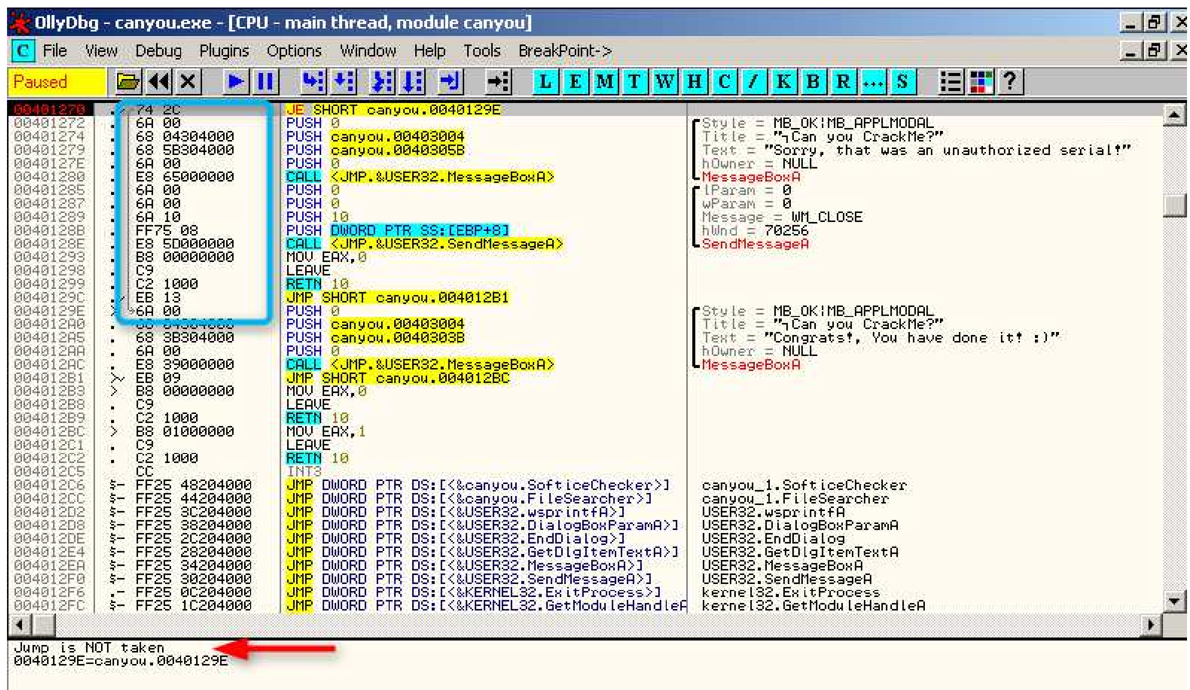
The **lstrcmpi** function compares two strings by checking the first characters against each other, the second characters against each other, and so on until it finds an inequality or reaches the ends of the strings.

The function returns the difference of the values of the first unequal characters it encounters. For example, **lstrcmpi** determines that "abcz" is greater than "abcdefg" and returns the difference of *z* and *d*.

Basicamente lo que hace esta API es comparar dos cadenas. Por lo general comparará la cadena que ha sido introducido por el usuario con la cadena que tiene guardada la aplicación en su memoria (o que se haya creado de forma dinamica). Si el resultado de la comparación es cero, entonces el usuario ha introducido las credenciales correctas, o lo que es lo mismo, ambas cadenas son iguales.

En nuestro caso si EAX devuelve el valor cero, ambas cadenas serán iguales, en caso contrario no lo serán.

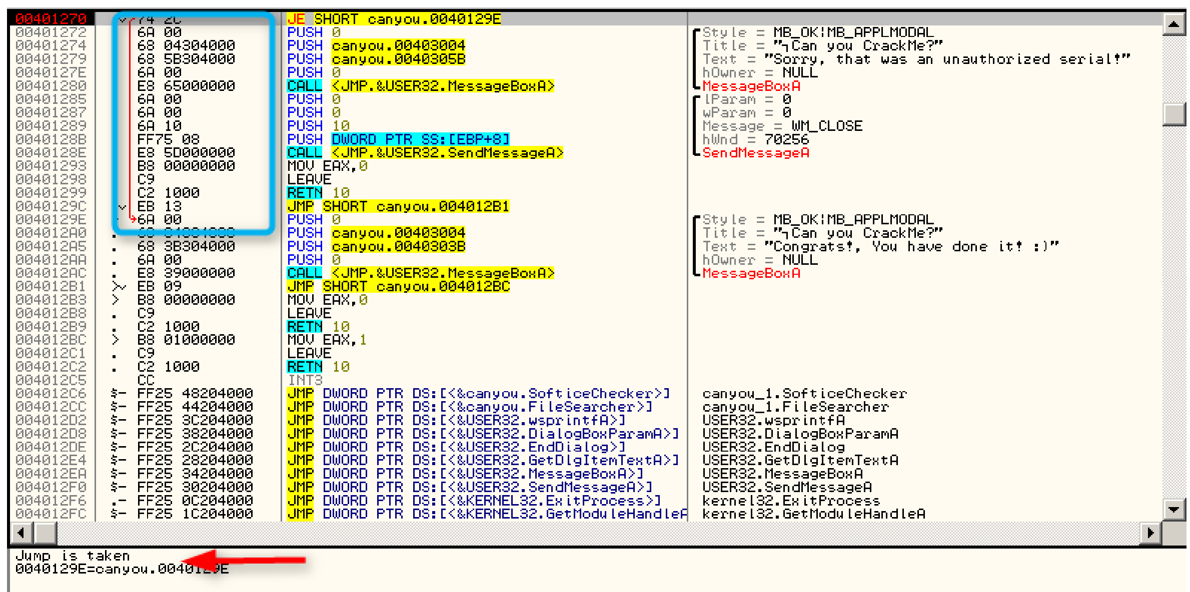
Una vez identificado la instrucción del salto, pongamos un Breakpoint en la dirección 401270, reiniciamos la aplicación, introducimos el nombre y el serial y vemos como Olly se detiene en nuestro Breakpoint.



Vemos, gracias a la flecha gris, que el salto no va a ser tomado y por lo tanto, Olly no nos mandará al área del "goog boy". Para remediar esto utilizaremos la técnica que ya conocemos: doble clic en el valor de la bandera Z para sustituir el cero por uno.

```
C 0  ES 0023 32bit 0(FFFFFFFF)
P 0  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDF000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)
```

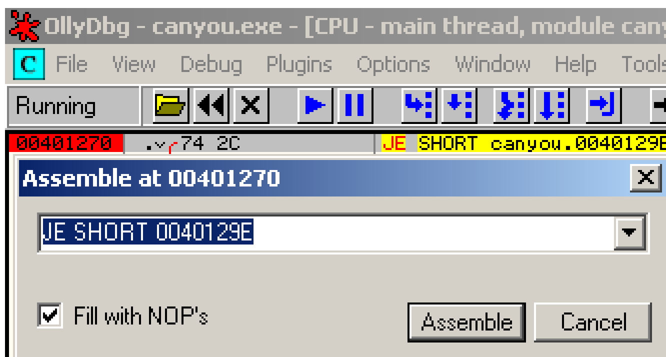
Ahora la flecha es de color rojo y podemos estar seguros de que Olly tomará el salto para llevarnos al "good boy".

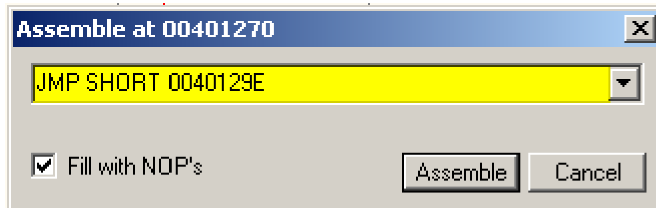


Pulsamos F9 para comprobarlo:

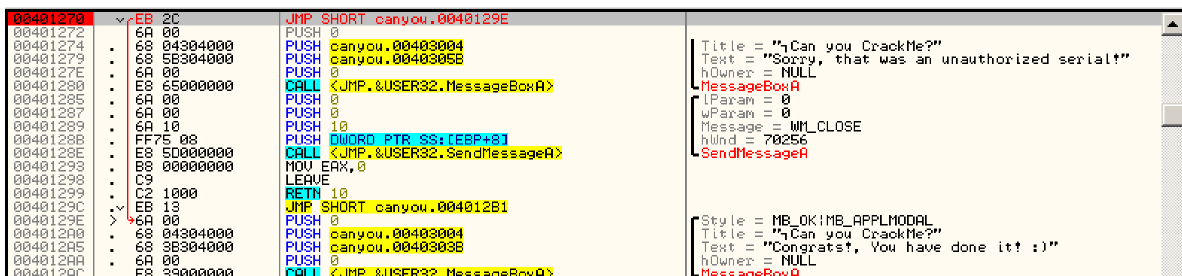
El siguiente paso consiste en parchear la aplicación. Pero esta vez no vamos a sustituir la instrucción de salto por la de un NOP, ya que esto nos llevaría al "bad boy" cada vez que ejecutemos el programa. En su lugar sustituiremos el JE (Jump on Equal) por un JMP, lo que hará que siempre saltemos al "good boy".

Busquemos por lo tanto nuestro Breakpoint, hacemos clic en la barra espaciadora y hacemos el cambio:





Hacemos clic en "Assemble" y en "Cancel" y vemos como se cambio el código en la venta de desensamblado:



Pulsamos F8 hasta pasar el área del "good boy".

El último paso consiste en guardar la aplicación parcheada en nuestro **disco duro.**

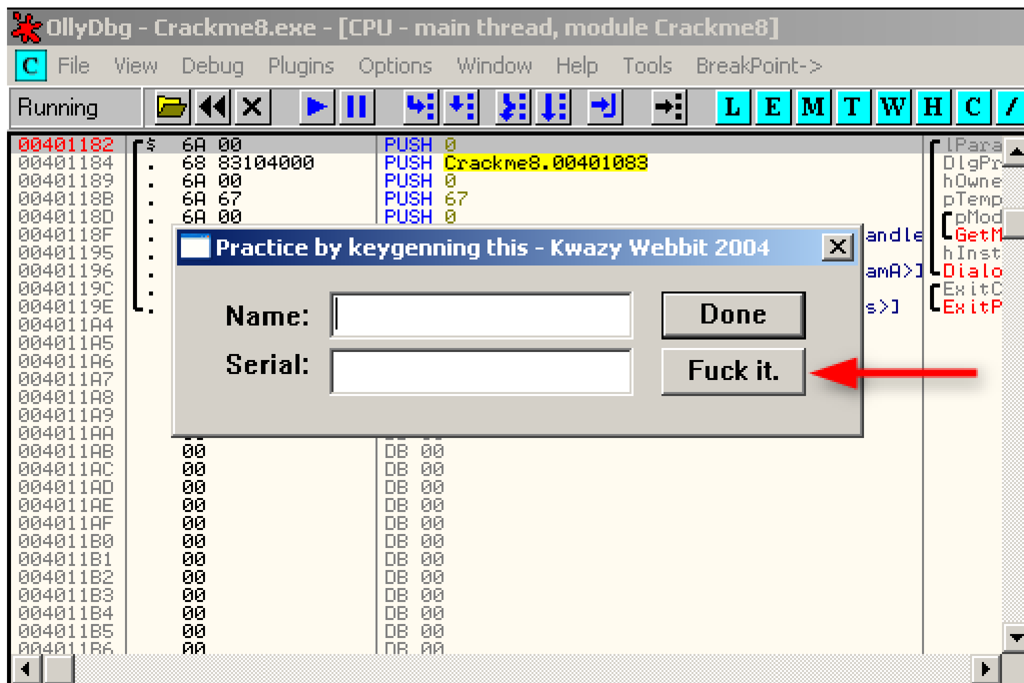
Cerramos Olly y hacemos doble clic sobre el nuevo ejecutable:

Introducimos un nombre y serial cualquiera:

Hacemos clic en "Gain Access!"

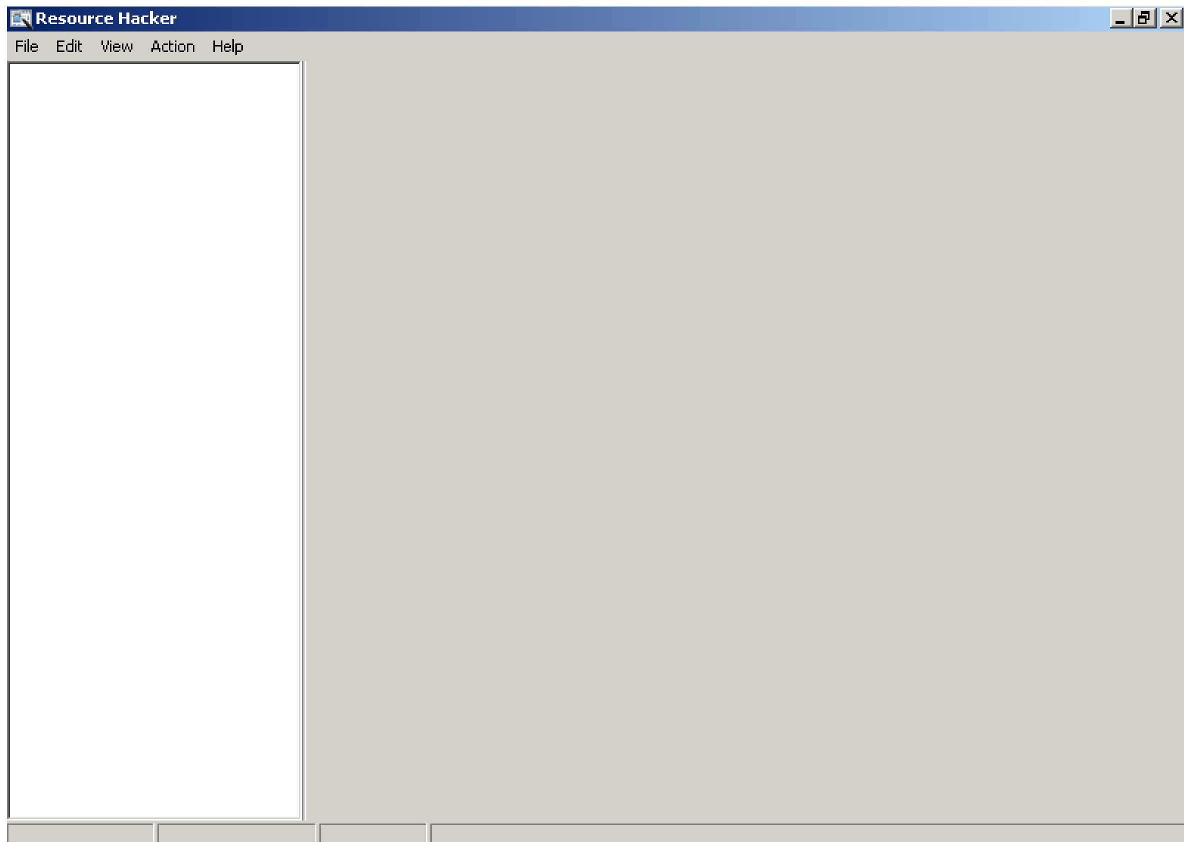
7.4 Caso práctico 4: Resource Hacker

Abrimos Olly y cargamos el ejecutable Crackme8.exe. Pulsamos F9.

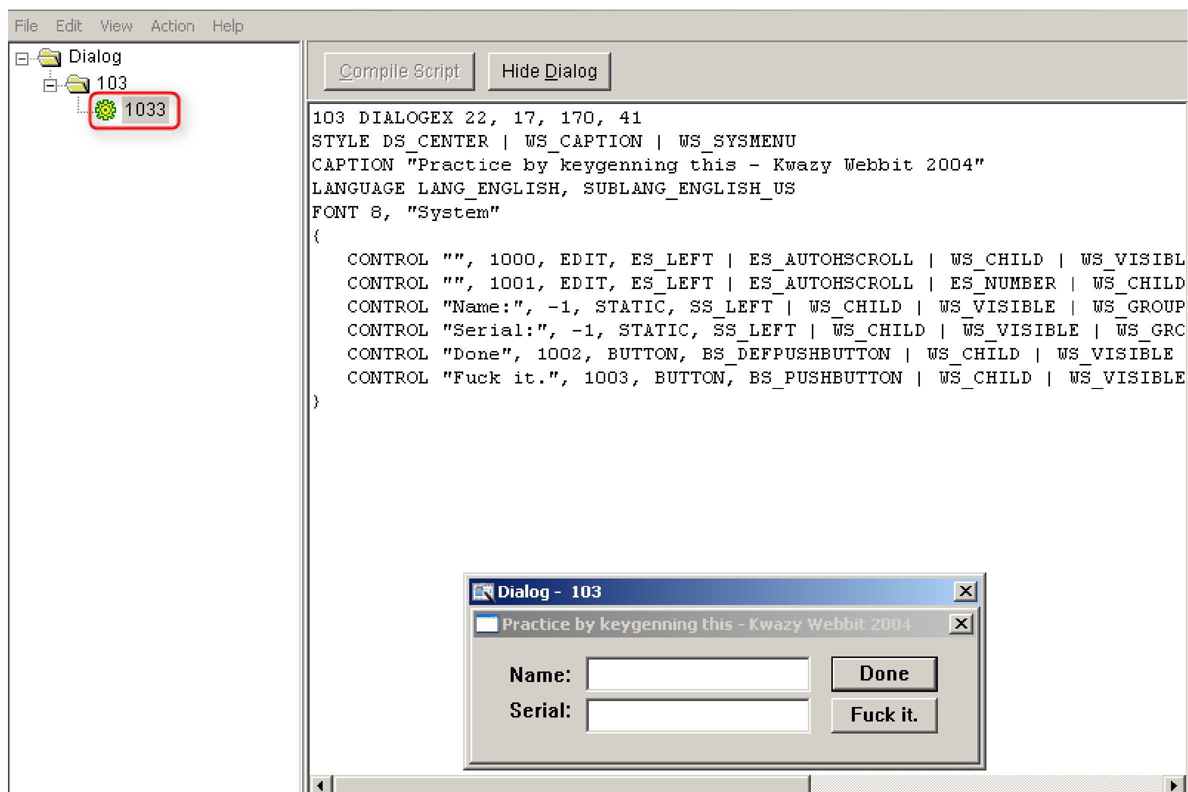


¡Una broma de mal gusto! Veamos cómo cambiar la apariencia del cuadro de dialogo.

Instalamos Resource Hacker haciendo doble clic sobre el ejecutable. Una vez instalado abrimos la aplicación.



Cargamos el ejecutable Crackme8.exe y desplegamos las carpetas. Hacemos clic sobre el icono 1033:



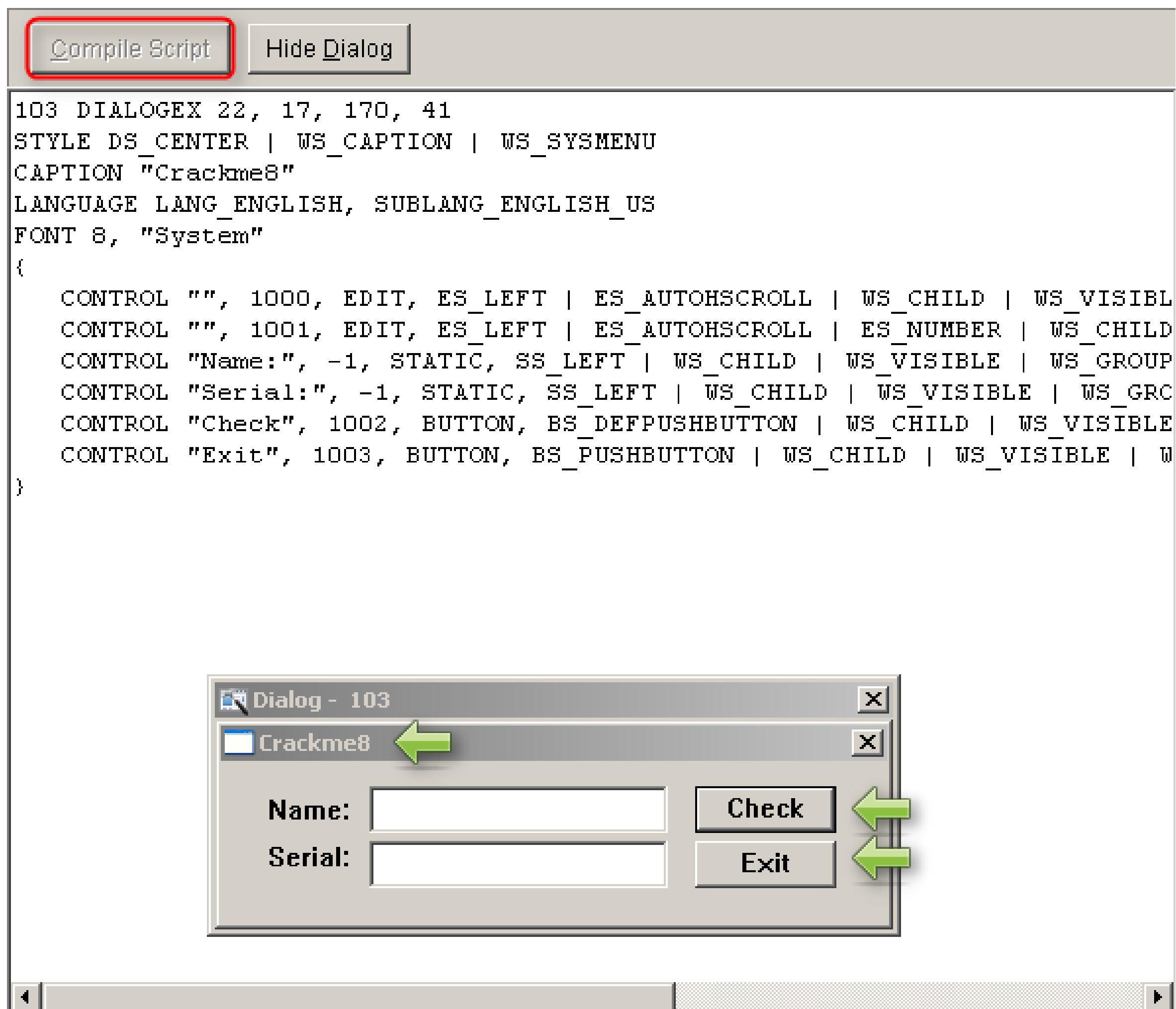
En la primera parte del panel aparecen especificaciones genéricas sobre el cuadro de dialogo como fuente, estilos, título de ventana… Debajo vemos los nombres de los cuadros de texto, Name y Serial y más abajo los nombres de los botones. Procedemos a cambiar los nombres de los botones:

```
{
    CONTROL "", 1000, EDIT, ES_LEFT | ES_AUTOHSCROLL | WS_CHILD | WS_VISIBL
    CONTROL "", 1001, EDIT, ES_LEFT | ES_AUTOHSCROLL | ES_NUMBER | WS_CHILD
    CONTROL "Name:", -1, STATIC, SS_LEFT | WS_CHILD | WS_VISIBLE | WS_GROUP
    CONTROL "Serial:", -1, STATIC, SS_LEFT | WS_CHILD | WS_VISIBLE | WS_GRC
    CONTROL "Check", 1002, BUTTON, BS_DEFPUSHBUTTON | WS_CHILD | WS_VISIBLE
    CONTROL "Exit", 1003, BUTTON, BS_PUSHBUTTON | WS_CHILD | WS_VISIBLE | W
}
```

Y ya que estamos, cambiamos también el título de la ventana:

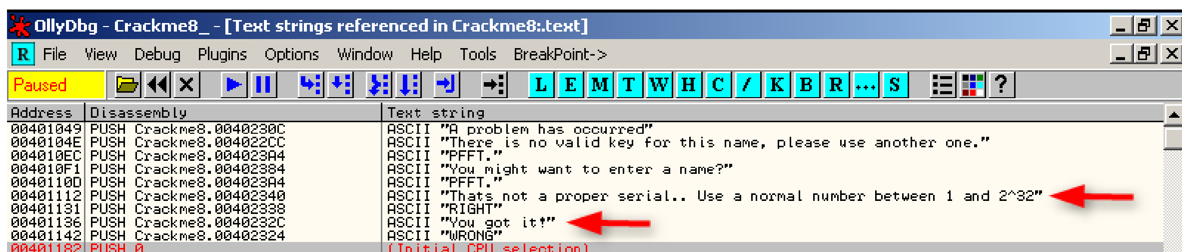
```
103 DIALOGEX 22, 17, 170, 41
STYLE DS_CENTER | WS_CAPTION | WS_SYSMENU
CAPTION "Crackme8"
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
FONT 8, "System"
```

Una vez hecho todos los cambios pulsamos “Compile Script” y guardamos los cambios.



Nota: Hemos incluido esta parte para concienciarnos de la importancia que tiene ejecutar una aplicación nada más cargarla en Olly. Es la única manera de obtener información tan valiosa como el periodo de prueba, características especiales del serial, (si es hard-coded o dinámico), campos de texto, etc.

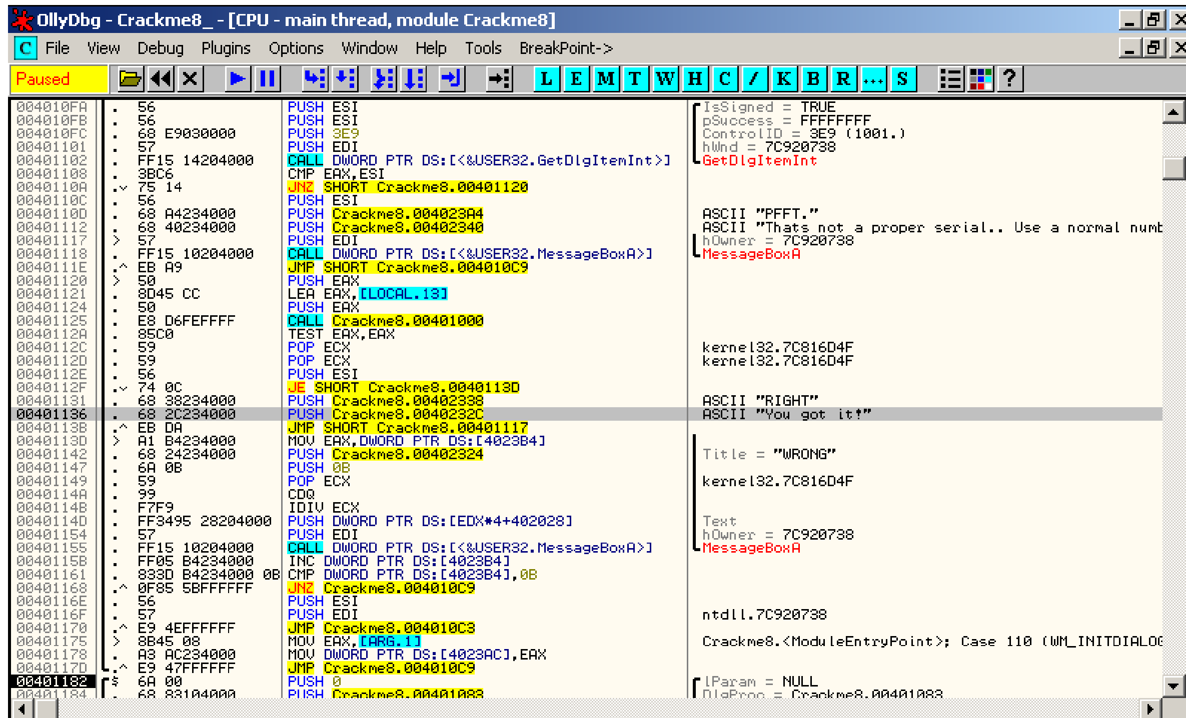
Volvamos a lo que realmente nos interesa. Abrimos Olly y cargamos el nuevo ejecutable. Una vez realizado el primer paso, busquemos las cadenas de texto:



De la busqueda anterior obtenemos la siguiente información:

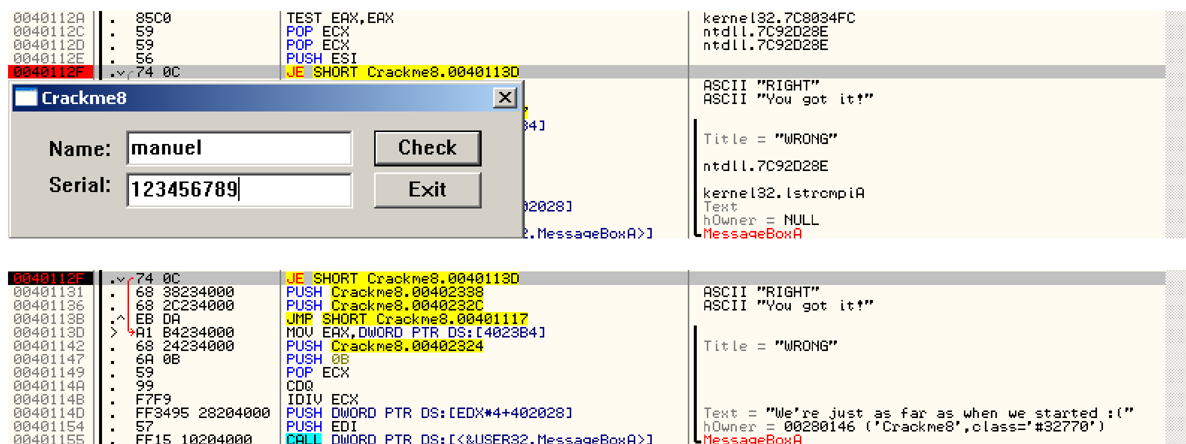
- La longitud del serial.
- La posición del “good boy”.

Hacemos doble clic en el mensaje de nuestro “good boy”, lo que nos abre la ventana de desensamblaje con el área que nos interesa:



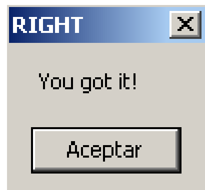
Dos líneas más arriba tenemos una instrucción de salto, en la dirección 40112F (JE SHORT Crackme8.0040113D) que es precedida por una intrucción de comparación en la dirección 40112A (TEST EAX,EAX) y el “bad boy” cuya rutina empieza en la dirección 40113D.

Ponemos un Breakpoint en la instrucción del salto y ejecutamos el programa. Introducimos un nombre y un serial, hacemos clic en “Check” y vemos como Olly se detiene en el Breakpoint.



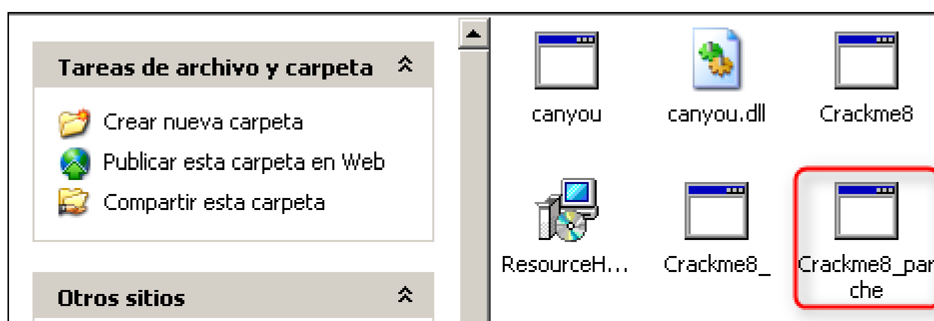
Vemos que Olly tomará el salto hacia el “bad boy”, lo que significa que tenemos que cambiar el valor de la bandera Z haciendo doble clic sobre el uno.

Vemos que el color de flecha cambio a gris. Si ahora pulsamos F9 saltamos directamente hacia el “good boy”.

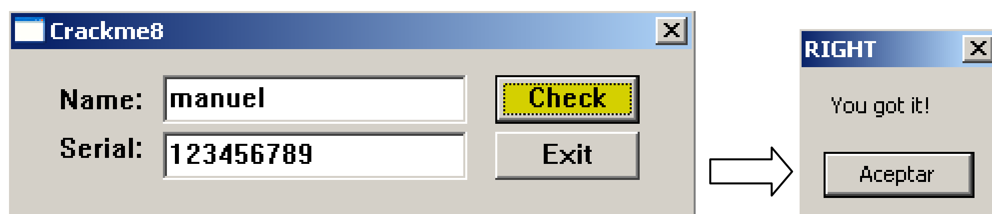


Por último nos queda parchear el programa. Reiniciamos la aplicación, buscamos el Breakpoint a través de la ventana (B), nos situamos encima, pulsamos la barra espaciadora y cambiamos la instrucción del salto por NOP. De esta forma nos aseguramos de saltar siempre al “good boy”.

Guardamos el binario modificado como un ejecutable nuevo en nuestro disco duro.

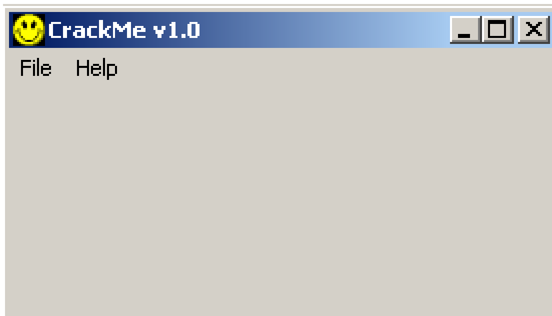


Hacemos doble clic sobre el parche e introducimos un nombre y serial cualquiera:

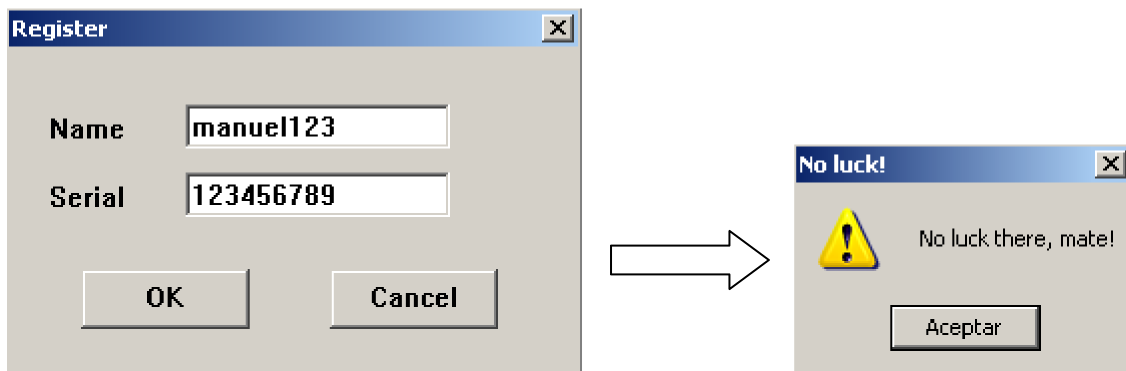


7.5 Caso práctico 5: Marco de referencias

Abrimos Olly y cargamos carckme3.exe. Una vez cargado se detendrá en el "Entry Point". Pulsamos F9 para ejecutar el programa y ver su comportamiento.



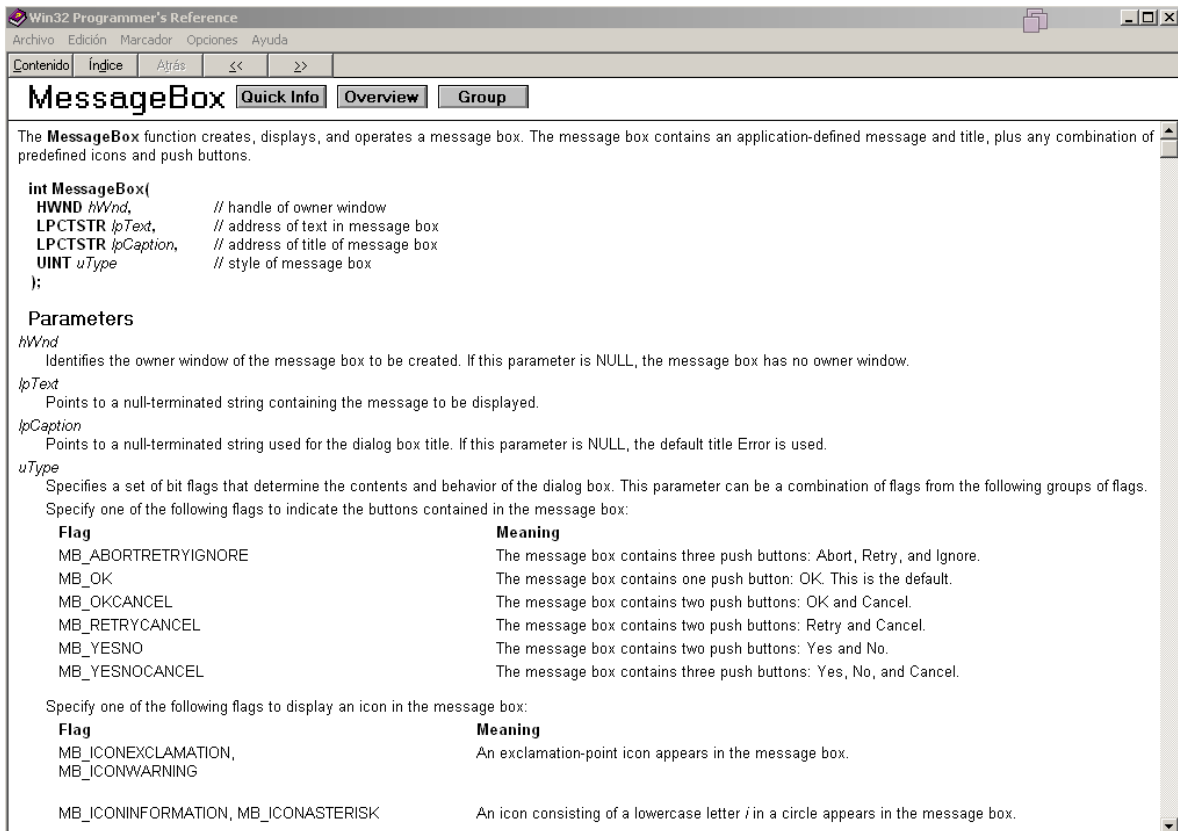
Seleccionamos "Help" -> "Register". Introducimos un nombre y serial para ver la respuesta del programa.



En programas tan pequeños como el que estamos analizando a veces basta con bajar unas cuantas páginas para encontrar algo interesante.

```
OllyDbg - Crackme3.EXE - [CPU - main thread, module Crackme3]
C File View Debug Plugins Options Window Help Tools BreakPoint->
Running

0040130E  .  53              PUSH EBX
0040130F  .  56              PUSH ESI
00401310  .  57              PUSH EDI
00401311  .  817D 0C 11010000 CMP [ARG.2],111
00401318  .v 74 12           JE SHORT Crackme3.0040132C
0040131A  .  837D 0C 10      CMP [ARG.2],10
0040131E  .v 74 15           JE SHORT Crackme3.00401335
00401320  .  B8 00000000     MOV EAX,0
00401325  >  5F              POP EDI                                  0013F254
00401326  .  5E              POP ESI                                  0013F254
00401327  .  5B              POP EBX                                  0013F254
00401328  .  C9              LEAVE
00401329  .  C2 1000         RETN 10
0040132C  >  817D 10 F2030000 CMP [ARG.3],3F2
00401333  .v 75 11           JNZ SHORT Crackme3.00401346
00401335  >  6A 00           PUSH 0                                   Result = 0
00401337  .  FF75 08         PUSH [ARG.1]                             hWnd = 7C91EE18
0040133A  .  E8 73010000     CALL <JMP.&USER32.EndDialog>             EndDialog
0040133F  .  B8 01000000     MOV EAX,1
00401344  .^ EB DF           JMP SHORT Crackme3.00401325
00401346  >  B8 00000000     MOV EAX,0
0040134B  .^ EB D8           JMP SHORT Crackme3.00401325
0040134D  $  6A 30           PUSH 30                                  Style = MB_OK|MB_ICONEXCLAMATION|MB_APPLMODAL
0040134F  .  68 29214000     PUSH Crackme3.00402129                   Title = "Good work!"
00401354  .  68 34214000     PUSH Crackme3.00402134                   Text = "Great work, mate!\rNow try the next CrackMe!"
00401359  .  FF75 08         PUSH [ARG.1]                             hOwner = 7C91EE18
0040135C  .  E8 D9000000     CALL <JMP.&USER32.MessageBoxA>           MessageBoxA
00401361  .  C3              RETN
00401362  $  6A 00           PUSH 0                                   BeepType = MB_OK
00401364  .  E8 AD000000     CALL <JMP.&USER32.MessageBeep>           MessageBeep
00401369  .  6A 30           PUSH 30                                  Style = MB_OK|MB_ICONEXCLAMATION|MB_APPLMODAL
0040136B  .  68 60214000     PUSH Crackme3.00402160                   Title = "No luck!"
00401370  .  68 69214000     PUSH Crackme3.00402169                   Text = "No luck there, mate!"
00401375  .  FF75 08         PUSH [ARG.1]                             hOwner = 7C91EE18
00401378  .  E8 BD000000     CALL <JMP.&USER32.MessageBoxA>           MessageBoxA
0040137D  .  C3              RETN
0040137E  $  8B7424 04       MOV ESI,DWORD PTR SS:[ESP+4]
00401382  .  56              PUSH ESI
00401383  >  8A06            MOV AL,BYTE PTR DS:[ESI]
00401385  .  84C0            TEST AL,AL
00401387  .v 74 13           JE SHORT Crackme3.0040139C
00401389  .  3C 41           CMP AL,41
0040138B  .v 72 1F           JB SHORT Crackme3.004013AC
0040138D  .  3C 5A           CMP AL,5A
0040138F  .v 73 03           JNB SHORT Crackme3.00401394
00401391  .  46              INC ESI
00401392  .^ EB EF           JMP SHORT Crackme3.00401383
00401394  >  E8 39000000     CALL Crackme3.004013D2
```

Hemos encontrado al “good boy” y al “bad boy”. Ambas regiones están delimitadas por la API “MessageBoxA”. Veamos el significado de la API. Seleccionamos la línea y hacemos clic con el botón derecho -> “Help on symbolic names”.



Win32 Programmer's Reference

Archivo Edición Marcador Opciones Ayuda

Contenido | Índice | Atrás | << | >>

MessageBox

Quick Info | Overview | Group

The **MessageBox** function creates, displays, and operates a message box. The message box contains an application-defined message and title, plus any combination of predefined icons and push buttons.

```
int MessageBox(
  HWND hWnd,            // handle of owner window
  LPCTSTR lpText,       // address of text in message box
  LPCTSTR lpCaption,    // address of title of message box
  UINT uType            // style of message box
 );
```

Parameters

hWnd
Identifies the owner window of the message box to be created. If this parameter is NULL, the message box has no owner window.

lpText
Points to a null-terminated string containing the message to be displayed.

lpCaption
Points to a null-terminated string used for the dialog box title. If this parameter is NULL, the default title Error is used.

uType
Specifies a set of bit flags that determine the contents and behavior of the dialog box. This parameter can be a combination of flags from the following groups of flags.

Specify one of the following flags to indicate the buttons contained in the message box:

Flag	Meaning
MB_ABORTRETRYIGNORE	The message box contains three push buttons: Abort, Retry, and Ignore.
MB_OK	The message box contains one push button: OK. This is the default.
MB_OKCANCEL	The message box contains two push buttons: OK and Cancel.
MB_RETRYCANCEL	The message box contains two push buttons: Retry and Cancel.
MB_YESNO	The message box contains two push buttons: Yes and No.
MB_YESNOCANCEL	The message box contains three push buttons: Yes, No, and Cancel.

Specify one of the following flags to display an icon in the message box:

Flag	Meaning
MB_ICONEXCLAMATION, MB_ICONWARNING	An exclamation-point icon appears in the message box.
MB_ICONINFORMATION, MB_ICONASTERISK	An icon consisting of a lowercase letter *i* in a circle appears in the message box.

De la definición podemos sacar la siguiente conclusión: Olly prepara los argumentos para ser pasados a la función junto con la llamada a esa función. Si nos fijamos en el área del "good boy" vemos que:

- El primer argumento corresponde al estilo de la ventana.
- El segundo argumento corresponde al título de la ventana: "Good Work".
- El tercer argumento corresponde al texto de la ventana: "Great work…".
- El cuarto argumento corresponde al controlador del propietario de la ventana.

Por último se llama a "MessageBoxA" a través de la instrucción CALL.

```
PUSH 30                          ┌Style = MB_OK|MB_ICONEXCLAMATION|MB_APPLMODAL
PUSH Crackme3.00402129           │Title = "Good work!"
PUSH Crackme3.00402134           │Text = "Great work, mate!\rNow try the next CrackMe!"
PUSH [ARG.1]                     │hOwner = 7C91EE18
CALL <JMP.&USER32.MessageBoxA>   └MessageBoxA
```

Unas cuantas líneas más arriba vemos varias instrucciones de salto, que determinarán si vamos a parar en el "good boy" o en el "bad boy".

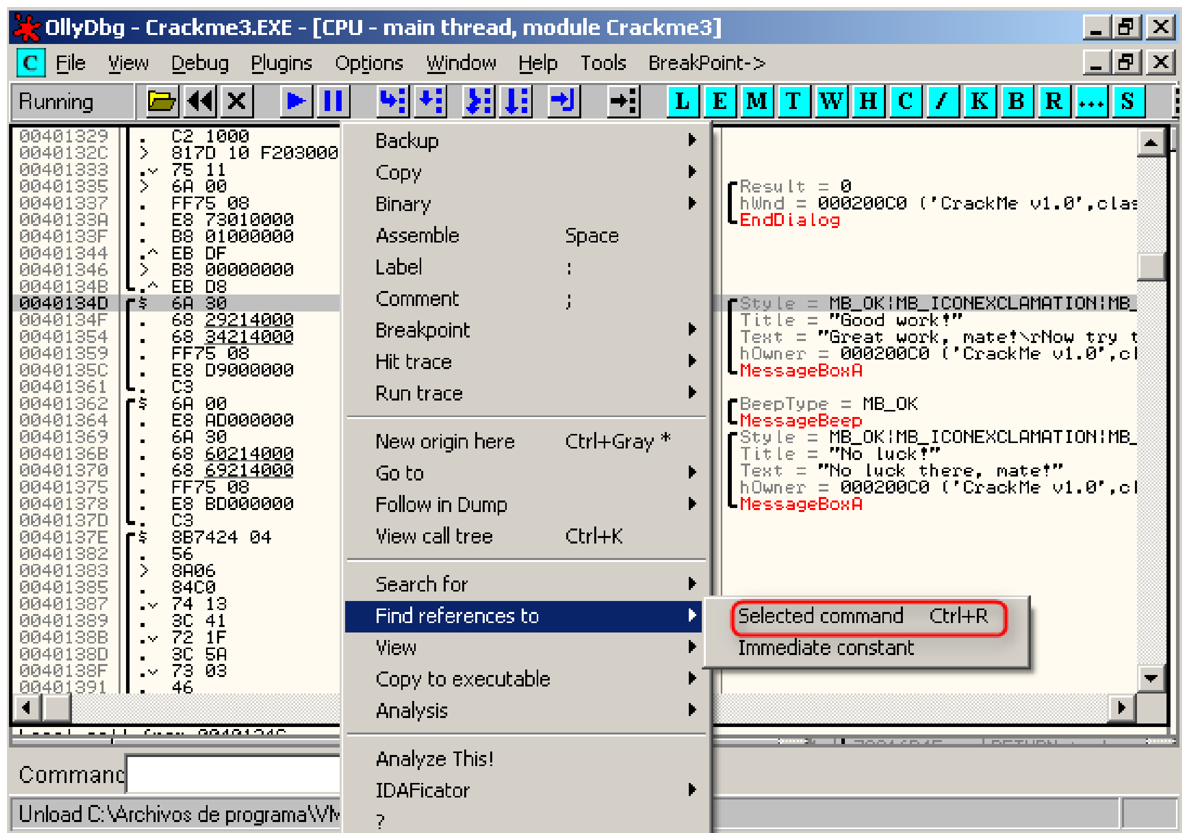
Empezemos por estudiar el primer salto en la dirección 40134B. Si nos situamos encima, vemos que no nos lleva ni al "good boy" ni al "bad boy" sino a la dirección contraria.

La siguiente instrucción de salto en la dirección 401344, salta hacia la misma dirección que el primero.

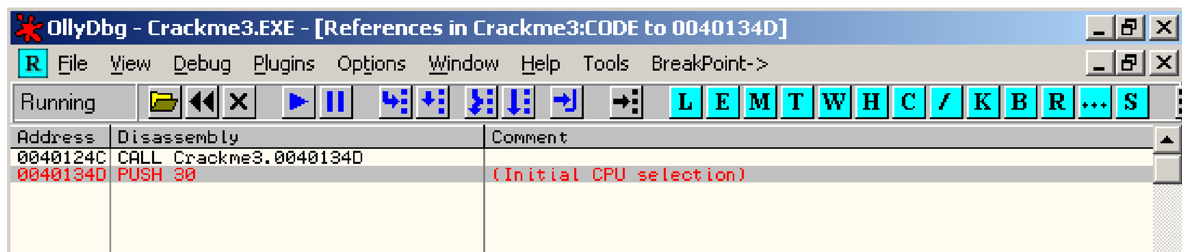
Si nos fijamos bien vemos que ambas instrucciones pertenecen a una misma rutina. Entre las dos primeras columnas vemos varias líneas negras que parecen separar ‘algo’. Pues bien Olly incorpora esas líneas para separar funciones diferentes. (Esta caracteristica de Olly no es aplicable a todos los programas, ya que a veces Olly es incapaz de diferenciar las distintas funciones).

En nuestro caso vemos que ambas instrucciones de salto estudiadas anteriormente pertenecen a la misma función. Y como ninguna de ellas salta hacia el “good boy” o el “bad boy”, no son de gran ayuda. Otra cosa que podemos observar de la figura anterior es que el “good boy” no pertenece a la misma función que el “bad boy”. Esto significa que estas funciones serán llamadas desde sitios diferentes donde habrá que tomar una decisión que nos llevará al mensaje bueno o al mensaje malo. Veamos cómo abordar este obstáculo;

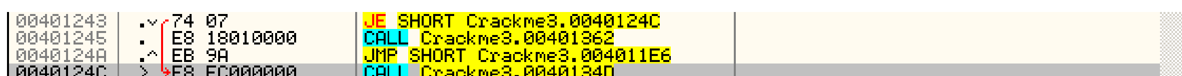
Hacemos clic con el botón derecho en la primera línea de la función correspondiente al mensaje bueno (40134D) y seleccionamos “Find references to” -> “Selected command”.



Se abrirá la ventana de “References”:



Lo que nos muestra esta ventana son todas las referencias (CALL o JMP) que apuntan a la dirección 40134D, nuestro “good boy”. Hacemos doble clic sobre la primera línea.



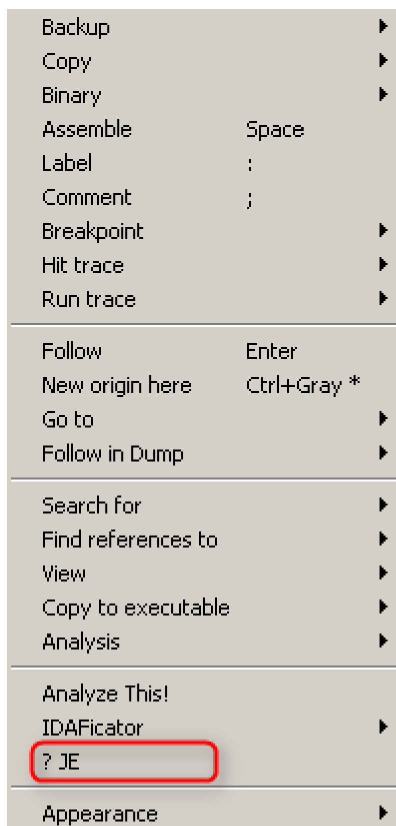
En la línea 40124C podemos ver la instrucción CALL Crackme3.0040134D, donde 40134D es la primera linea del “good boy”. Ponemos un Breakpoint en esta línea y hacemos lo mismo con el “bad boy”.

```
00401243  .v 74 07        JE SHORT Crackme3.0040124C
00401245  .  E8 18010000  CALL Crackme3.00401362
0040124A  .^ EB 9A        JMP SHORT Crackme3.004011E6
0040124C  >  E8 FC000000  CALL Crackme3.0040134D
00401251  .^ EB 93        JMP SHORT Crackme3.004011E6
```

Nos situamos en la primera linea de la función hacia el “bad boy”, la dirección 401362. Hacemos clic con el botón derecho y seleccionamos “Find references to” -> “Selected command”, lo que abrirá la ventana de referencias. Seleccionamos la primera línea y hacemos doble clic, lo que nos situa en la dirección 401245. Vemos que la dirección que llama a nuestro “bad boy” se situa solo a dos lineas del Breakpoint anterior. Pongamos otro Breakpoint en 401245.

```
00401241  .  3BC3         CMP EAX,EBX
00401243  .v 74 07        JE SHORT Crackme3.0040124C
00401245  .  E8 18010000  CALL Crackme3.00401362
0040124A  .^ EB 9A        JMP SHORT Crackme3.004011E6
0040124C  >  E8 FC000000  CALL Crackme3.0040134D
00401251  .^ EB 93        JMP SHORT Crackme3.004011E6
```

Fijemonos que justo por encima del primer CALL aparece la primera instrucción de salto. Situemonos pues en la dirección 401243 con la instrucción JE SHORT Crackme3.0040124C, hacemos clic con el botón derecho y seleccionamos el signo de interrogación para abrir la ayuda mnemonic.



Intel x86 Instructions

Archivo Edición Marcador Opciones Ayuda

Contenido Índice Atrás Imprimir

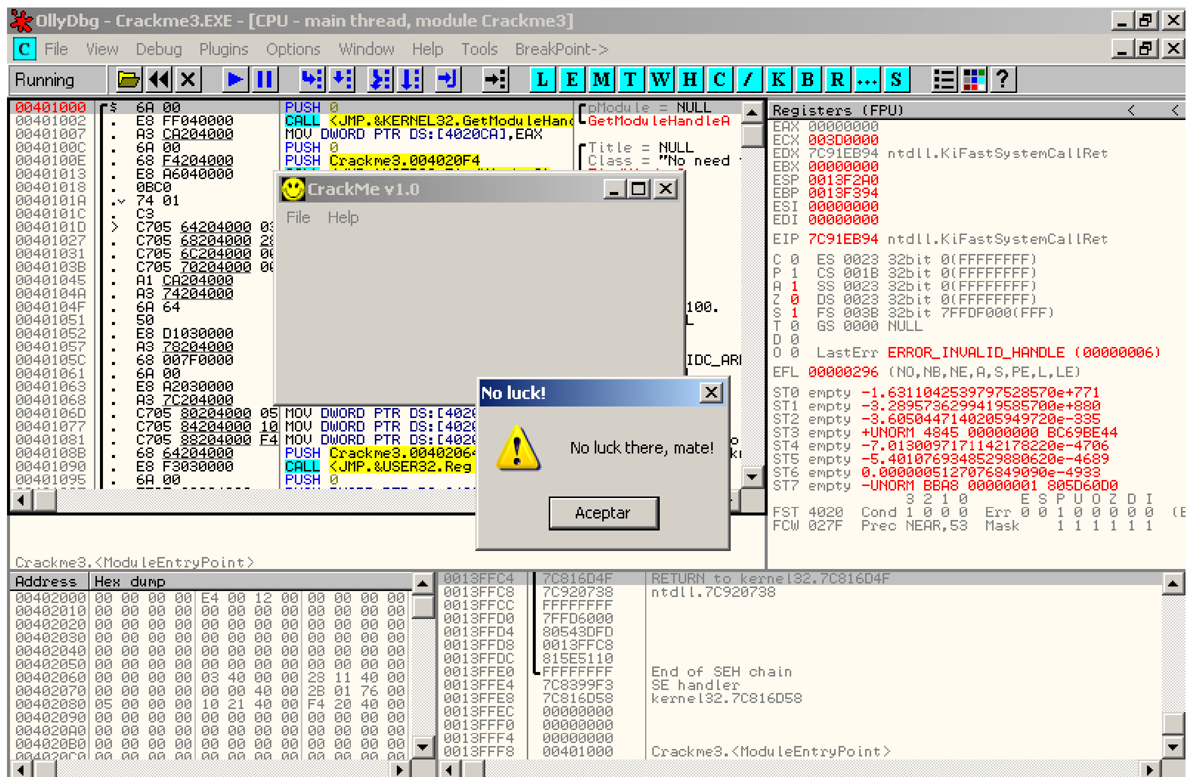
Jcc—Jump if Condition Is Met

See also

Opcode	Instruction	Description
77 *cb*	JA *rel8*	Jump short if above (CF=0 and ZF=0)
73 *cb*	JAE *rel8*	Jump short if above or equal (CF=0)
72 *cb*	JB *rel8*	Jump short if below (CF=1)
76 *cb*	JBE *rel8*	Jump short if below or equal (CF=1 or ZF=1)
72 *cb*	JC *rel8*	Jump short if carry (CF=1)
E3 *cb*	JCXZ *rel8*	Jump short if CX register is 0
E3 *cb*	JECXZ *rel8*	Jump short if ECX register is 0
74 *cb*	JE *rel8*	Jump short if equal (ZF=1)
7F *cb*	JG *rel8*	Jump short if greater (ZF=0 and SF=OF)
7D *cb*	JGE *rel8*	Jump short if greater or equal (SF=OF)
7C *cb*	JL *rel8*	Jump short if less (SF<>OF)
7E *cb*	JLE *rel8*	Jump short if less or equal (ZF=1 or SF<>OF)
76 *cb*	JNA *rel8*	Jump short if not above (CF=1 or ZF=1)
72 *cb*	JNAE *rel8*	Jump short if not above or equal (CF=1)
73 *cb*	JNB *rel8*	Jump short if not below (CF=0)
77 *cb*	JNBE *rel8*	Jump short if not below or equal (CF=0 and ZF=0)
73 *cb*	JNC *rel8*	Jump short if not carry (CF=0)
75 *cb*	JNE *rel8*	Jump short if not equal (ZF=0)
7E *cb*	JNG *rel8*	Jump short if not greater (ZF=1 or SF<>OF)
7C *cb*	JNGE *rel8*	Jump short if not greater or equal (SF<>OF)
7D *cb*	JNL *rel8*	Jump short if not less (SF=OF)
7F *cb*	JNLE *rel8*	Jump short if not less or equal (ZF=0 and SF=OF)
71 *cb*	JNO *rel8*	Jump short if not overflow (OF=0)

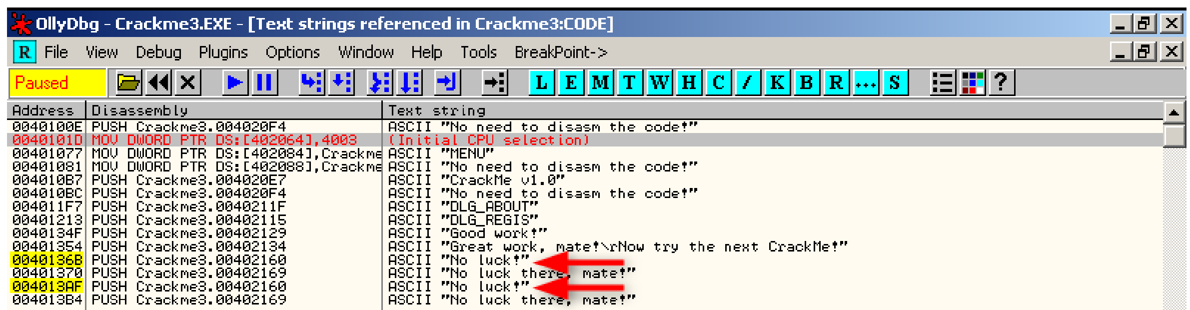
Se abre una nueva ventana donde podemos ver el significado de JE. Jump if Equal (ZF = 1). O lo que es lo mismo, se saltará siempre y cuando la bandera Z = 1. (o cuando las dos cadenas que se comparen son iguales). Sabemos que si dos cadenas son iguales, JE va a tomar el salto. Podemos ver que la instrucción JE salta por encima del “bad boy” hacia el “good boy”. Si por el contrario JE no saltara, iríamos directamente hacia el “bad boy”. Así que queremos tomar este salto para poder hacer la llamada al mensaje bueno.

Reiniciamos y pulsamos F9. En “Help” -> “Register” introducimos nuestras credenciales y hacemos clic en “OK”.

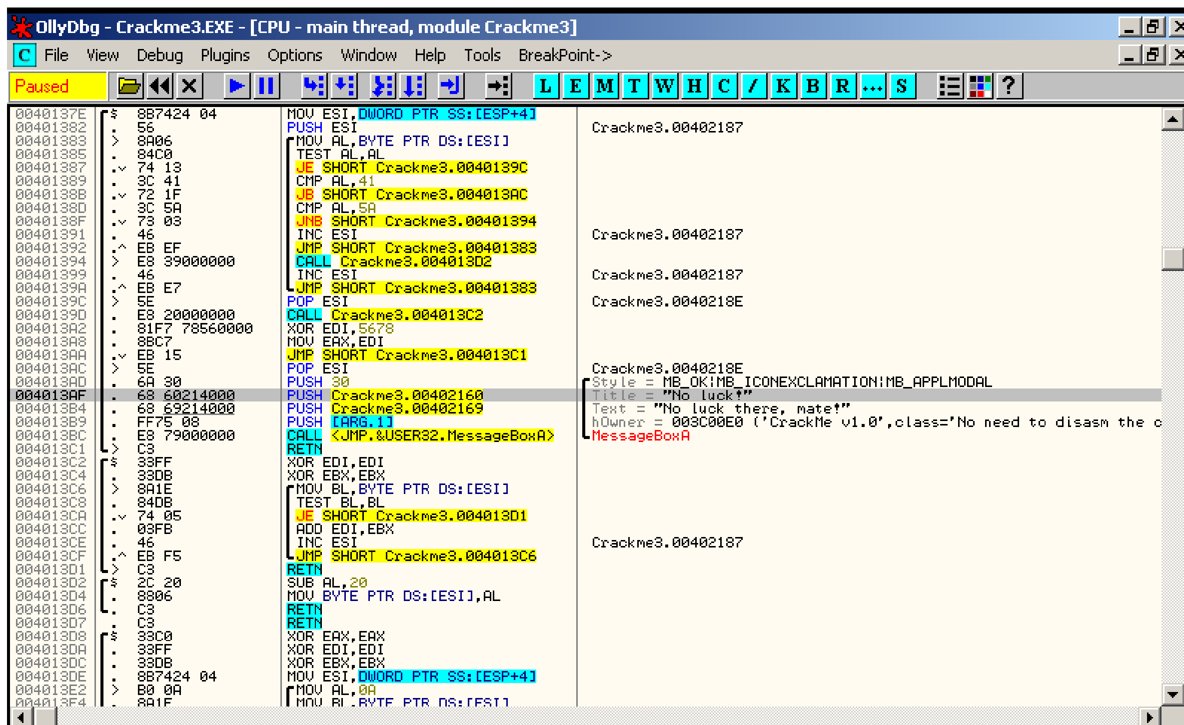


¡ Ha aparecido un "bad boy" sin que el programa se haya detenido en el primer Breakpoint !

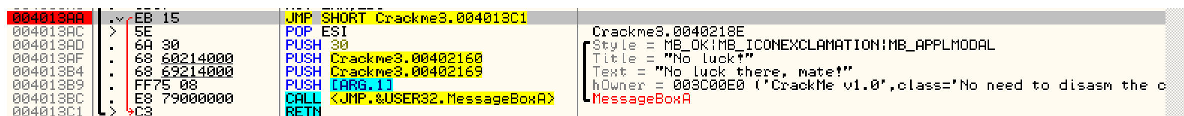
Reiniciemos la aplicación y busquemos por cadenas de texto:



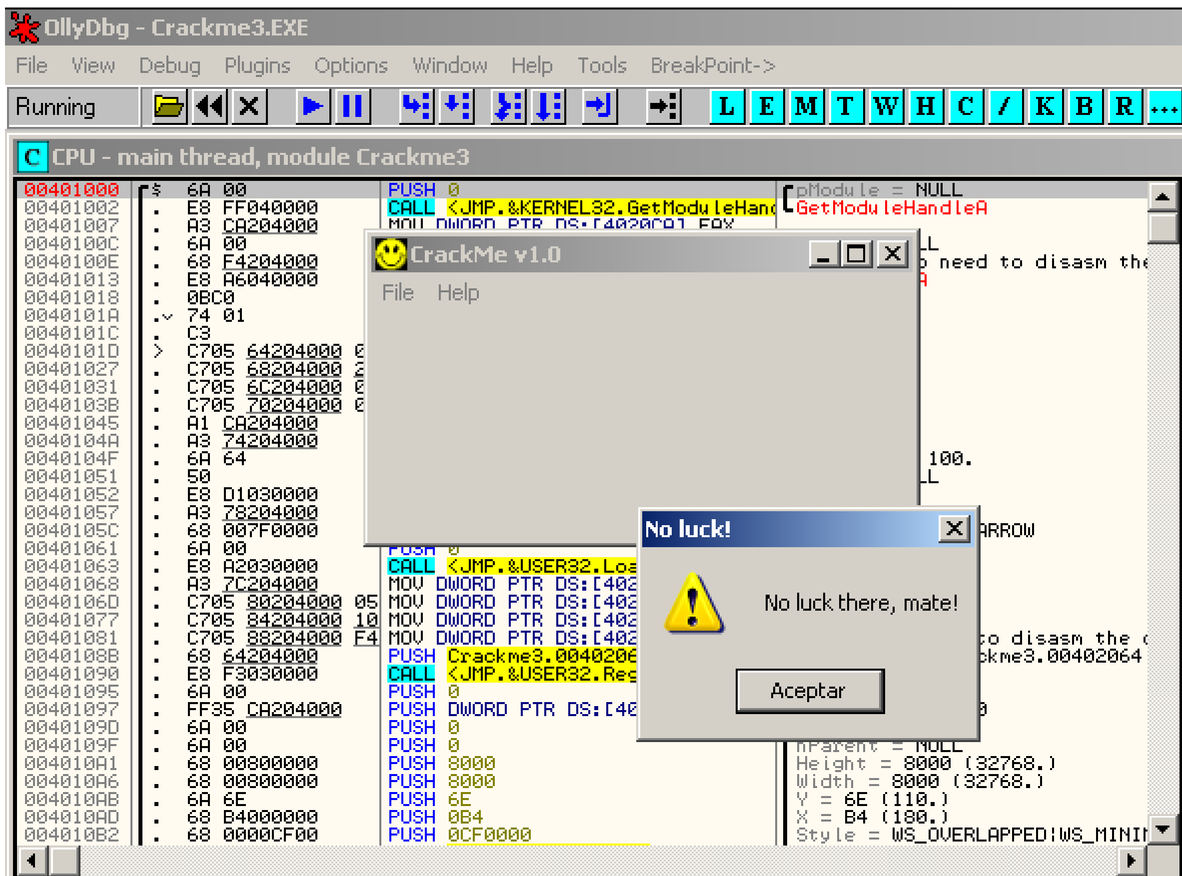
Vemos que hay **dos** "bad boys" y solo un "good boy". Hemos comprobado que la dirección 40136B va hacia el "bad boy", ¿pero qué pasa si hacemos doble clic sobre la dirección 4013AF?



Este “bad boy” se encuentra en una sección completamente diferente de la memoria. La instrucción de salto más próxima está en la dirección 4013AA, y si nos situamos encima de ella Olly nos muestra una flecha roja cuyo destino está fuera del área del mensaje malo.

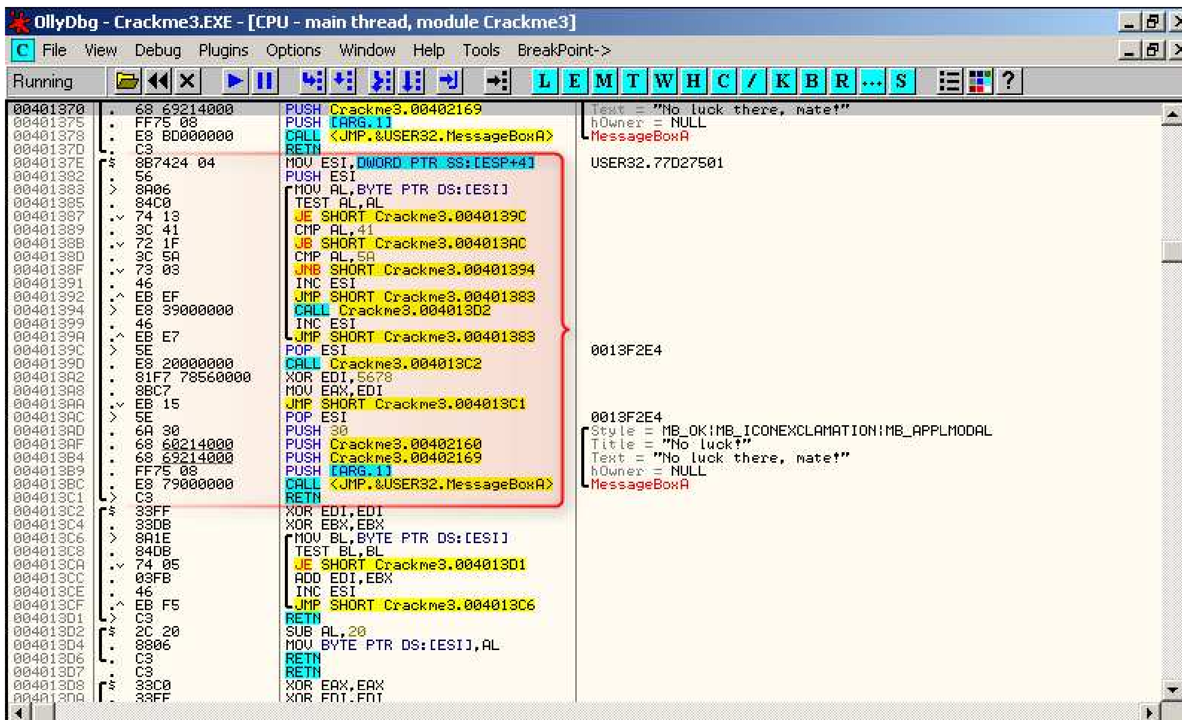


Ponemos un Breakpoint en la instrucción del salto, reiniciamos la aplicación y pulsamos F9. Introducimos nuestro nombre y contraseña.



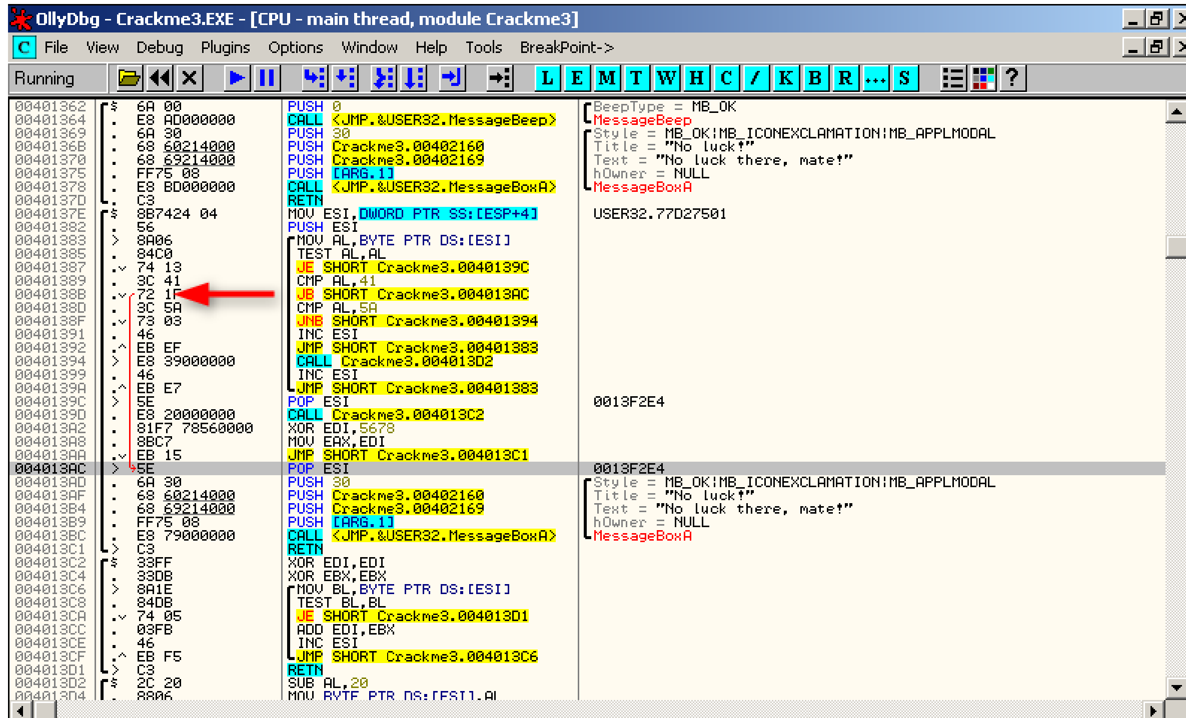
¡ Parace que tenemos que profundicar aún más en el estudio del código !

Centremonos en la sección del código que está delimitada por una barra negra en la columna de los opcodes (Sabemos del analisis anterior que se trata del principio y el fin de una función):

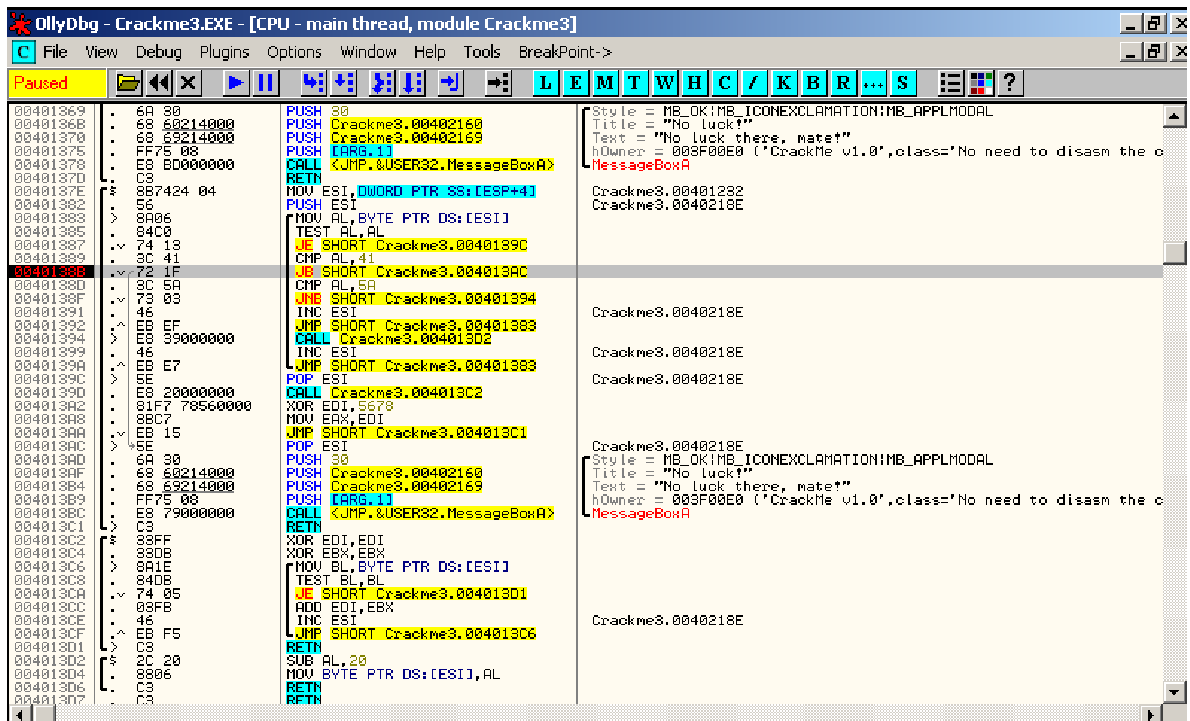


En la primera parte vemos un loop que se inicia con la instrucción TEST AL, AL. Esta instrucción compara si AL es igual a cero. A continuación sigue comparando AL con diferentes caracteres hexadecimales y según el resultado de esa comparación se realizará un salto o no.

Averiguemos primero quien está llamando a nuestro “bad boy”. Para ello hacemos clic en la dirección 4013AC, y vemos como aparece una flecha roja cuyo origen está en la dirección 40138B.

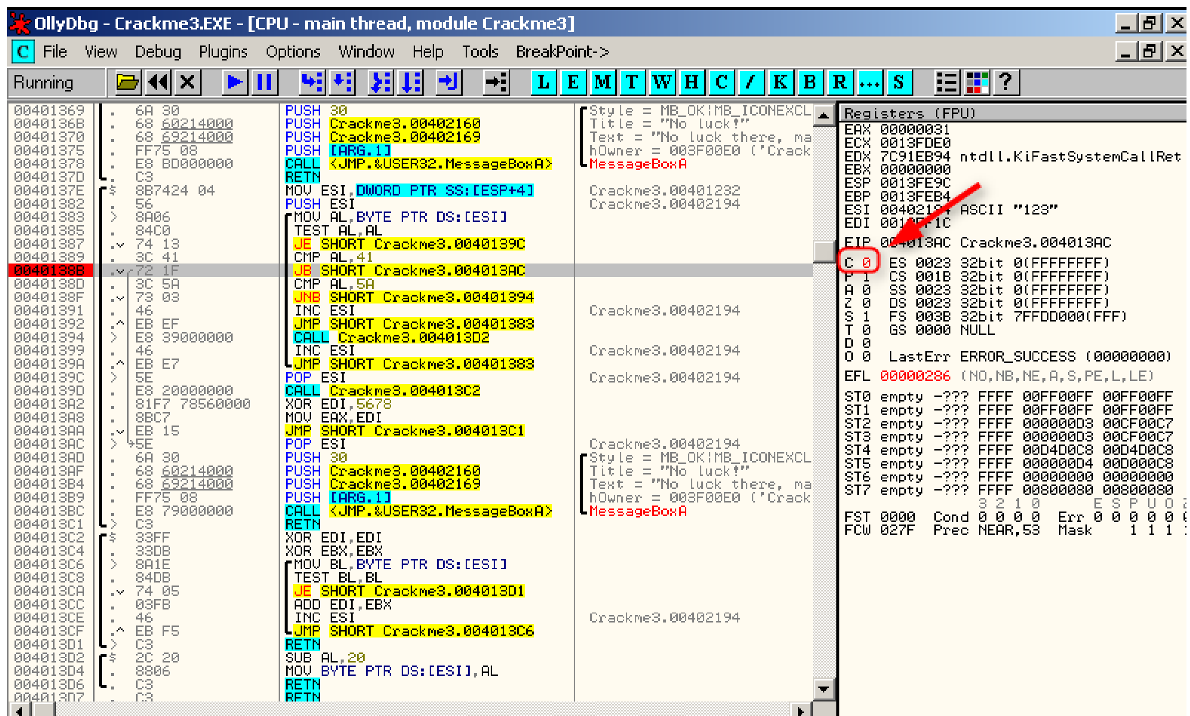


Ponemos un Breakpoint, reiniciamos la aplicación y pulsamos F9. Introducimos nuestro nombre y contraseña.



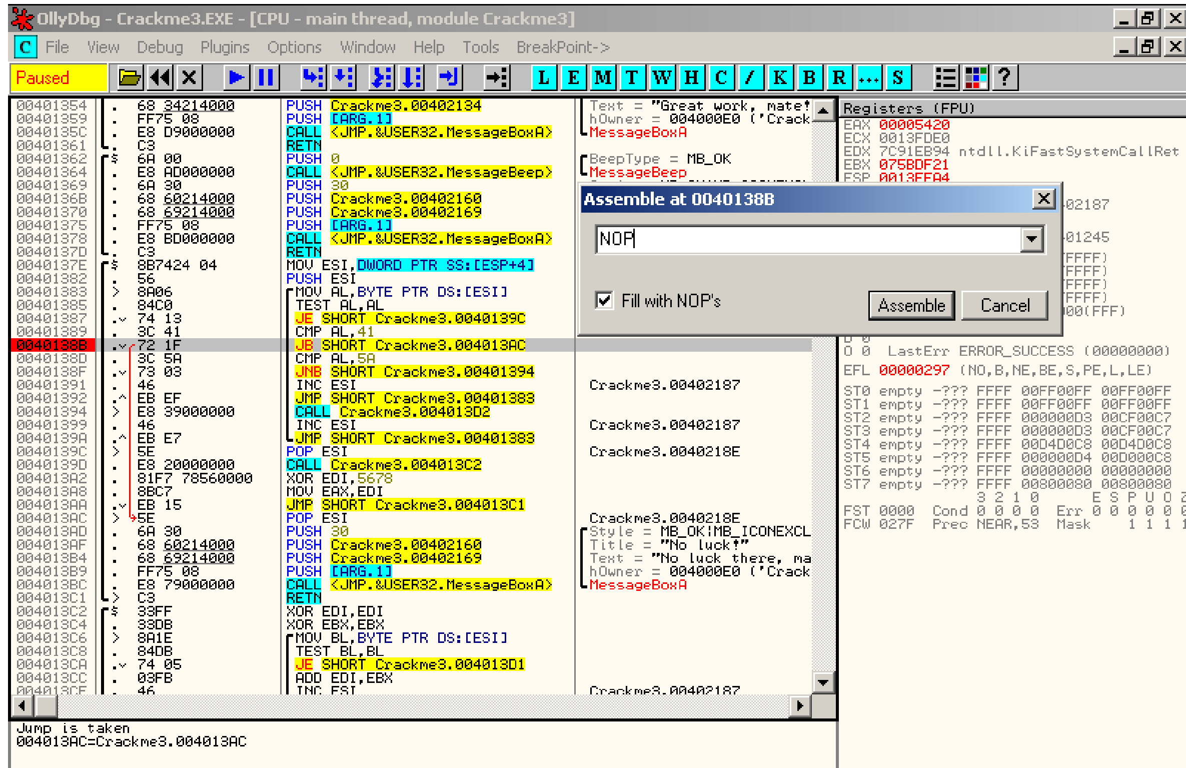
Vemos como Olly se detiene en nuestro Breakpoint. (¡Vamos por buen camino!)

El color de la flecha se volvio gris lo que nos indica que no vamos a saltar hacia el “bad boy”. Como estamos detenidos dentro de un loop, pulsamos F9 hasta ejecutar todas las iteraciones. En algunas vueltas, la flecha cambiará al color rojo y como estamos ante una instrucción JB bastará con cambiar el valor de la bandera C para devolver el color gris a la flecha y asi asegurarnos de no saltar.



Repetiremos el proceso hasta que Olly salga del loop para detenerse en el Breakpoint situado en 401245. Llegados a este punto sabemos que hemos pasado el primer “bad boy”, asi que podemos parchearlo para no tener que preocuparnos más por el .

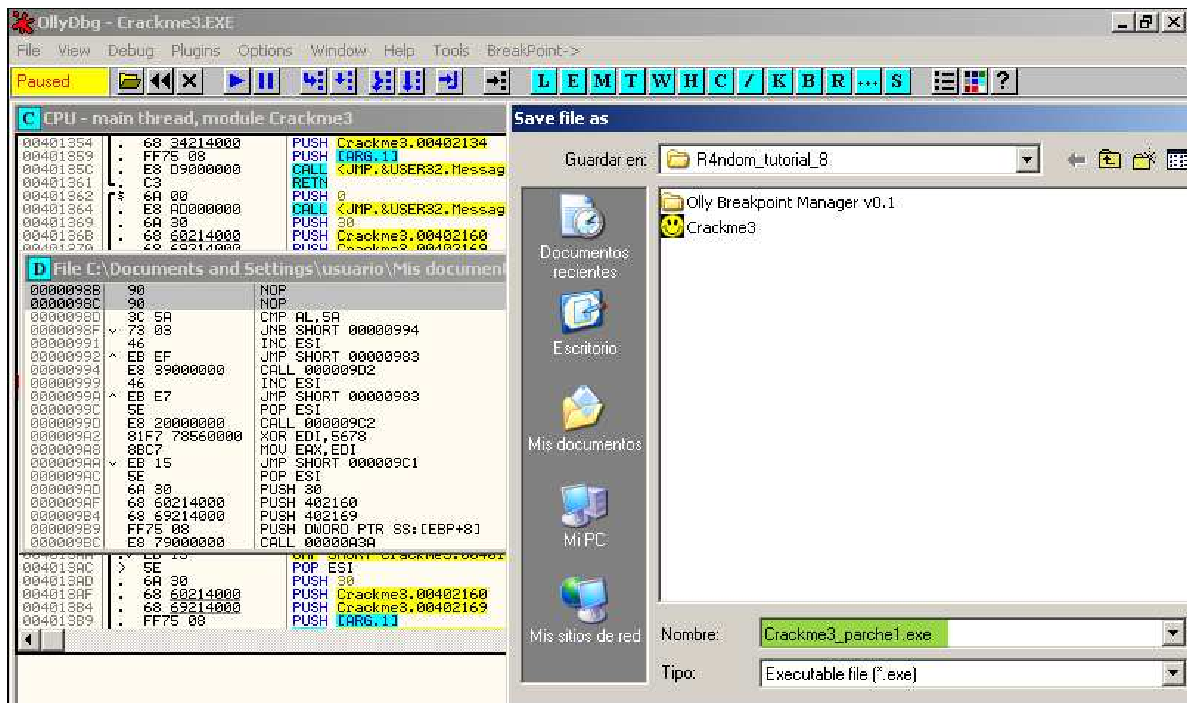
Volvamos a la dirección del Breakpoint 40138B y cambiemos la instrucción JB por NOP de forma que no tengamos que saltar.



Hacemos clic en “Assemble” y en “Cancel”.

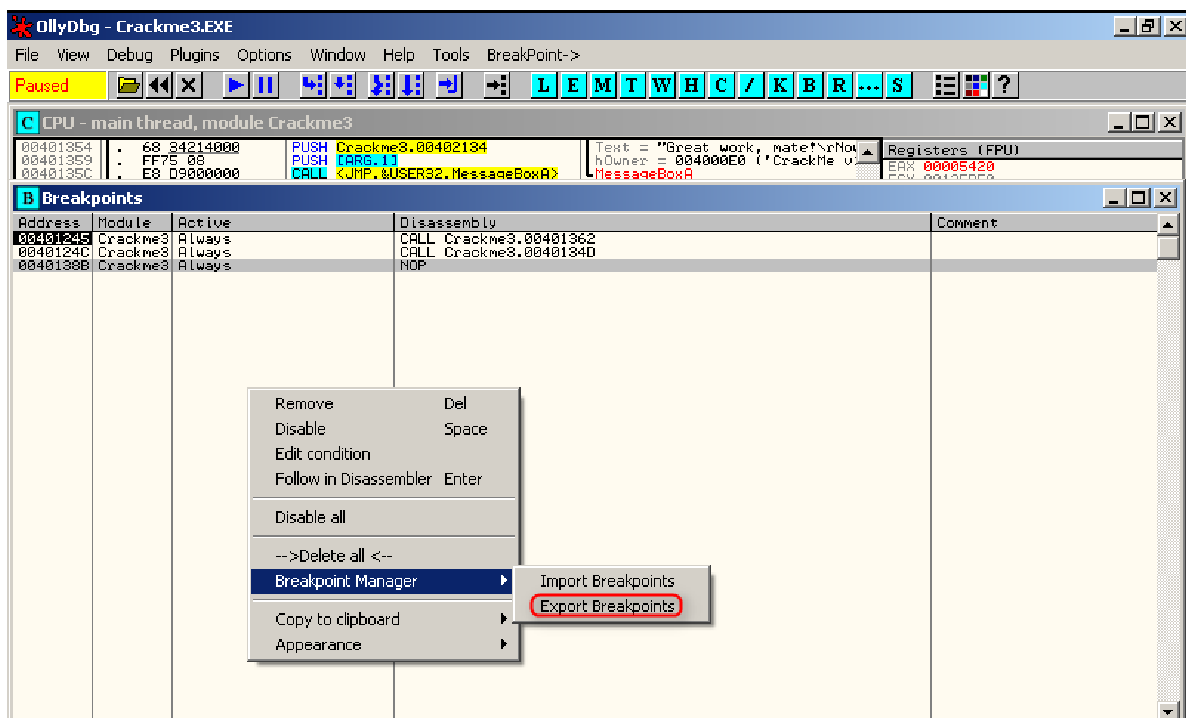
```
OllyDbg - Crackme3.EXE - [CPU - main thread, module Crackme3]
C File View Debug Plugins Options Window Help Tools BreakPoint->
Paused                                               L E M T W H C / K B

00401354  .  68 34214000     PUSH Crackme3.00402134                  Text = "Great work, mate!"
00401359  .  FF75 08         PUSH [ARG.1]                            hOwner = 004000E0 ('Crack
0040135C  .  E8 D9000000     CALL <JMP.&USER32.MessageBoxA>          MessageBoxA
00401361  .  C3              RETN
00401362  $  6A 00           PUSH 0                                  BeepType = MB_OK
00401364  .  E8 AD000000     CALL <JMP.&USER32.MessageBeep>          MessageBeep
00401369  .  6A 30           PUSH 30                                 Style = MB_OK|MB_ICONEXCL
0040136B  .  68 60214000     PUSH Crackme3.00402160                  Title = "No luck!"
00401370  .  68 69214000     PUSH Crackme3.00402169                  Text = "No luck there, ma
00401375  .  FF75 08         PUSH [ARG.1]                            hOwner = 004000E0 ('Crack
00401378  .  E8 BD000000     CALL <JMP.&USER32.MessageBoxA>          MessageBoxA
0040137D  .  C3              RETN
0040137E  $  8B7424 04       MOV ESI,DWORD PTR SS:[ESP+4]
00401382  .  56              PUSH ESI                                Crackme3.00402187
00401383  >  8A06            MOV AL,BYTE PTR DS:[ESI]
00401385  .  84C0            TEST AL,AL
00401387  .v 74 13           JE SHORT Crackme3.0040139C
00401389  .  3C 41           CMP AL,41
0040138B     90              NOP
0040138C     90              NOP
0040138D  .  3C 5A           CMP AL,5A
0040138F  .v 73 03           JNB SHORT Crackme3.00401394
00401391  .  46              INC ESI                                 Crackme3.00402187
00401392  .^ EB EF           JMP SHORT Crackme3.00401383
00401394  >  E8 39000000     CALL Crackme3.004013D2
00401399  .  46              INC ESI                                 Crackme3.00402187
0040139A  .^ EB E7           JMP SHORT Crackme3.00401383
0040139C  >  5E              POP ESI                                 Crackme3.0040218E
0040139D  .  E8 20000000     CALL Crackme3.004013C2
004013A2  .  81F7 78560000   XOR EDI,5678
004013A8  .  8BC7            MOV EAX,EDI
004013AA  .v EB 15           JMP SHORT Crackme3.004013C1
004013AC  >  5E              POP ESI                                 Crackme3.0040218E
004013AD  .  6A 30           PUSH 30                                 Style = MB_OK|MB_ICONEXCL
004013AF  .  68 60214000     PUSH Crackme3.00402160                  Title = "No luck!"
004013B4  .  68 69214000     PUSH Crackme3.00402169                  Text = "No luck there, ma
004013B9  .  FF75 08         PUSH [ARG.1]                            hOwner = 004000E0 ('Crack
004013BC  .  E8 79000000     CALL <JMP.&USER32.MessageBoxA>          MessageBoxA
004013C1  >  C3              RETN
004013C2  $  33FF            XOR EDI,EDI
004013C4  .  33DB            XOR EBX,EBX
004013C6  >  8A1E            MOV BL,BYTE PTR DS:[ESI]
004013C8  .  84DB            TEST BL,BL
004013CA  .v 74 05           JE SHORT Crackme3.004013D1
004013CC  .  03FB            ADD EDI,EBX
```

A continuación hacemos clic con el botón derecho y seleccionamos “Copy to executable” -> “All modifications”. Esto abrirá una nueva ventana en la memoria. Hacemos clic con el botón derecho, seleccionamos “Save file” y lo guardamos como crackme3_prache1.exe.

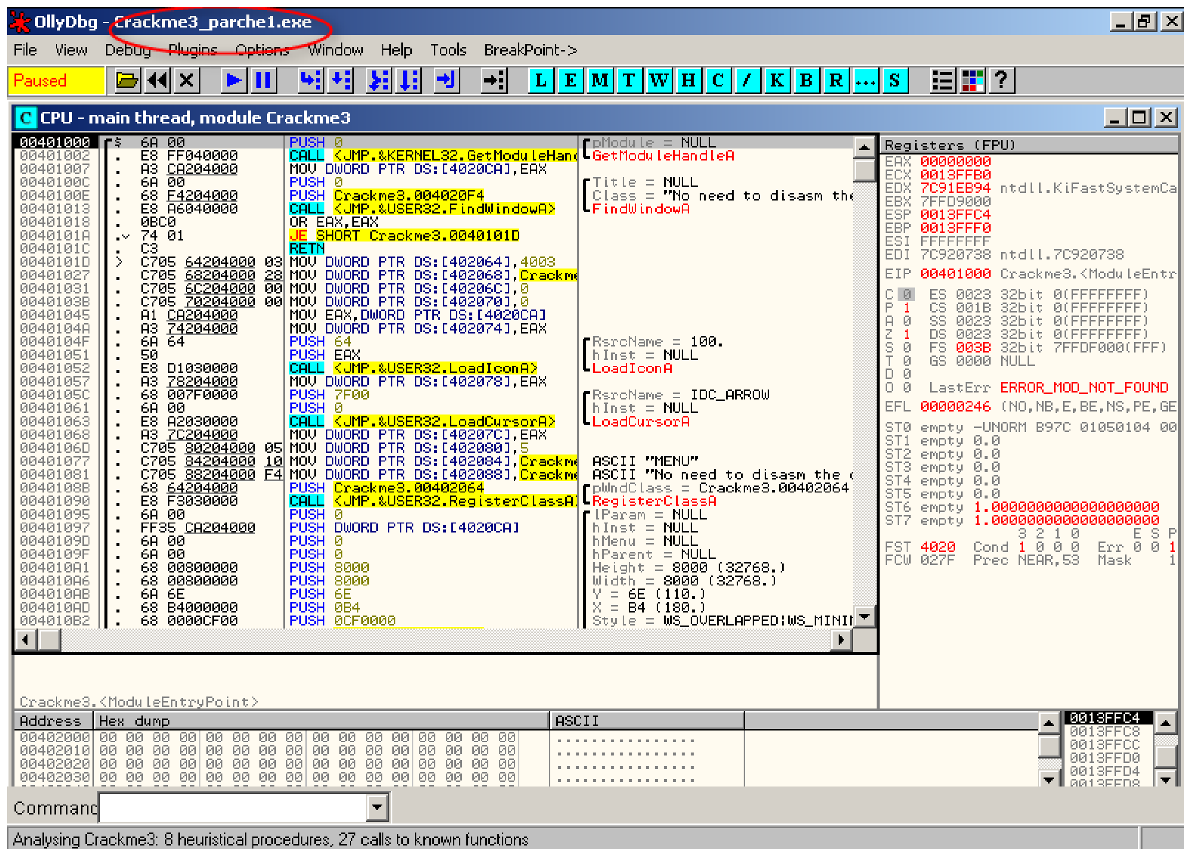


Antes de reiniciar la nueva versión parcheada de la aplicación, tenemos que tener en cuenta que todos los parches, comentarios y Breakpoints van a ser borrados, ya que toda esta información está almacenada en el archivo UDD de Crackme3.udd. Al ejecutar Crackme3_parche1.exe por primera vez, este todavía no va a tener un archivo UDD asociado a él.

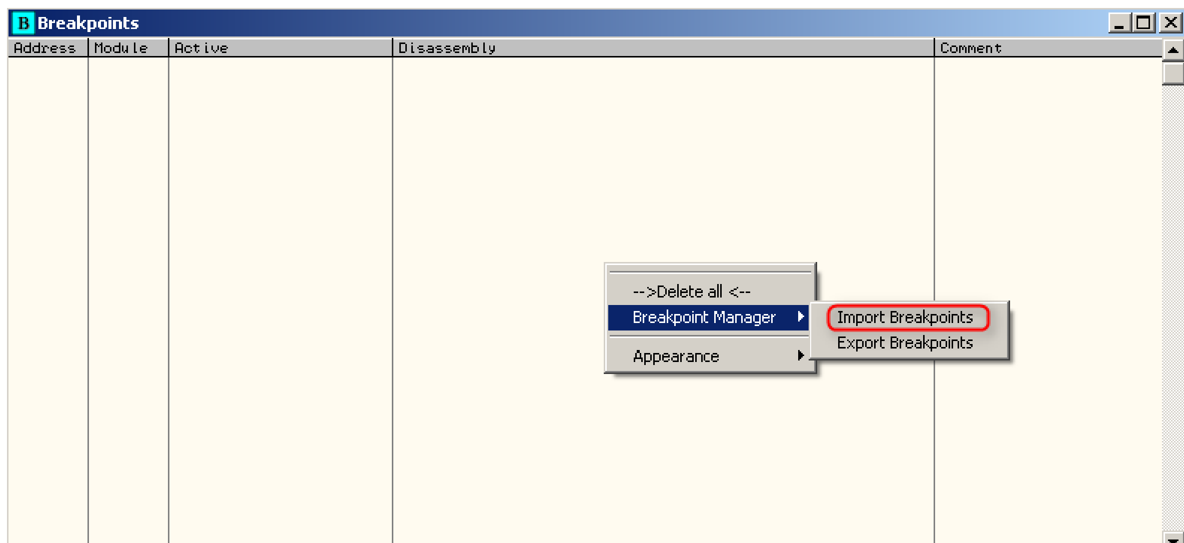
Para solucionar este problema podemos descargarnos el plugin Breakpoint manager. Una vez instalado, abrimos la ventana de los Breakpoints pulsando “B” y con el botón derecho seleccionamos “Breakpoint manager” - > “Export Breakpoints”:



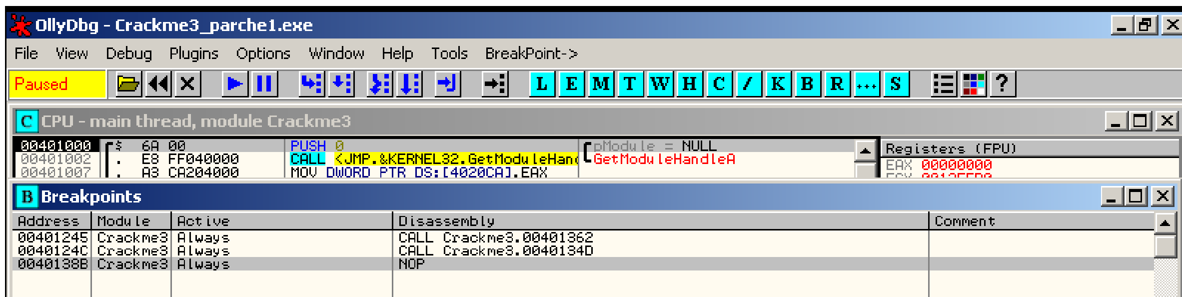
Guardamos el archivo y cargamos la nueva aplicación Crackme3_parche1.exe en Olly:



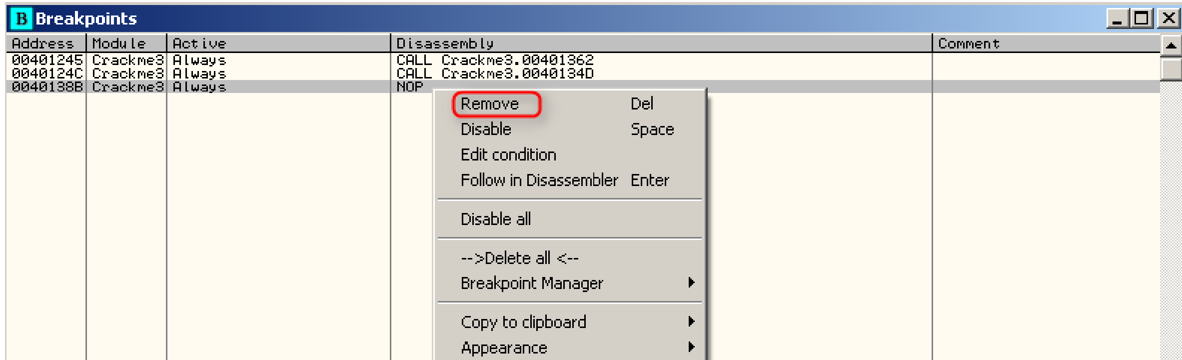
Abrimos la ventana de los Breakpoints (B), hacemos clic con el botón derecho y seleccionamos “Breakpoint Manager” -> “Import Breakpoints”.



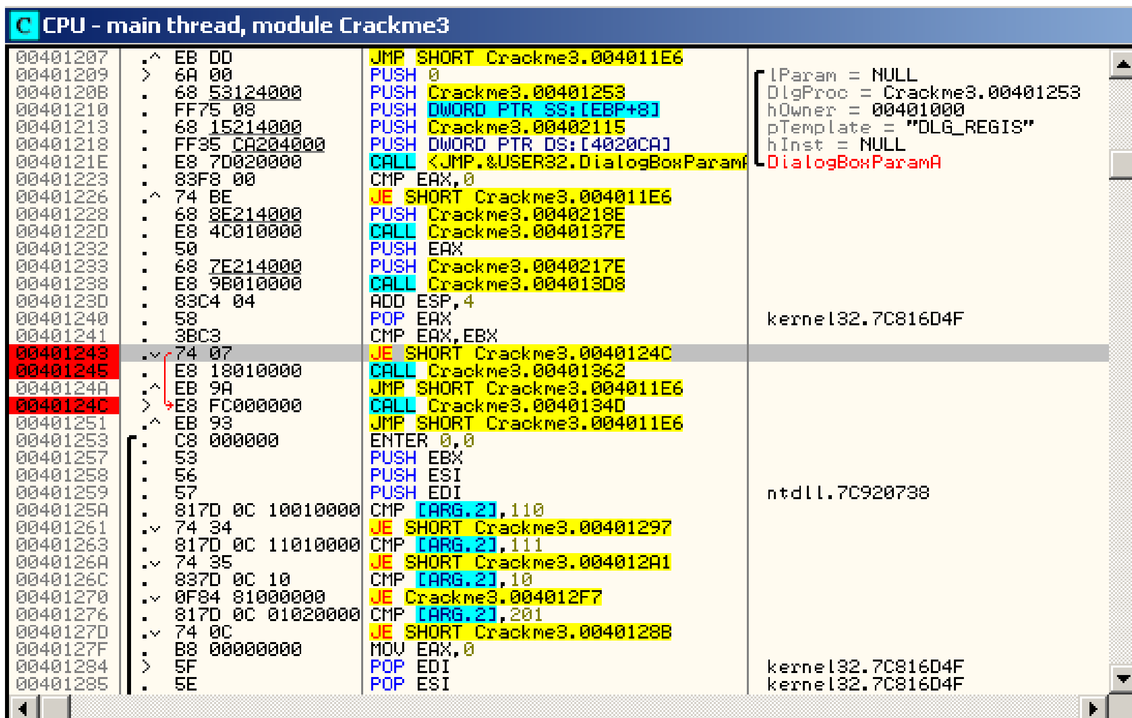
De esta forma importamos todos los Breakpoints a nuestra nueva aplicación Crackme3_parche1.exe:



Vamos a eliminar el Breakpoint que estaba situado dentro del loop, puesto que ya no hace falta:



Y ponemos un Breakpoint en la instrucción JE (401243), situada delante de las dos funciones que llaman al “bad boy” y al “good boy” respectivamente.

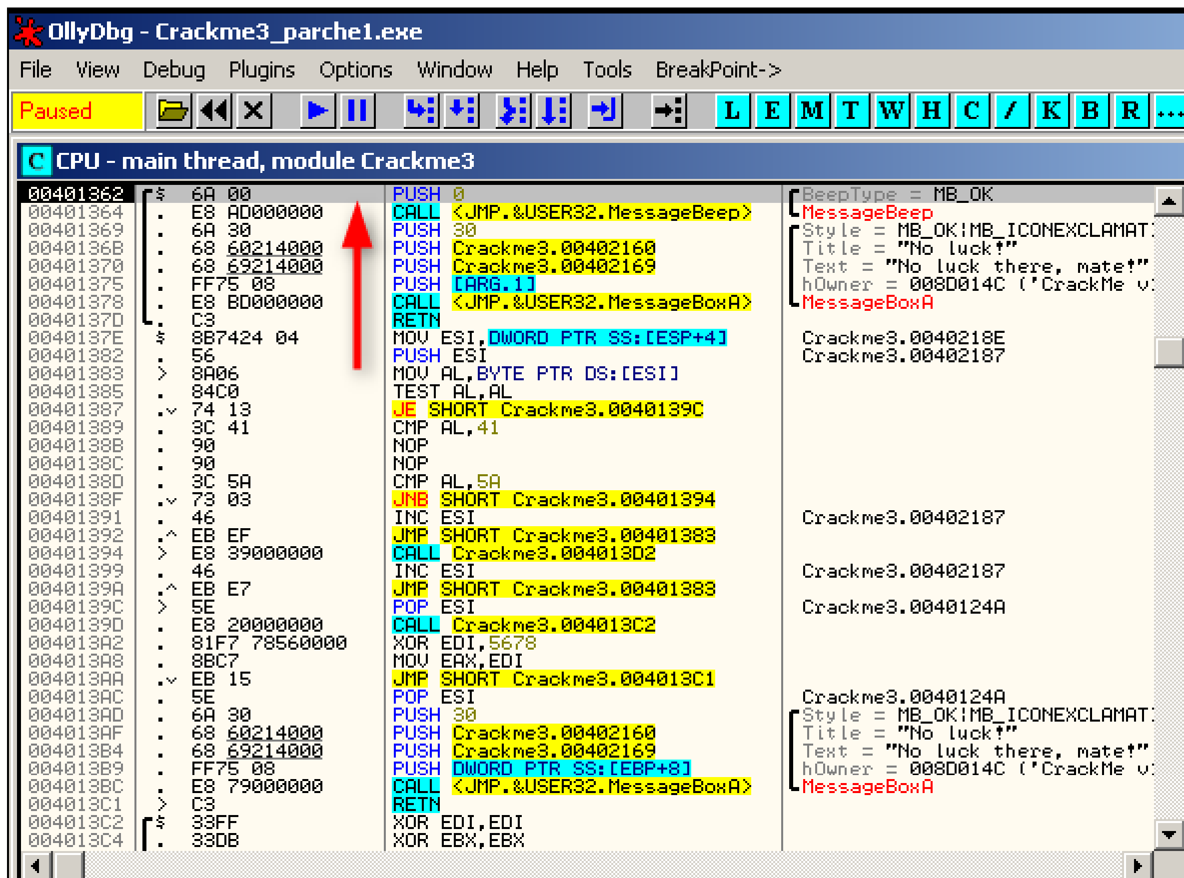


Reiniciamos la aplicación, pulsamos F9 e introducimos el nombre y el serial. Olly se detiene en el Breakpoint que acabamos de poner (401243).

CPU - main thread, module Crackme3

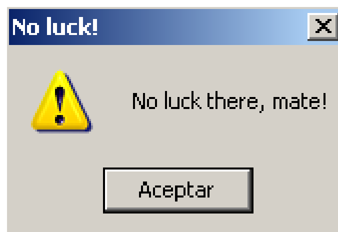
```
00401218  .  FF35 CA204000     PUSH DWORD PTR DS:[4020CA]               hInst = 00400000
0040121E  .  E8 7D020000       CALL <JMP.&USER32.DialogBoxParamA>       DialogBoxParamA
00401223  .  83F8 00           CMP EAX,0
00401226  .^ 74 BE             JE SHORT Crackme3.004011E6
00401228  .  68 8E214000       PUSH Crackme3.0040218E                   ASCII "MANUEL123"
0040122D  .  E8 4C010000       CALL Crackme3.0040137E
00401232  .  50                PUSH EAX
00401233  .  68 7E214000       PUSH Crackme3.0040217E                   ASCII "123456789"
00401238  .  E8 9B010000       CALL Crackme3.004013D8
0040123D  .  83C4 04           ADD ESP,4
00401240  .  58                POP EAX                                  Crackme3.0040218E
00401241  .  3BC3              CMP EAX,EBX
00401243  .v 74 07             JE SHORT Crackme3.0040124C
00401245  .  E8 18010000       CALL Crackme3.00401362
0040124A  .^ EB 9A             JMP SHORT Crackme3.004011E6
0040124C  >  E8 FC000000       CALL Crackme3.0040134D
00401251  .^ EB 93             JMP SHORT Crackme3.004011E6
00401253  .  C8 000000         ENTER 0,0
00401257  .  53                PUSH EBX
00401258  .  56                PUSH ESI
00401259  .  57                PUSH EDI                                 Crackme3.00402187
0040125A  .  817D 0C 10010000  CMP [ARG.2],110
00401261  .v 74 34             JE SHORT Crackme3.00401297
00401263  .  817D 0C 11010000  CMP [ARG.2],111
0040126A  .v 74 35             JE SHORT Crackme3.004012A1
0040126C  .  837D 0C 10        CMP [ARG.2],10
00401270  .v 0F84 81000000     JE Crackme3.004012F7
00401276  .  817D 0C 01020000  CMP [ARG.2],201
0040127D  .v 74 0C             JE SHORT Crackme3.0040128B
0040127F  .  B8 00000000       MOV EAX,0
00401284  >  5F                POP EDI                                  Crackme3.0040218E
00401285  .  5E                POP ESI                                  Crackme3.0040218E
00401286  .  5B                POP EBX                                  Crackme3.0040218E
00401287  .  C9                LEAVE
00401288  .  C2 1000           RETN 10
0040128B  >  6A 01             PUSH 1                                   Erase = TRUE
0040128D  .  6A 00             PUSH 0                                   pRect = NULL
```

Si nos fijamos en el color de la flecha (gris), sabemos que el salto hacia la dirección 40124C no va a ser tomado y que el siguiente movimiento será llamar la función CALL en 401245, que saltará hacia nuestro “bad boy”. Comprobemoslo: Pulsamos F8 una vez y después F7 para saltar dentro de la función CALL y nos situaremos en la primera línea del mensaje malo. Si a continuación presionamos F9, saldrá el “bad boy”.

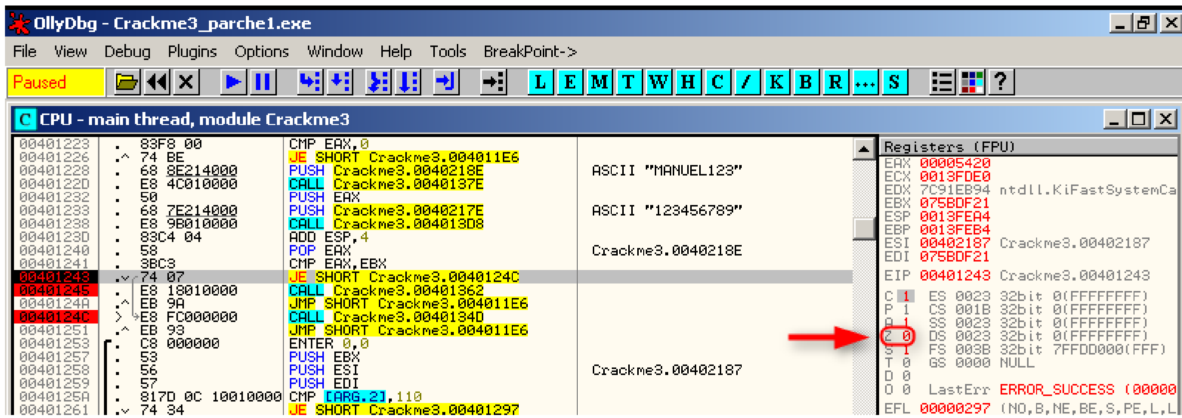


Primera instrucción del "bad boy"

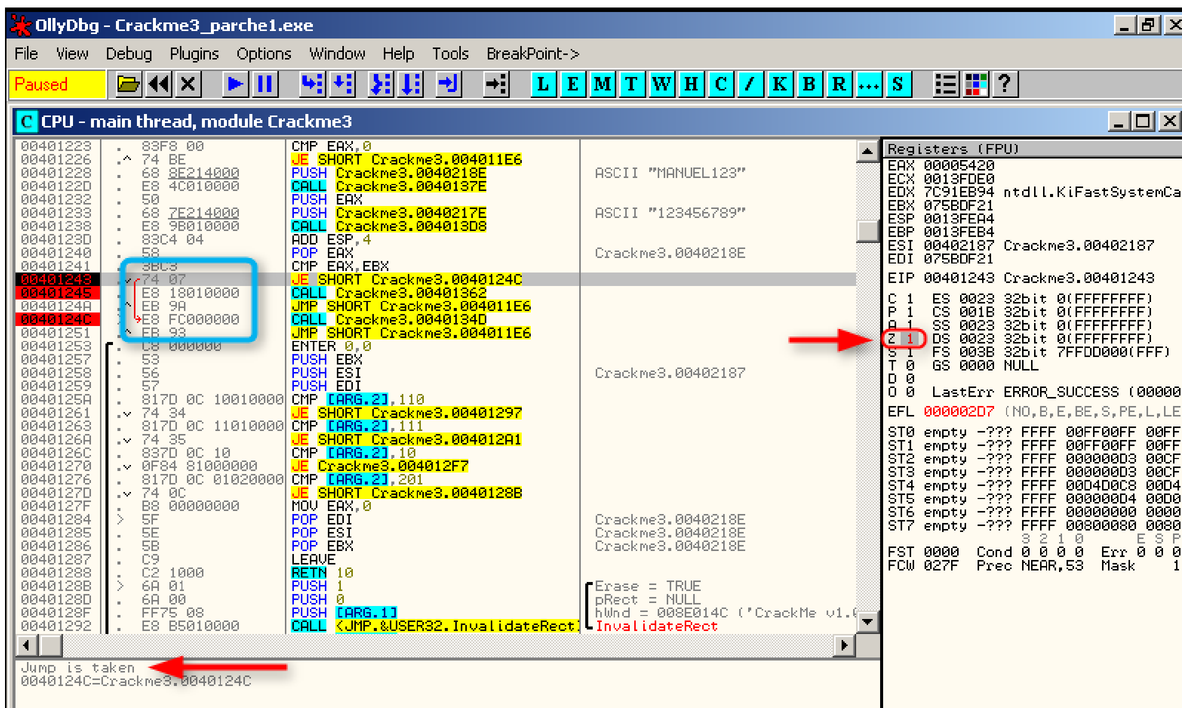
Pulsamos F9:



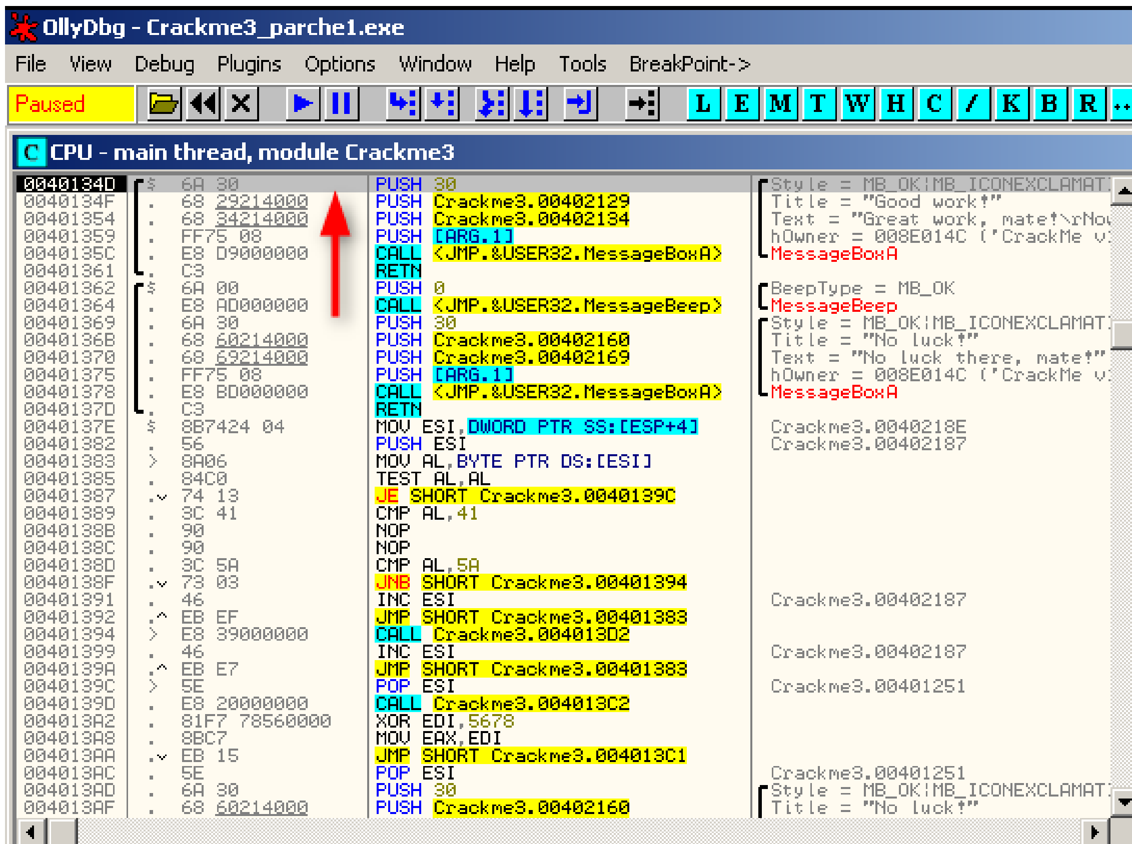
Veamos como solucionar esto: Reiniciamos la aplicación, pulsamos F9 e introducimos el usuario y el serial. Olly se detiene en nuestro primer Breakpoint (401243).



La flecha gris en el opcode nos indica que no vamos a saltar al “good boy”, asi que ayudemos a Olly a efectuar el salto cambiando el valor de la bandera Z haciendo doble clic sobre el cero.

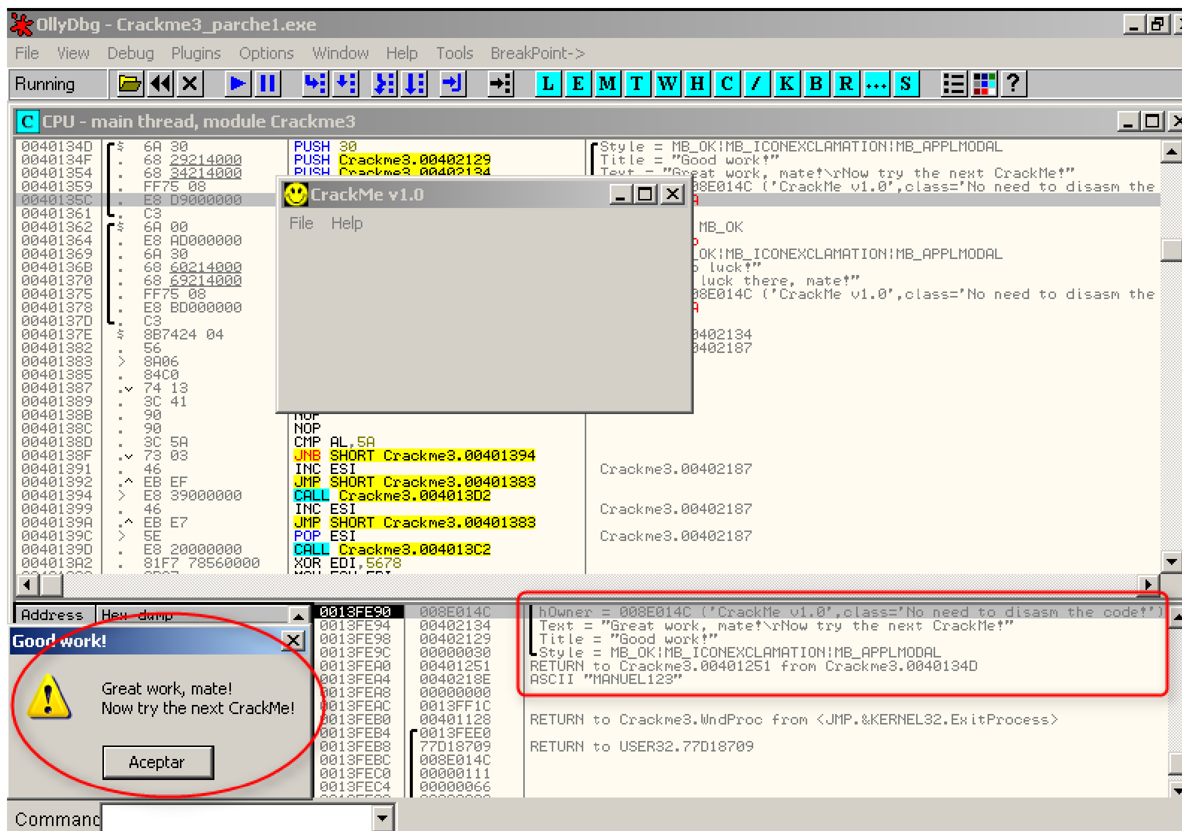


Comprobemoslo pulsando F8 una vez y después F7 para saltar dentro de la función CALL que nos debería llevar a la primera instrucción del “good boy”.



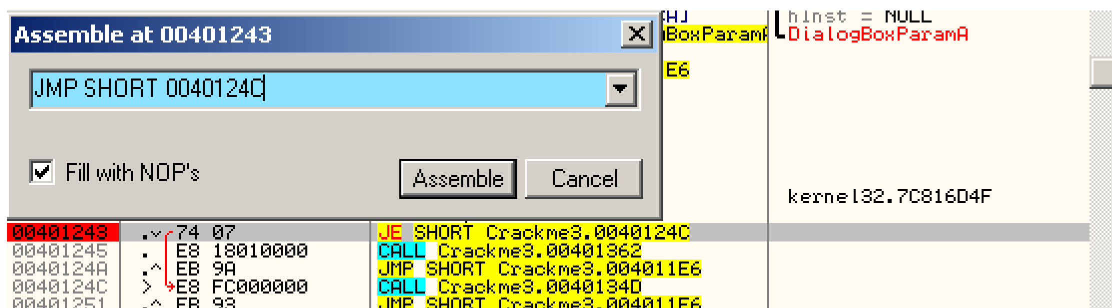
Primera instrucción del mensaje bueno

Si pulsamos F8 unas cuantas veces y nos fijamos en la pila podemos observar como los argumentos de MessageBoxA son empujados en la pila. Tanpronto pasemos la llamada a la función en 40135C, debería aparecer nuestro “good boy”.

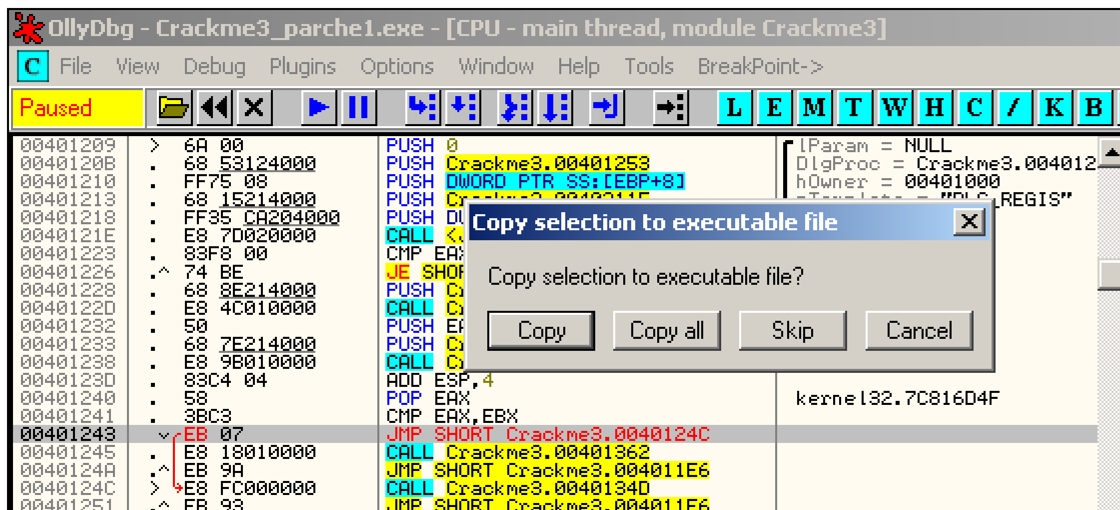


Finalmente parcheamos y guardamos la aplicación en el disco duro.

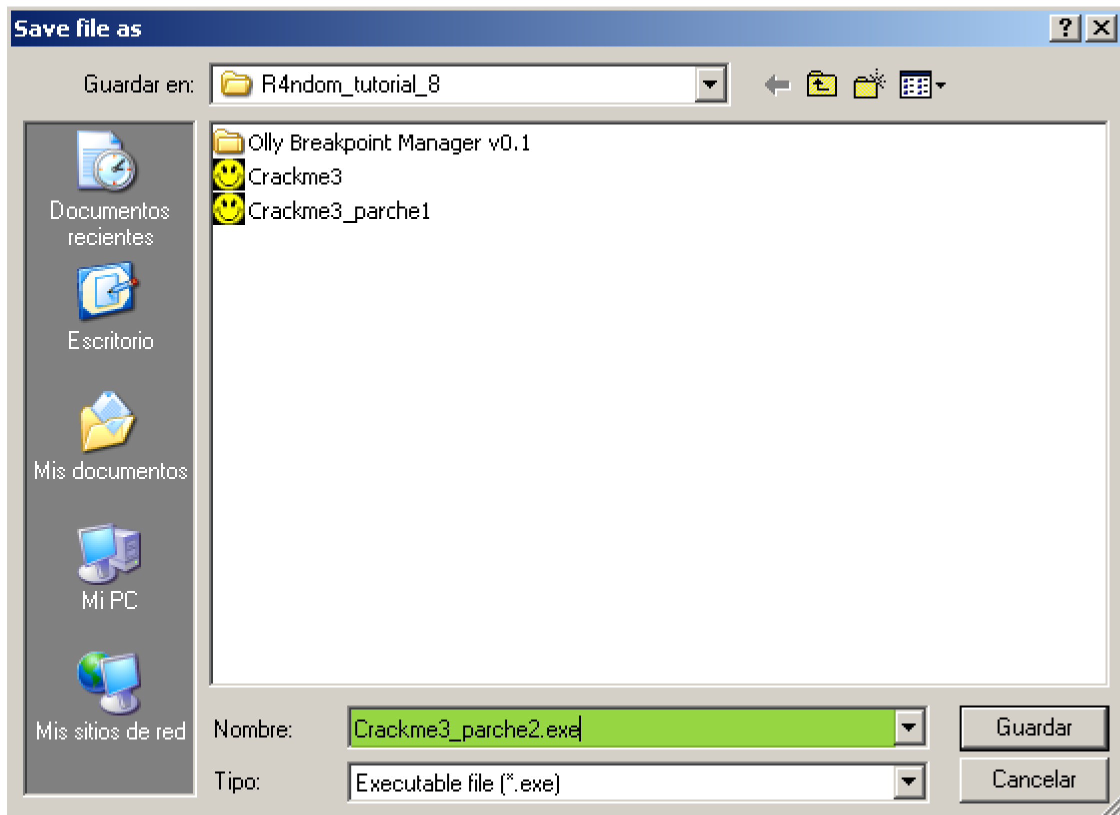
Para parchear la aplicación primero eliminamos los Breakpoints en las instrucciones CALL y luego cambiamos la instrucción JE por JMP (que salte siempre).



Hacemos clic en "Assemble" y "Cancel". Sacamos el Breakpoint y seleccionamos todo.



Guardamos el nuevo ejecutable en el disco duro con el nombre “Crackme3_prache2.exe”



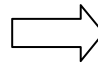
Cerramos Olly y ejecutamos la aplicación.

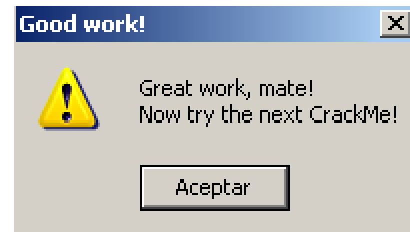
Register

Name manuel123

Serial 123456789

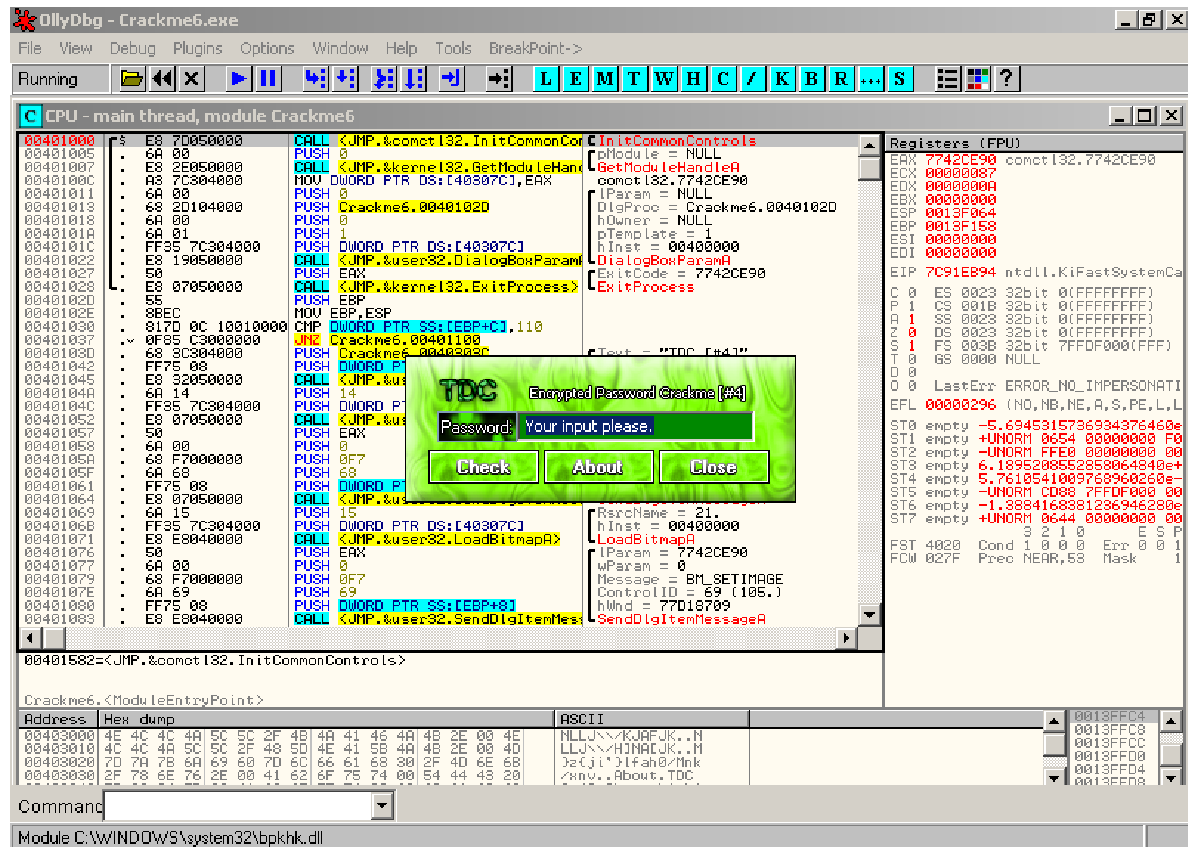
OK Cancel



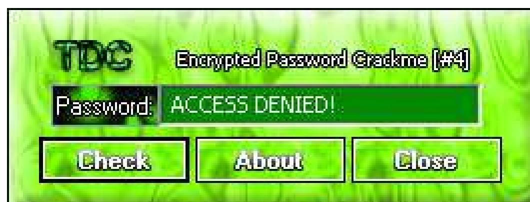


7.6 Caso práctico 6: Intermodular Call

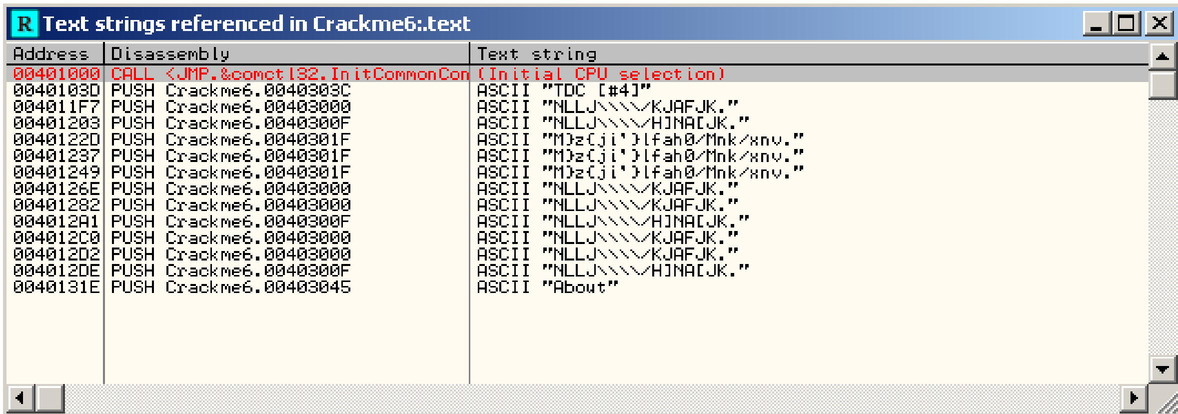
Abrimos Olly y cargamos Crackme6.exe. Pulsamos F9 para averiguar lo que nos sale.



Introducimos una contraseña y pulsamos “Check”.



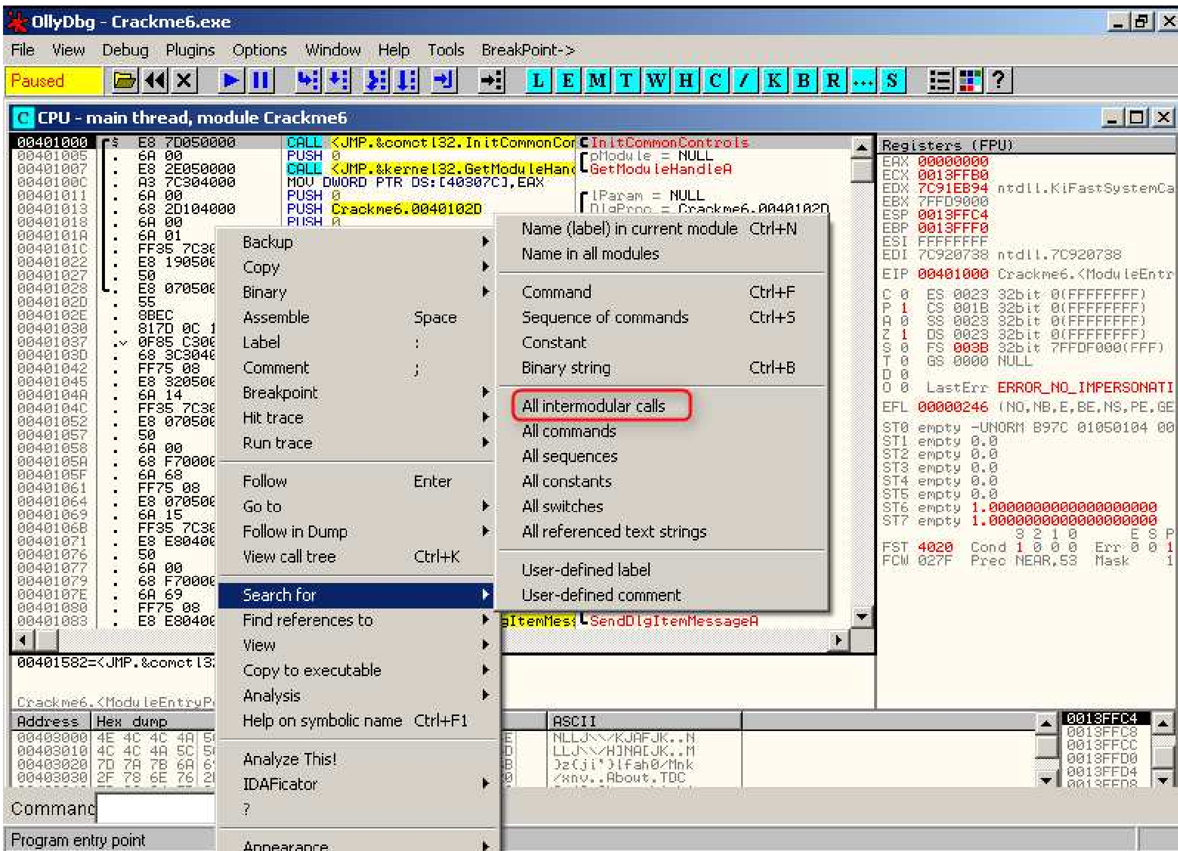
El primer paso será buscar cadenas de texto:



¡Parece que el autor de la aplicación ha cifrado las cadenas de texto!

La mayoría de las aplicaciones de Windows utilizan un conjunto estándar de API's para realizar acciones específicas. Por ejemplo, *MessageBoxA* se llama para mostrar un simple mensaje, *TerminateProcess* se llama cuando la aplicación desea terminar… La mayoría de las aplicaciones utilizan las mismas API's, lo que revierte en nuestro beneficio. Por ejemplo, hay API's para obtener el texto de un cuadro de diálogo (como un nombre de usuario y número de serie). Hay API's para configurar temporizadores (esperar 10 segundos antes de pulsar 'continuar'). Hay funciones que comparan cadenas. Por ejemplo, ¿La contraseña introducida es la misma que la almacenada en el programa? Y hay API's que leen y escriben en los registros (para almacenar y recuperar las credenciales).

Olly tiene una herramienta para buscar estos API's. Haciendo clic con el botón derecho en la ventana de desensamblado, seleccionamos "Search for" -> "All Intermodular calls".



Hacemos clic en la columna “Destination” para ordenar las funciones de forma alfabética.

Address	Disassembly	Destination
0040110E	CALL <JMP.&gdi32.CreateSolidBrush	GDI32.CreateSolidBrush
00401141	CALL <JMP.&gdi32.CreateSolidBrush	GDI32.CreateSolidBrush
00401154	CALL <JMP.&gdi32.CreateSolidBrush	GDI32.CreateSolidBrush
0040118A	CALL <JMP.&gdi32.CreateSolidBrush	GDI32.CreateSolidBrush
0040135D	CALL <JMP.&gdi32.CreateSolidBrush	GDI32.CreateSolidBrush
00401390	CALL <JMP.&gdi32.CreateSolidBrush	GDI32.CreateSolidBrush
004013A3	CALL <JMP.&gdi32.CreateSolidBrush	GDI32.CreateSolidBrush
004013D6	CALL <JMP.&gdi32.CreateSolidBrush	GDI32.CreateSolidBrush
00401022	CALL <JMP.&user32.DialogBoxParamA	user32.DialogBoxParamA
004011E1	CALL <JMP.&user32.DialogBoxParamA	user32.DialogBoxParamA
004012B9	CALL <JMP.&user32.EnableWindow>	user32.EnableWindow
004012F5	CALL <JMP.&user32.EndDialog>	user32.EndDialog
00401307	CALL <JMP.&user32.EndDialog>	user32.EndDialog
00401416	CALL <JMP.&user32.EndDialog>	user32.EndDialog
00401028	CALL <JMP.&kernel32.ExitProcess>	kernel32.ExitProcess
004010C9	CALL <JMP.&user32.GetDlgItem>	user32.GetDlgItem
00401264	CALL <JMP.&user32.GetDlgItemTextA	user32.GetDlgItemTextA
00401007	CALL <JMP.&kernel32.GetModuleHand	kernel32.GetModuleHandleA
00401000	CALL <JMP.&comctl32.InitCommonCon	(Initial CPU selection)
00401052	CALL <JMP.&user32.LoadBitmapA>	user32.LoadBitmapA
00401071	CALL <JMP.&user32.LoadBitmapA>	user32.LoadBitmapA
00401090	CALL <JMP.&user32.LoadBitmapA>	user32.LoadBitmapA
00401333	CALL <JMP.&user32.LoadBitmapA>	user32.LoadBitmapA
004010AF	CALL <JMP.&user32.LoadIconA>	user32.LoadIconA
004011A1	CALL <JMP.&user32.ReleaseCapture>	user32.ReleaseCapture
004013EA	CALL <JMP.&user32.ReleaseCapture>	user32.ReleaseCapture
00401064	CALL <JMP.&user32.SendDlgItemMess	user32.SendDlgItemMessageA
00401083	CALL <JMP.&user32.SendDlgItemMess	user32.SendDlgItemMessageA
004010A2	CALL <JMP.&user32.SendDlgItemMess	user32.SendDlgItemMessageA
00401345	CALL <JMP.&user32.SendDlgItemMess	user32.SendDlgItemMessageA
004010BF	CALL <JMP.&user32.SendMessageA>	user32.SendMessageA
004011B2	CALL <JMP.&user32.SendMessageA>	user32.SendMessageA
004013FB	CALL <JMP.&user32.SendMessageA>	user32.SendMessageA
0040114A	CALL <JMP.&gdi32.SetBkColor>	GDI32.SetBkColor
00401180	CALL <JMP.&gdi32.SetBkColor>	GDI32.SetBkColor
00401399	CALL <JMP.&gdi32.SetBkColor>	GDI32.SetBkColor
004013CC	CALL <JMP.&gdi32.SetBkColor>	GDI32.SetBkColor
00401137	CALL <JMP.&gdi32.SetBkMode>	GDI32.SetBkMode
00401386	CALL <JMP.&gdi32.SetBkMode>	GDI32.SetBkMode
0040112D	CALL <JMP.&gdi32.SetTextColor>	GDI32.SetTextColor
00401173	CALL <JMP.&gdi32.SetTextColor>	GDI32.SetTextColor
0040137C	CALL <JMP.&gdi32.SetTextColor>	GDI32.SetTextColor
004013BF	CALL <JMP.&gdi32.SetTextColor>	GDI32.SetTextColor
00401045	CALL <JMP.&user32.SetWindowTextA>	user32.SetWindowTextA
004010EA	CALL <JMP.&user32.SetWindowTextA>	user32.SetWindowTextA
00401242	CALL <JMP.&user32.SetWindowTextA>	user32.SetWindowTextA
00401279	CALL <JMP.&user32.SetWindowTextA>	user32.SetWindowTextA
0040128D	CALL <JMP.&user32.SetWindowTextA>	user32.SetWindowTextA

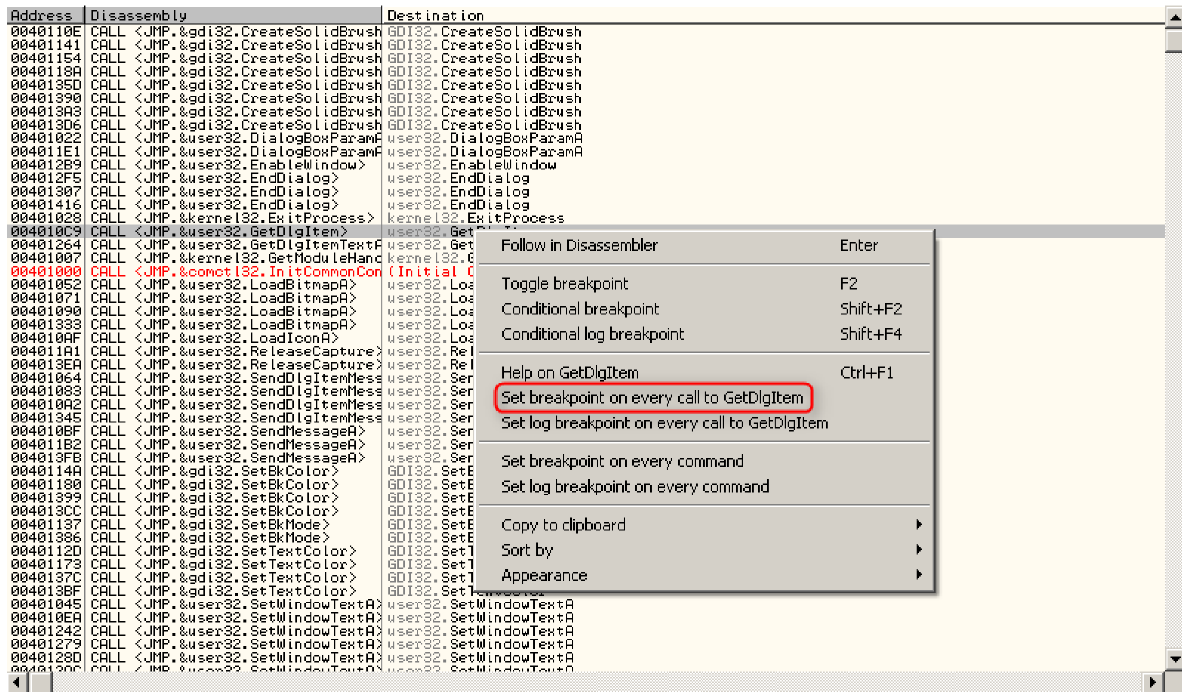
A continuación mostramos las 8 API’s más utilizados en ingeniería inversa:

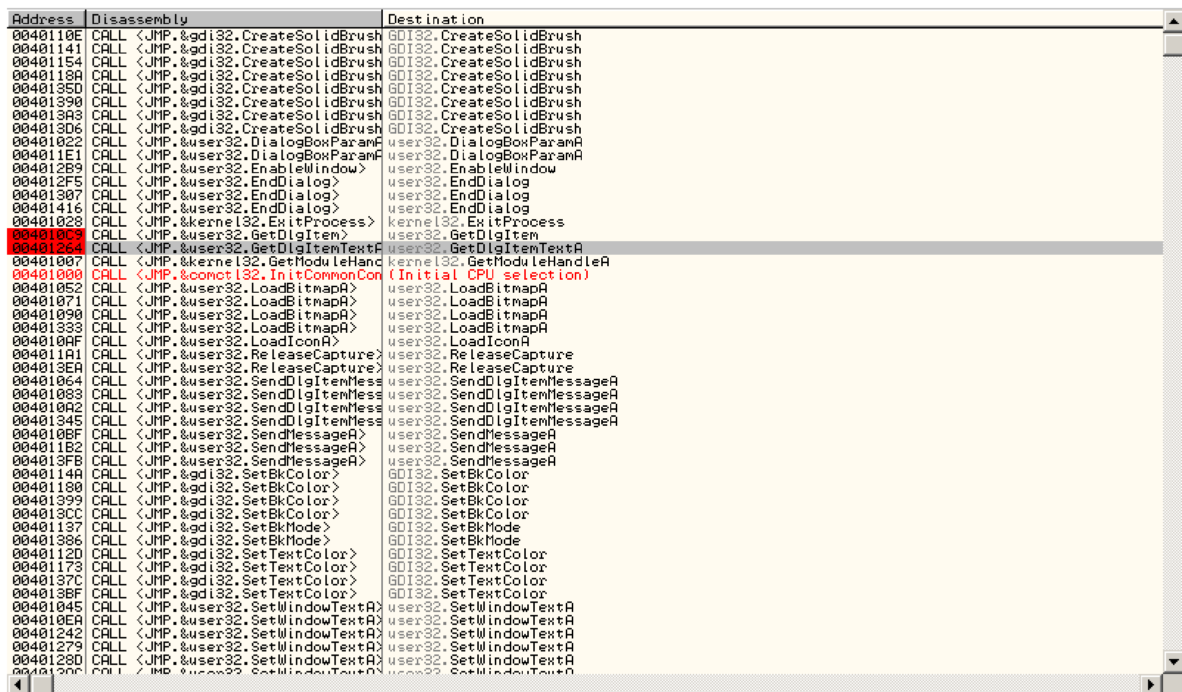
- ***GetDlgItemTextA***
- ***GetWindowTextA***
- ***lstrcmpA***
- ***GetPrivateProfileStringA***
- ***GetPrivateProfileIntA***
- ***RegQueryValueExA***
- ***WritePrivateProfileStringA***
- ***GetPrivateProfileIntA***

No obstante siempre podemos contar con la ayuda de Olly accediendo a “Get help on symbolic name”

Si nos fijamos en la lista de los calls que Olly nos muestra en el Crackme8, podemos observar que hay dos coincidencias con respecto a nuestra pequeña lista de 8 API’s: ***GetDlgItem*** y ***GetDlgItemTextA***.

La función básica de estas dos API’s es recuperar todo aquello que se haya introducido en un cuadro de texto. En nuestro caso se trata de la contraseña. Lo que queremos hacer es decirle a Olly que se detenga en el momento que se cruza con estas dos API’s. La forma de hacerlo es seleccionando la línea que tiene el CALL que queremos, hacer clic con el botón derecho y seleccionar “Set breakpoint on every call to _____”, escribiendo en el espacio libre el nombre de la API (En nuestro caso *GetDlgItem* y *GetDlgItemTextA*).

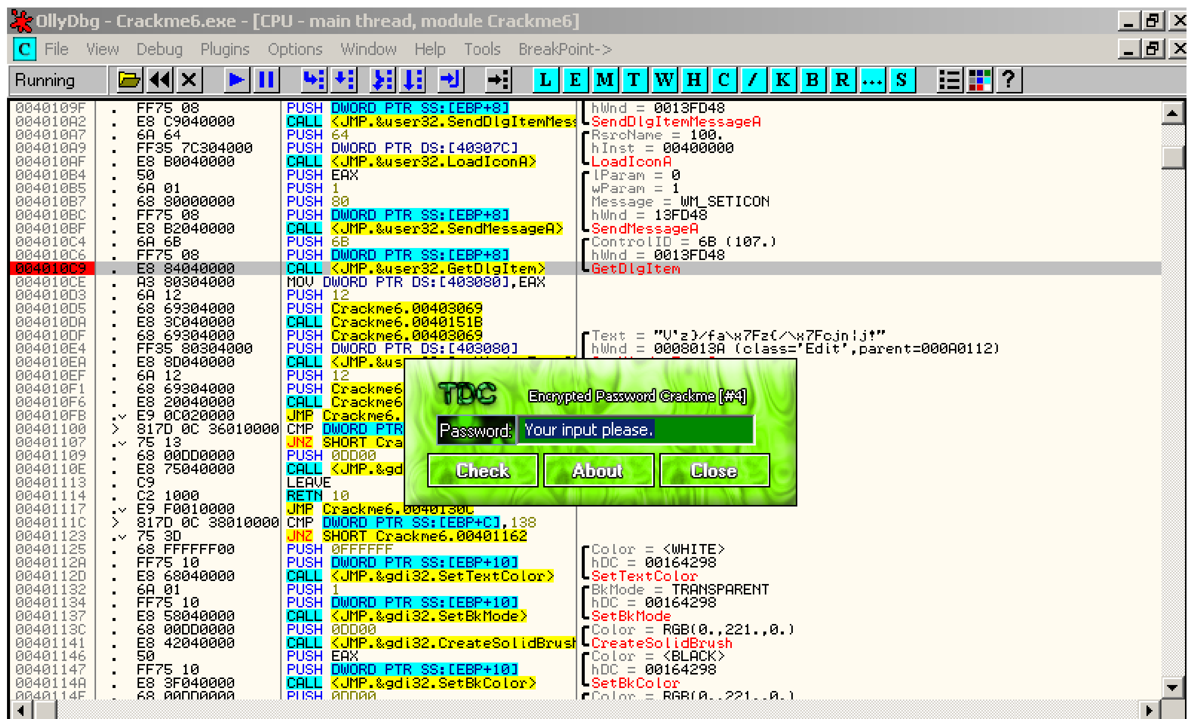




Cada vez que Olly alcanza estos dos calls, se detendrá (antes de ejecutar la llamada).

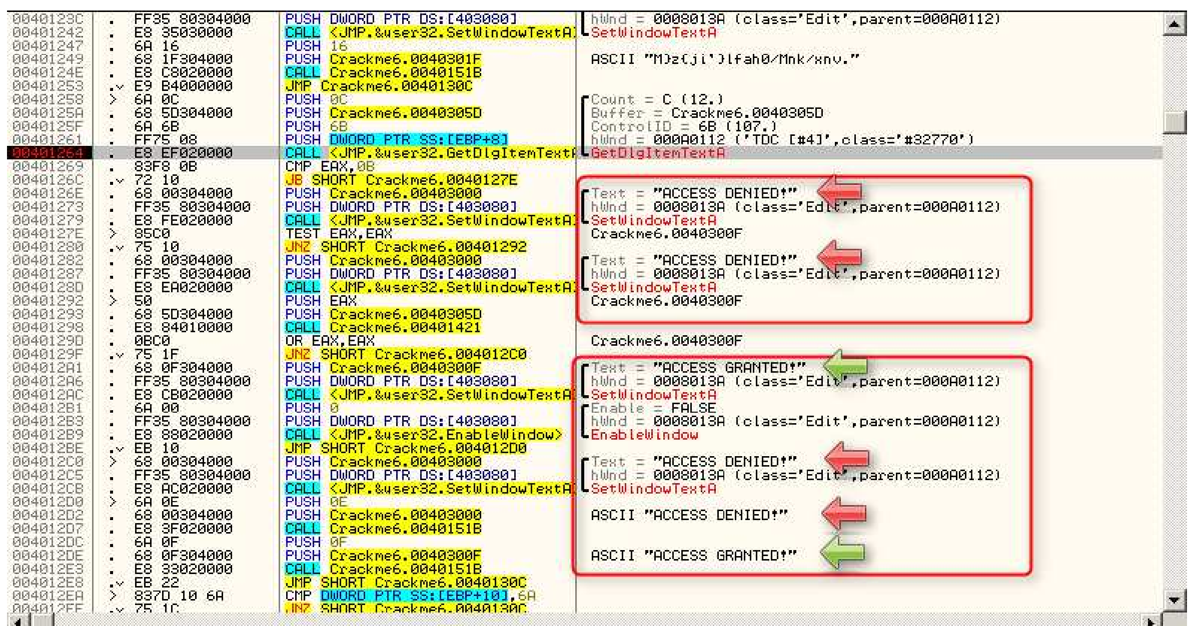
Reiniciamos la aplicación, pulsamos F9 y vemos como Olly se detiene en el CALL GetDigItem:

Ya que no hemos introducido nada hasta ahora, realmente no nos interesa lo que contenga GetDigItem. Así que seguiremos pulsando F9.



Ahora si introducimos una contraseña y hacemos clic en “Check”:

Y Olly volverá a detenerse, esta vez en GetDigItemTextA:



Si miramos un poco alrededor, podemos apreciar algunas cadenas de texto muy valiosas que al principio no aparecían o estaban cifradas.

Empezemos a crackear la aplicación. Vemos una instrucción JB que salta por encima de nuestro pirmer “ACCESS DENIED”.

```
00401258  >  6A 0C          PUSH 0C                                   ┌Count = C (12.)
0040125A  .  68 5D304000    PUSH Crackme6.0040305D                    │Buffer = Crackme6.0040305D
0040125F  .  6A 6B          PUSH 6B                                   │ControlID = 6B (107.)
00401261  .  FF75 08        PUSH DWORD PTR SS:[EBP+8]                 │hWnd = 000A0112 ('TDC [#4]',class='#32770')
00401264  .  E8 EF020000    CALL <JMP.&user32.GetDlgItemTextA>        └GetDlgItemTextA
00401269  .  83F8 0B        CMP EAX,0B
0040126C  .v 72 10          JB SHORT Crackme6.0040127E
0040126E  .  68 00304000    PUSH Crackme6.00403000                    ┌Text = "ACCESS DENIED!"
00401273  .  FF35 80304000  PUSH DWORD PTR DS:[403080]                │hWnd = 0008013A (class='Edit',parent=000A0112)
00401279  .  E8 FE020000    CALL <JMP.&user32.SetWindowTextA>         └SetWindowTextA
0040127E  >  85C0           TEST EAX,EAX                              Crackme6.0040300F
```

A continuación nos encontramos con la instrucción JNZ, que saltará por encima de nuestro segundo “bad boy”. Posteriormente saltaremos hacia nuestro “good boy”.

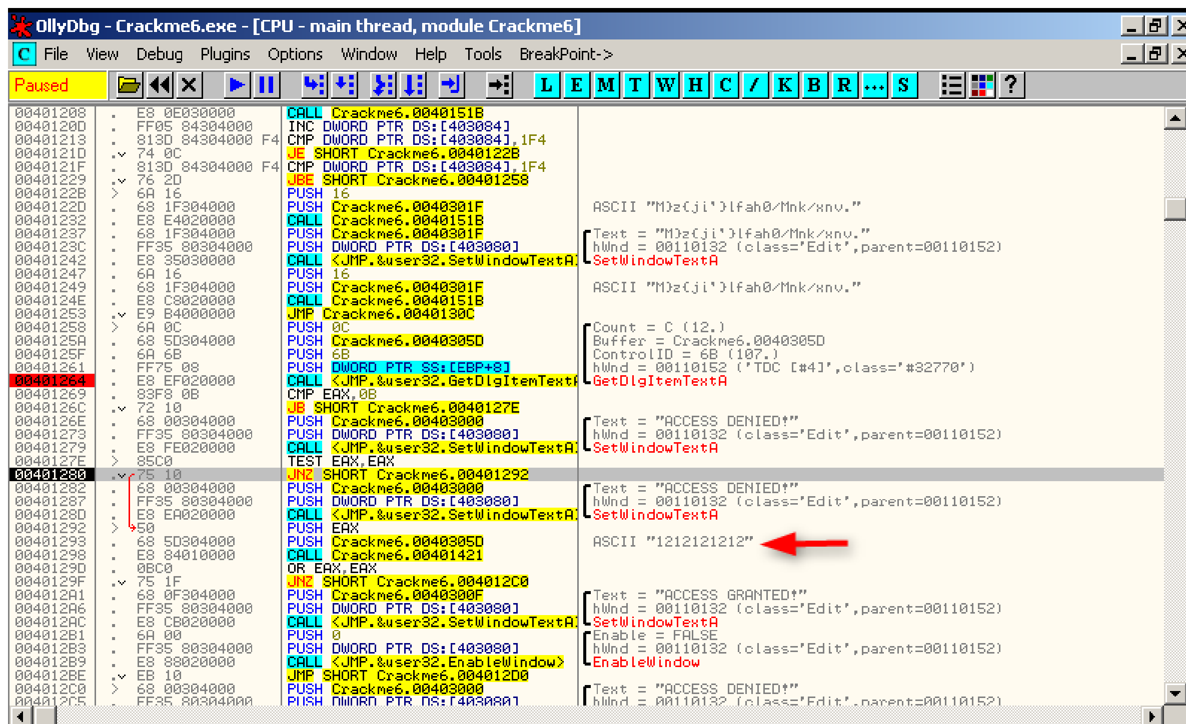
```
00401264  .  E8 EF020000    CALL <JMP.&user32.GetDlgItemTextA>        └GetDlgItemTextA
00401269  .  83F8 0B        CMP EAX,0B
0040126C  .v 72 10          JB SHORT Crackme6.0040127E
0040126E  .  68 00304000    PUSH Crackme6.00403000                    ┌Text = "ACCESS DENIED!"
00401273  .  FF35 80304000  PUSH DWORD PTR DS:[403080]                │hWnd = 0008013A (class='Edit',parent=000A0112)
00401279  .  E8 FE020000    CALL <JMP.&user32.SetWindowTextA>         └SetWindowTextA
0040127E  >  85C0           TEST EAX,EAX                              Crackme6.0040300F
00401280  .v 75 10          JNZ SHORT Crackme6.00401292
00401282  .  68 00304000    PUSH Crackme6.00403000                    ┌Text = "ACCESS DENIED!"
00401287  .  FF35 80304000  PUSH DWORD PTR DS:[403080]                │hWnd = 0008013A (class='Edit',parent=000A0112)
0040128D  .  E8 EA020000    CALL <JMP.&user32.SetWindowTextA>         └SetWindowTextA
00401292  >  50             PUSH EAX                                  Crackme6.0040300F
00401293  .  68 5D304000    PUSH Crackme6.0040305D
00401298  .  E8 84010000    CALL Crackme6.00401421
0040129D  .  0BC0           OR EAX,EAX                                Crackme6.0040300F
0040129F  .v 75 1F          JNZ SHORT Crackme6.004012C0
004012A1  .  68 0F304000    PUSH Crackme6.0040300F                    ┌Text = "ACCESS GRANTED!"
004012A6  .  FF35 80304000  PUSH DWORD PTR DS:[403080]                │hWnd = 0008013A (class='Edit',parent=000A0112)
004012AC  .  E8 CB020000    CALL <JMP.&user32.SetWindowTextA>         └SetWindowTextA
```

Nustro objetivo es por tanto saltar por encima de los dos “bad boys”. Reiniciamos la aplicación, pulsamos F9, (eliminamos el primer Breakpoint) e introducimos la contraseña. La aplicación se detiene en nuestro segundo Breakpoint.

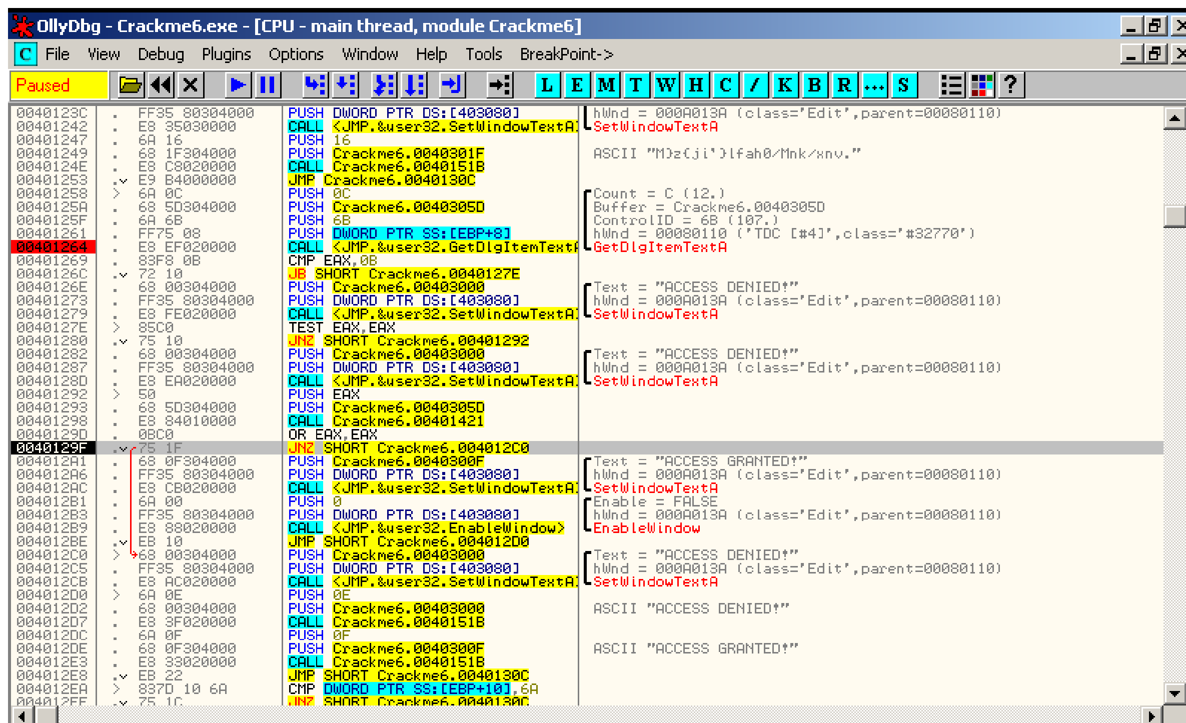
Pulsamos F8 dos veces hasta llegar a la instrucción JB. Tomamos el salto que nos llevará a la dirección 40127E, donde nos encontramos con una comparación TEST.

```
OllyDbg - Crackme6.exe - [CPU - main thread, module Crackme6]
C File View Debug Plugins Options Window Help Tools BreakPoint->
Paused                                  L E M T W H C / K B R ... S

00401208  .  E8 0E030000    CALL Crackme6.0040151B
0040120D  .  FF05 84304000  INC DWORD PTR DS:[403084]
00401213  .  813D 84304000 F4 CMP DWORD PTR DS:[403084],1F4
0040121D  .v 74 0C          JE SHORT Crackme6.0040122B
0040121F  .  813D 84304000 F4 CMP DWORD PTR DS:[403084],1F4
00401229  .v 76 2D          JBE SHORT Crackme6.00401258
0040122B  >  6A 16          PUSH 16
0040122D  .  68 1F304000    PUSH Crackme6.0040301F                    ASCII "M)z(ji')lfah0/Mnk/xnv."
00401232  .  E8 E4020000    CALL Crackme6.0040151B
00401237  .  68 1F304000    PUSH Crackme6.0040301F                    ┌Text = "M)z(ji')lfah0/Mnk/xnv."
0040123C  .  FF35 80304000  PUSH DWORD PTR DS:[403080]                │hWnd = 00110132 (class='Edit',parent=00110152)
00401242  .  E8 35030000    CALL <JMP.&user32.SetWindowTextA>         └SetWindowTextA
00401247  .  6A 16          PUSH 16
00401249  .  68 1F304000    PUSH Crackme6.0040301F                    ASCII "M)z(ji')lfah0/Mnk/xnv."
0040124E  .  E8 C8020000    CALL Crackme6.0040151B
00401253  .v E9 B4000000    JMP Crackme6.0040130C
00401258  >  6A 0C          PUSH 0C                                   ┌Count = C (12.)
0040125A  .  68 5D304000    PUSH Crackme6.0040305D                    │Buffer = Crackme6.0040305D
0040125F  .  6A 6B          PUSH 6B                                   │ControlID = 6B (107.)
00401261  .  FF75 08        PUSH DWORD PTR SS:[EBP+8]                 │hWnd = 00110152 ('TDC [#4]',class='#32770')
00401264  .  E8 EF020000    CALL <JMP.&user32.GetDlgItemTextA>        └GetDlgItemTextA
00401269  .  83F8 0B        CMP EAX,0B
0040126C  .v 72 10          JB SHORT Crackme6.0040127E
0040126E  .  68 00304000    PUSH Crackme6.00403000                    ┌Text = "ACCESS DENIED!"
00401273  .  FF35 80304000  PUSH DWORD PTR DS:[403080]                │hWnd = 00110132 (class='Edit',parent=00110152)
00401279  .  E8 FE020000    CALL <JMP.&user32.SetWindowTextA>         └SetWindowTextA
0040127E  >  85C0           TEST EAX,EAX
00401280  .v 75 10          JNZ SHORT Crackme6.00401292
00401282  .  68 00304000    PUSH Crackme6.00403000                    ┌Text = "ACCESS DENIED!"
00401287  .  FF35 80304000  PUSH DWORD PTR DS:[403080]                │hWnd = 00110132 (class='Edit',parent=00110152)
0040128D  .  E8 EA020000    CALL <JMP.&user32.SetWindowTextA>         └SetWindowTextA
00401292  >  50             PUSH EAX
00401293  .  68 5D304000    PUSH Crackme6.0040305D                    ASCII "1212121212"
00401298  .  E8 84010000    CALL Crackme6.00401421
0040129D  .  0BC0           OR EAX,EAX
0040129F  .v 75 1F          JNZ SHORT Crackme6.004012C0
004012A1  .  68 0F304000    PUSH Crackme6.0040300F                    ┌Text = "ACCESS GRANTED!"
004012A6  .  FF35 80304000  PUSH DWORD PTR DS:[403080]                │hWnd = 00110132 (class='Edit',parent=00110152)
004012AC  .  E8 CB020000    CALL <JMP.&user32.SetWindowTextA>         └SetWindowTextA
004012B1  .  6A 00          PUSH 0                                    ┌Enable = FALSE
004012B3  .  FF35 80304000  PUSH DWORD PTR DS:[403080]                │hWnd = 00110132 (class='Edit',parent=00110152)
004012B9  .  E8 88020000    CALL <JMP.&user32.EnableWindow>           └EnableWindow
004012BE  .v EB 10          JMP SHORT Crackme6.004012D0
004012C0  >  68 00304000    PUSH Crackme6.00403000                    ┌Text = "ACCESS DENIED!"
004012C5  .  FF35 80304000  PUSH DWORD PTR DS:[403080]                │hWnd = 00110132 (class='Edit',parent=00110152)
```

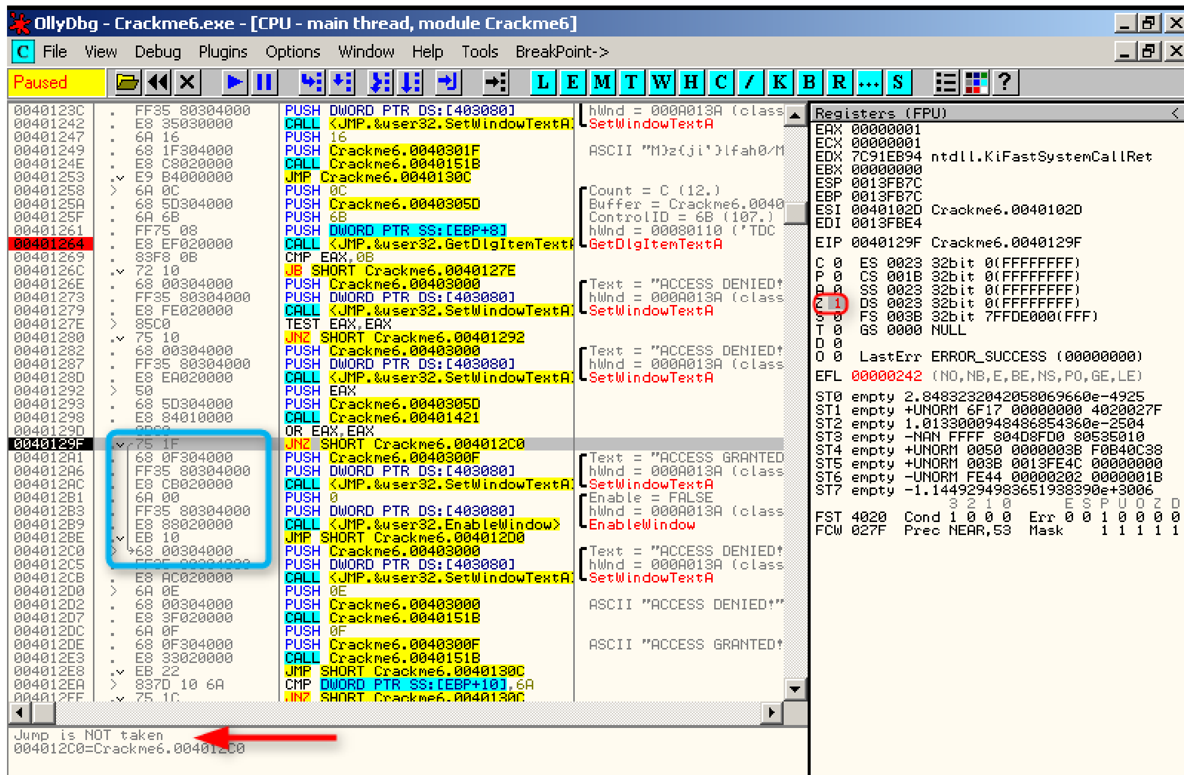
Pulsamos F8 para detenernos en la dirección 401280 donde nos espera una instrucción JNZ (En este momento podemos ver nuestra contraseña en la columna de los comentarios).



Tomamos el salto por encima de nuestro segundo “bad boy” y seguimos pulsando F8 hasta llegar a la siguiente instrucción JNZ en la dirección 40129F.



Este salto pasará por encima de nuestro “good boy”. Cosa que queremos evitar. Cambiamos el valor de la bandera Z para no tomar el salto y pasamos por nuestro “good boy”.



Finalmente pulsamos F9 y vemos como hemos crackeado el programa de forma exitosa.

7.7 Caso práctico 7: Niveles de parcheo

En Ingeniería Inversa hay una especie de regla no escrita sobre los distintos niveles de parchear un binario. Básicamente hay cuatro niveles para parchear un binario. Estudiaremos cada uno de estos niveles a través del crackme del ejercicio anterior.

Nivel 1 – LAME

El método LAME, o *Localized Assembly Manipulation and Enhancing method,* es el que estuvimos empleando hasta ahora. Trata de descubrir el primer lugar en el código donde aparece la combinación comparar/saltar y sustituirla por una instrucción NOP o forzar el salto con la instrucción JNZ. Este método ha funcionado de forma mágica en todos los ejercicios estudiados hasta ahora, gracias a la simplicidad de los binarios. Pero desafortunadamente no todas las aplicaciones son así de fáciles por lo que el método LAME se vuelve ineficaz. Veremos a continuación algunas limitaciones con respecto a este método:

- Muchas aplicaciones realizan varias comprobaciones a la hora de verificar si un programa está correctamente registrado o no. El hecho de parchear un “bad boy” no significa que no puede haber más a lo largo de todo el código. Además muchas comprobaciones no son descubiertas hasta que sucedan otros eventos en el código lo que podría llevarnos a buscar sitios alternativos para parchear.

- Hay aplicaciones que tratan de esconder deliberadamente la combinación comparar/saltar. Estas pueden aparecer en una DLL, en distintas secciones del código, modificados polimórficamente,.. Hay muchas formas de ofuscar la combinación ganadora.

- En ocasiones nos podemos ver envueltos parcheando 7 comprobaciones de registros, nopeando otras tantas comprobaciones, etc. Esto puede resultar bastante confuso y si somos sinceros no resulta ser una solución muy elegante.

- No vamos aprender mucho utilizando solo este método.

Dicho esto conviene resaltar que aunque parezca bastante rudimentario no debemos menospreciar este método ya que hay muchas aplicaciones ahí afuera que pueden ser crackeados poniendo un simple parche a una combinación de instrucciones comparar/saltar.

Nivel 2 – NOOB

Este método (Not Only Obvious Breakpoints method) implica ir un paso más allá del método LAME. Generalmente conlleva entrar en un CALL justo antes de la combinación comparar/saltar para averiguar lo que causa la combinación de instrucciones comparar/saltar actuar de una forma u otra. El beneficio de esta técnica reside en que podemos encontrarnos con otras partes del código que hagan una llamada a la misma rutina de comprobar el proceso de registrarse. Parcheando un sitio puede llevar a parchear todos las demás lugares donde el binario llama a la rutina que comprueba el proceso de registro. No obstante conviene señalar los siguientes inconvenientes de este método:

- A veces se utiliza este método para comprobar algo más que un sencillo proceso de registro. Por ejemplo, puede haber una función genérica que compara dos cadenas, devolviendo ‘True’ o ‘False’ dependiendo si coinciden o no.

➢ Este método conlleva más tiempo y experiencia para descubrir las mejores opciones que llevan a devolver los valores correctos.

Nivel 3 – SKILLED

El método SKILLED (*Some Knowledge In Lower Level Engineered Data*) es similar al NOOB, excepto que en este caso se estudia la rutina completa del código para ver exactamente lo que está pasando. Esto aporta un conjunto de beneficios como el de comprender todos los 'trucos' que se están usando (como por ejemplo el de almacenar variables en la memoria para después recuperarlas), o el de ofrecer mucho más lugares para parchear y menos intrusivos. También aporta una percepción más profunda en cuanto al funcionamiento de un programa y por ende una mejora en la comprensión del lenguaje ensamblador.

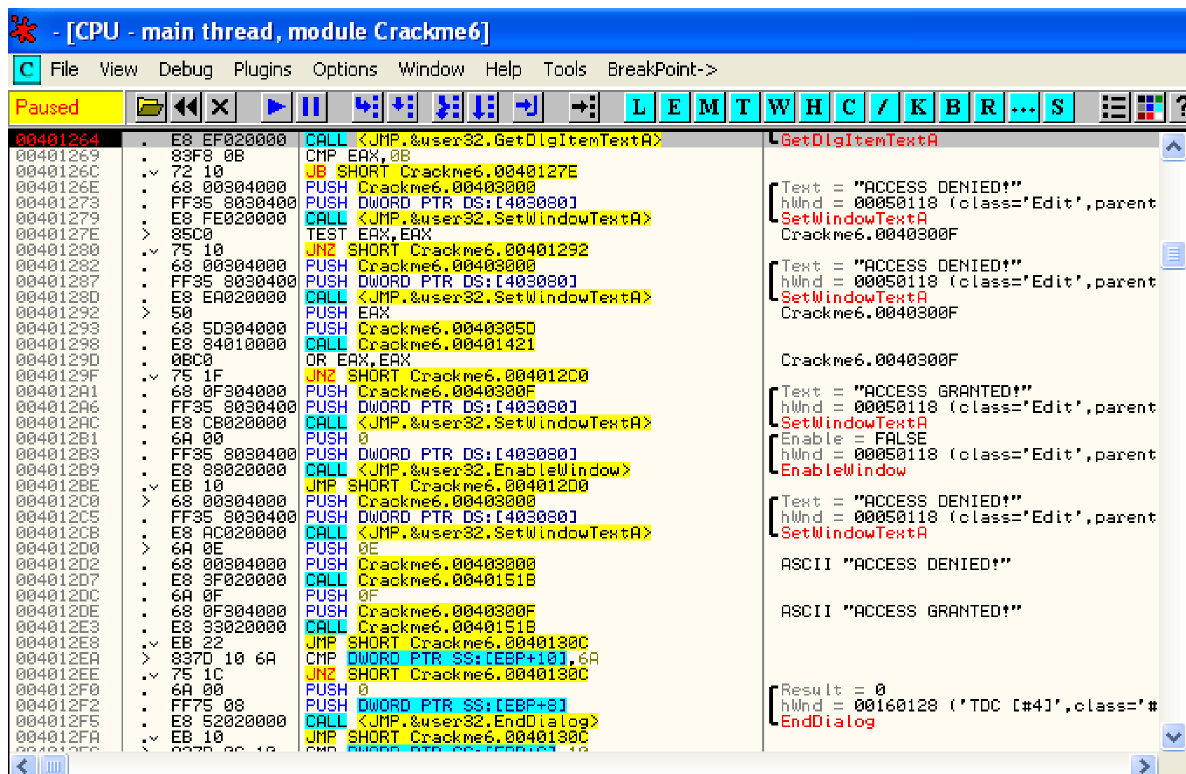
La mayor desventaja de este método es que es más difícil y precisa de mucho más tiempo.

Nivel 4 – SK1LL$

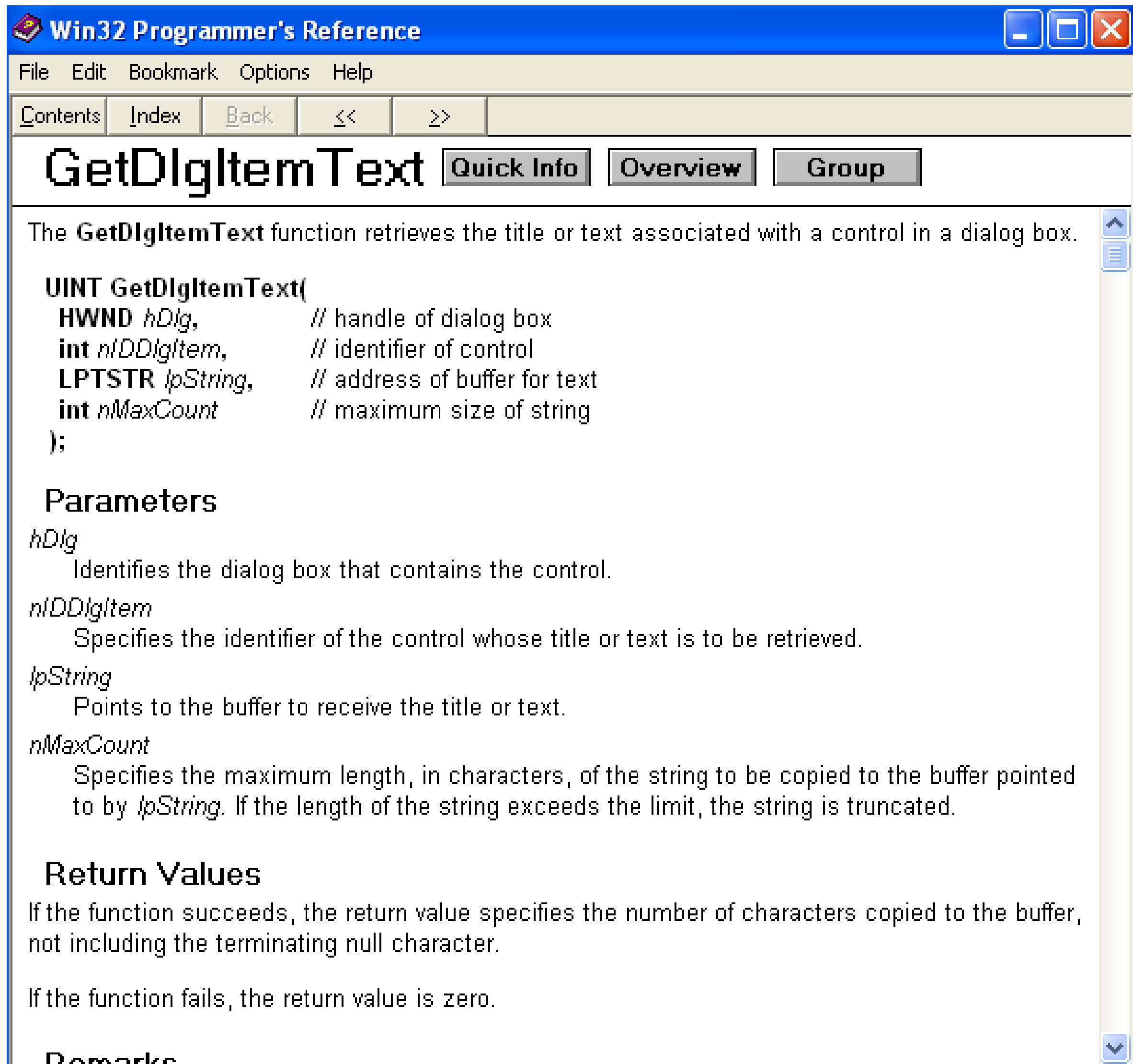
Pensado como el santo grial del cracking, (*Serial Keygenning In Low-level Languages)* significa que no solo hemos descubierto la forma exacta de funcionamiento del proceso de registro, sino que también somos capaces de re-crearlo. Esto permitirá a un nuevo usuario escribir cualquier nombre de usuario y el código del generador de claves nos dará el serial apropiado para que funcione el binario correctamente.

Estudiando la aplicación crackme6.exe desde el nivel 2

Cargamos la aplicación en Olly, ponemos el Breakpoint en *GetDIgItemTextA* (401264), pulsamos F9, introducimos un serial, hacemos clic en "Check" y Olly se detiene en nuestro Breakpoint:



Buscamos algo de ayuda sobre el significado de GetDIgitItemTextA:



Win32 Programmer's Reference

File Edit Bookmark Options Help

Contents Index Back << >>

GetDlgItemText Quick Info Overview Group

The **GetDlgItemText** function retrieves the title or text associated with a control in a dialog box.

```
UINT GetDlgItemText(
  HWND hDlg,            // handle of dialog box
  int nIDDlgItem,       // identifier of control
  LPTSTR lpString,      // address of buffer for text
  int nMaxCount         // maximum size of string
 );
```

Parameters

hDlg
Identifies the dialog box that contains the control.

nIDDlgItem
Specifies the identifier of the control whose title or text is to be retrieved.

lpString
Points to the buffer to receive the title or text.

nMaxCount
Specifies the maximum length, in characters, of the string to be copied to the buffer pointed to by *lpString*. If the length of the string exceeds the limit, the string is truncated.

Return Values

If the function succeeds, the return value specifies the number of characters copied to the buffer, not including the terminating null character.

If the function fails, the return value is zero.

Destacamos lo que nos interesa: Uno de los argumentos es un puntero hacia el buffer donde será almacenado nuestra contraseña (lpString), y el valor de regreso en EAX es la longitud de la cadena.

El puntero a la cadena del buffer es 40305D.

```
00401258  > 6A 0C        PUSH 0C                                 Count = C (12.)
0040125A  . 68 5D304000  PUSH Crackme6.0040305D                  Buffer = Crackme6.0040305D
0040125F  . 6A 6B        PUSH 6B                                 ControlID = 6B (107.)
00401261  . FF75 08      PUSH DWORD PTR SS:[EBP+8]               hWnd = 00160128 ('TDC [#4]',class='#
00401264  . E8 EF020000  CALL <JMP.&user32.GetDlgItemTextA>      GetDlgItemTextA
```

Esto significa que la función va a copiar el texto introducido en el cuadro de dialogo en un buffer que comienza en la dirección 40305D, y devolverá la longitud de la cadena en EAX. En nuestro caso la contraseña que hemos introducido, “12121212” va a ser recuperada y devuelta con la longitud de la contraseña en EAX, en este caso 8. Si ahora nos fijamos en las dos líneas siguientes veremos que este valor se va a comparar con 0x0B (11d) y el programa saltará si EAX es menor que esa cantidad. Es decir, si la longitud de nuestra contraseña (EAX) es menor que 0x0B (11 digitos) entonces saltará. Si no saltamos veremos como aparece nuestro “bad boy”, así que nuestra contraseña tiene que tener menos de 11 digitos:

Una vez realizado el salto pulsamos F8 hasta llegar a la dirección 401280:

Si nos fijamos en la dirección anterior (40127E) vemos que en ella se comprueba que EAX, que sigue almacenando la longitud de nuestra contraseña, sea igual a cero y si no es igual a 0 entonces saltará por encima de nuestro segundo “bad boy”.

Ahora sabemos que el primer “bad boy” comprueba la longitud de nuestra contraseña y el segundo “bad boy” combrueba si la contraseña tiene 0 digitos.

Después de tomar el salto, aparece la instrucción PUSH EAX, que pondrá la longitud de la contraseña en la parte superior de la pila. Luego aparece la instrucción PUSH Crackme6.0040305D, que pondrá en la parte superior de la pila la contraseña almacenada en el buffer:

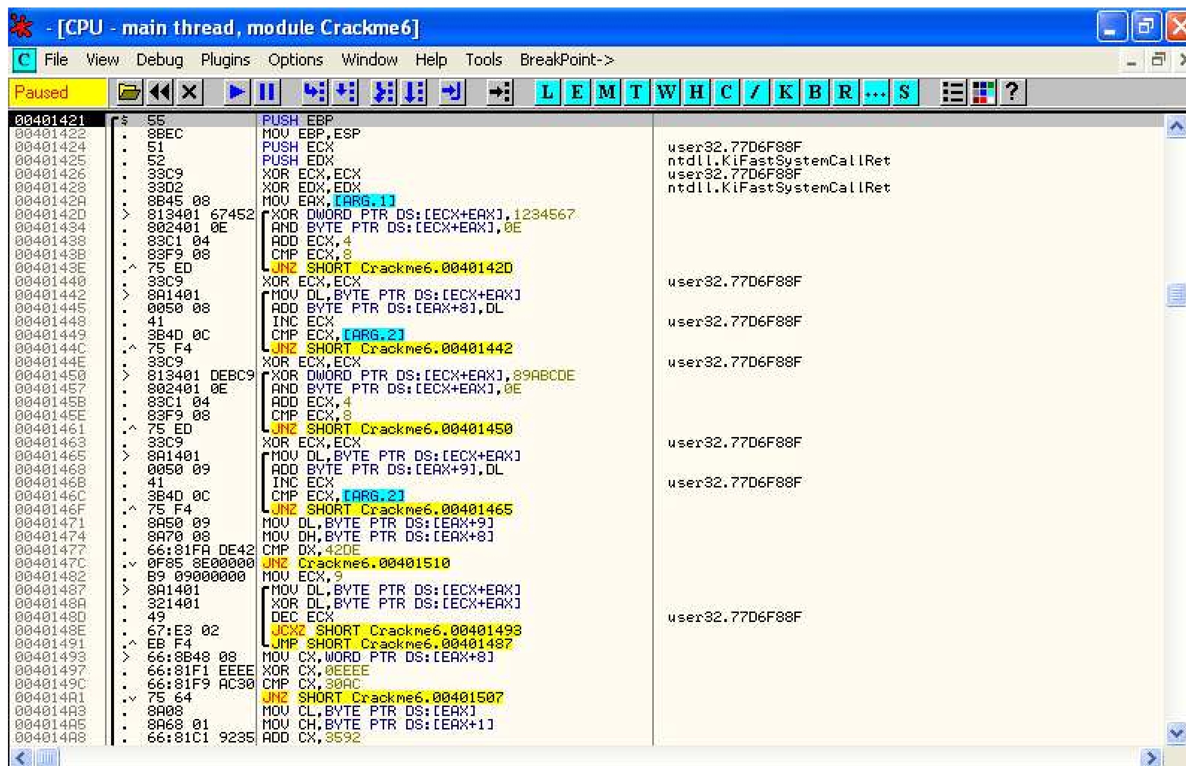
Ahora sabemos que nuestra contraseña está almacenada en la memoria correspondiente a la dirección 40305D.

Después de ser empujados estos dos valores en la pila, vamos a llamar a la rutina principal del proceso de registro en la dirección 401298 (sabemos esto porque es el CALL justo antes de la combinación comparar/saltar, cuyo resultado va a determinar si vamos a saltar al “good boy” o al “bad boy”).

En la siguiente línea correspondiente a la dirección 40129D, EAX recibe la instrucción OR consigo misma (esto configurará la bandera cero dependiendo si EAX es cero o no). Saltará por encima del “good boy” si no es igual a cero:

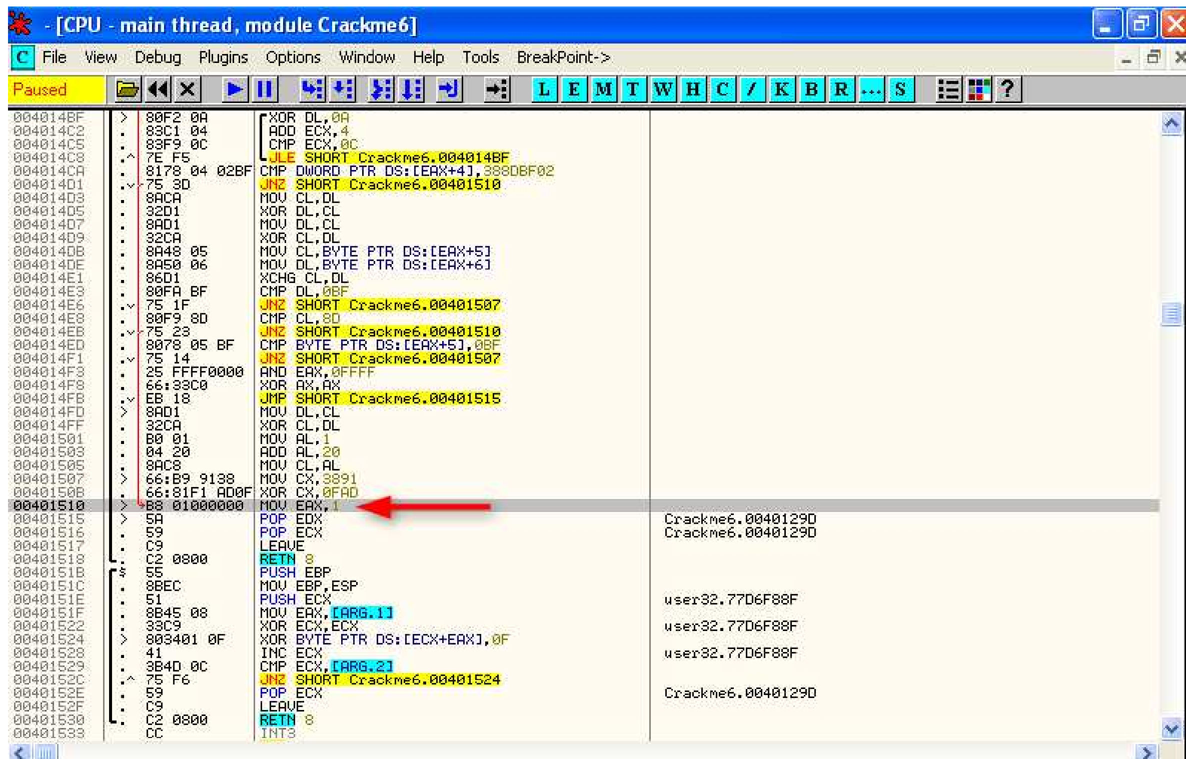
```
00401298  .  E8 84010000  CALL Crackme6.00401421
0040129D  .  0BC0         OR EAX,EAX
0040129F  .  75 1F        JNZ SHORT Crackme6.004012C0
004012A1  .  68 0F304000  PUSH Crackme6.0040300F          ; Text = "ACCESS GRANTED!"
004012A6  .  FF35 8030400 PUSH DWORD PTR DS:[403080]      ; hWnd = 000A0152 (class='Edit',parent=00170128)
004012AC  .  E8 CB020000  CALL <JMP.&user32.SetWindowTextA> ; SetWindowTextA
004012B1  .  6A 00        PUSH 0                          ; Enable = FALSE
004012B3  .  FF35 8030400 PUSH DWORD PTR DS:[403080]      ; hWnd = 000A0152 (class='Edit',parent=00170128)
004012B9  .  E8 88020000  CALL <JMP.&user32.EnableWindow>  ; EnableWindow
004012BE  .  EB 10        JMP SHORT Crackme6.004012D0
004012C0  >  68 00304000  PUSH Crackme6.00403000          ; Text = "ACCESS DENIED!"
004012C5  .  FF35 8030400 PUSH DWORD PTR DS:[403080]      ; hWnd = 000A0152 (class='Edit',parent=00170128)
004012CB  .  E8 AC020000  CALL <JMP.&user32.SetWindowTextA> ; SetWindowTextA
```

La rutina del registro que es llamada en la dirección 401298 va a poner en algún sitio un valor en EAX y una vez que finaliza la rutina devolverá EAX con ese valor a través de una instrucción RETN. Una vez de regreso al código de esta sección se procederá a comprobar si EAX es igual a cero o no, y si no lo es saldrá el mensaje del “bad boy”. Así que tenemos que asegurarnos de que en este CALL de la dirección 401298, EAX valga cero cuando regresa de la rutina. Si logramos conseguir esto, será el único parche que tengamos que hacer a esta aplicación (conjuntamente con el parche para modificar la longitud de la contraseña). Pulsamos pues F7 para entrar en la rutina de registro en la dirección 401298:



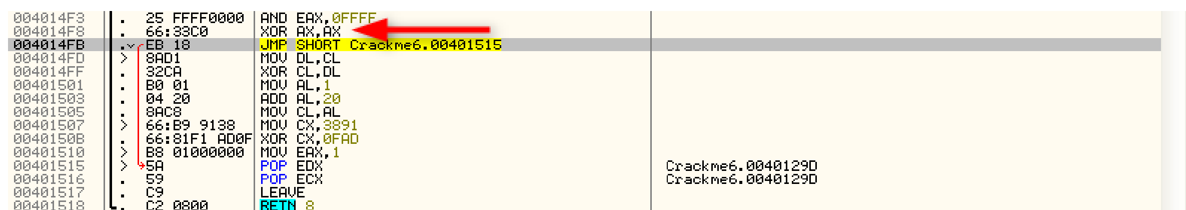
Por norma general nos vamos a situar al final de la rutina, para averiguar qué condiciones tienen que cumplirse para que EAX sea igual a cero cuando alcance la instrucción

RETN. Situémonos pues en la instrucción RETN para ir analizando el código de abajo hacia arriba:



Lo primero que queremos evitar es tomar el salto a la dirección 401510, ya que EAX va recibir el valor 1, justo antes de regresar (vemos el RETN en la dirección 401518).

Si ahora subimos un poco en el código veremos el lugar donde EAX va a recibir el valor cero (XOR AX, AX) y el camino que hay que seguir para que regrese con ese valor a nuestra rutina principal:



Hasta aquí hemos visto los conocimientos generales que corresponden a un nivel 2 para crackear programas. Lo único que faltaría por hacer es parchear la aplicación para que EAX sea siempre cero cuando regrese de la rutina.

Subiendo al Nivel 3

Una de las razones por las que nos interesa subir de nivel es para hacer por ejemplo un generador de claves. Y para ello necesitaremos una comprensión mayor del código que estamos estudiando. Volviendo al comienza de la rutina, empezaremos a parchear a un nivel SKILLED:

Al principio de la rutina vemos algunas instrucciones que empujan registros para crear el espacio donde almacenar algunas variables locales. Los valores en ECX y EDX son empujados en la pila para más tarde sacarlos y devolverlos a su estado original al finalizar la rutina. En la dirección 40142A el argumento local se mueve a la pila (es la dirección de nuestra contraseña) en EAX. Si nos fijamos en la ventana de registros veremos que EAX contiene la dirección 40305D, que es la dirección de nuestra contraseña.

A continuación veremos la siguiente instrucción:

XOR DWORD PTR DS:[ECX+EAX], 1234567

Aquí añadimos ECX (que es cero) a la dirección de nuestra contraseña (que está almacenada en 40305D), y después cogemos DWORD (4 bytes) en ese lugar en la memoria para hacer un XOR con el valor hexadecimal 1234567. Como ECX es cero, añadirlo a la dirección de nuestra contraseña no va a modificar esa dirección. Resumiendo, cogemos los primeros 4 bytes de nuestra contraseña y hacemos un XOR con 1234567, almacenando el resultado devuelta en la misma dirección de la memoria, que es el comienzo de nuestra contraseña.

Vamos a ver esto en acción; primero nos aseguramos que estamos detenidos en la dirección 40142D. Si nos fijamos en la ventana de la información veremos la dirección de [ECX+EAX] y su valor 32313231 (que en ASCII es igual a '2121' en Little endian):

```
DS:[0040305D]=32313231
Jump from 0040143E
```

Hacemos clic con el botón derecho sobre la primera línea y seleccionamos "Follow address in Dump". De esta forma podremos ver la memoria donde está almacenada nuestra contraseña:

Ahora la ventana Dump nos muestra la memoria comenzando por la dirección 40305D. Podemos ver los primeros 8 bytes de nuestra contraseña. Y si recordamos la línea de código que estamos analizando lo que va hacer la instrucción es coger los primeros 4 bytes en esta dirección (31,32,31,32) y aplicar XOR con 0x1234567, almacenando el resultado devuelta en esta dirección de la memoria.

```
Address   Hex dump                                         ASCII
0040305D  31 32 31 32 31 32 00 00 00 00 00 00 56 60 7A 7D  121212......V`z}
0040306D  2F 66 61 7F 7A 7B 2F 7F 63 6A 6E 7C 6A 21 00 00  /fa⌂z{/⌂cjn|j!..
0040307D  00 40 00 9C 01 07 00 01 00 00 00 00 00 00 00 00  .@.£0•.0........
0040308D  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0040309D  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030AD  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030BD  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030CD  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030DD  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030ED  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030FD  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

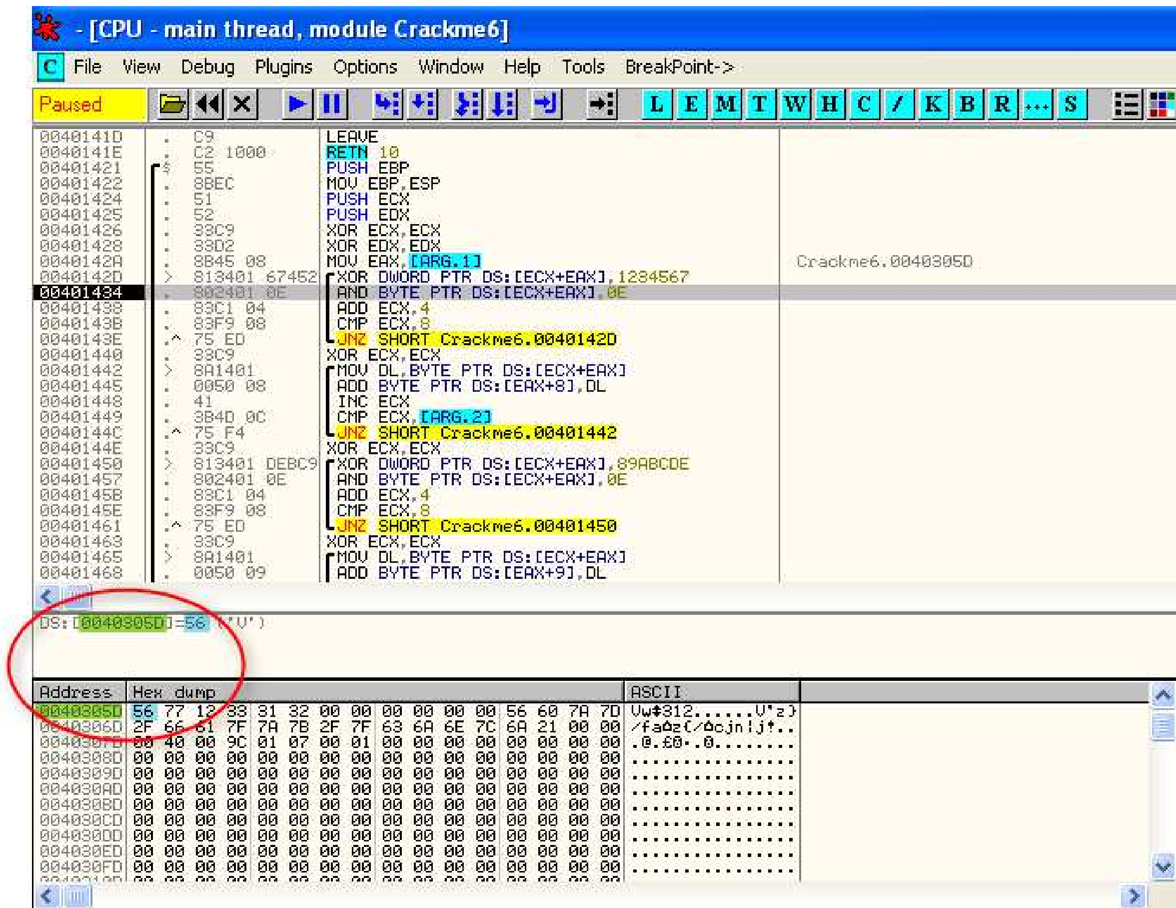
Pulsamos F8 y veremos cómo los primero 4 bytes de nuestra contraseña han cambiado, debido a la instrucción XOR con 0x1234567:

```
- [CPU - main thread, module Crackme6]
C File View Debug Plugins Options Window Help Tools BreakPoint->
Paused                                   L E M T W H C / K B R ... S

0040141D   .  C9            LEAVE
0040141E   .  C2 1000       RETN 10
00401421   $  55            PUSH EBP
00401422   .  8BEC          MOV EBP,ESP
00401424   .  51            PUSH ECX
00401425   .  52            PUSH EDX
00401426   .  33C9          XOR ECX,ECX
00401428   .  33D2          XOR EDX,EDX
0040142A   .  8B45 08       MOV EAX,[ARG.1]                          Crackme6.0040305D
0040142D   >  813401 67452  XOR DWORD PTR DS:[ECX+EAX],1234567
00401434   .  802401 0E     AND BYTE PTR DS:[ECX+EAX],0E
00401438   .  83C1 04       ADD ECX,4
0040143B   .  83F9 08       CMP ECX,8
0040143E   .^ 75 ED         JNZ SHORT Crackme6.0040142D
00401440   .  33C9          XOR ECX,ECX
00401442   >  8A1401        MOV DL,BYTE PTR DS:[ECX+EAX]
00401445   .  0050 08       ADD BYTE PTR DS:[EAX+8],DL
00401448   .  41            INC ECX
00401449   .  3B4D 0C       CMP ECX,[ARG.2]
0040144C   .^ 75 F4         JNZ SHORT Crackme6.00401442
0040144E   .  33C9          XOR ECX,ECX
00401450   >  813401 DEBC9  XOR DWORD PTR DS:[ECX+EAX],89ABCDE
00401457   .  802401 0E     AND BYTE PTR DS:[ECX+EAX],0E
0040145B   .  83C1 04       ADD ECX,4
0040145E   .  83F9 08       CMP ECX,8
00401461   .^ 75 ED         JNZ SHORT Crackme6.00401450
00401463   .  33C9          XOR ECX,ECX
00401465   >  8A1401        MOV DL,BYTE PTR DS:[ECX+EAX]
00401468   .  0050 09       ADD BYTE PTR DS:[EAX+9],DL

DS:[0040305D]=56 ('V')

Address   Hex dump                                         ASCII
0040305D  56 77 12 33 31 32 00 00 00 00 00 00 56 60 7A 7D  Vw‡312......V`z}
0040306D  2F 66 61 7F 7A 7B 2F 7F 63 6A 6E 7C 6A 21 00 00  /fa⌂z{/⌂cjn|j!..
0040307D  00 40 00 9C 01 07 00 01 00 00 00 00 00 00 00 00  .@.£0•.0........
0040308D  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0040309D  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030AD  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030BD  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030CD  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030DD  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030ED  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030FD  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

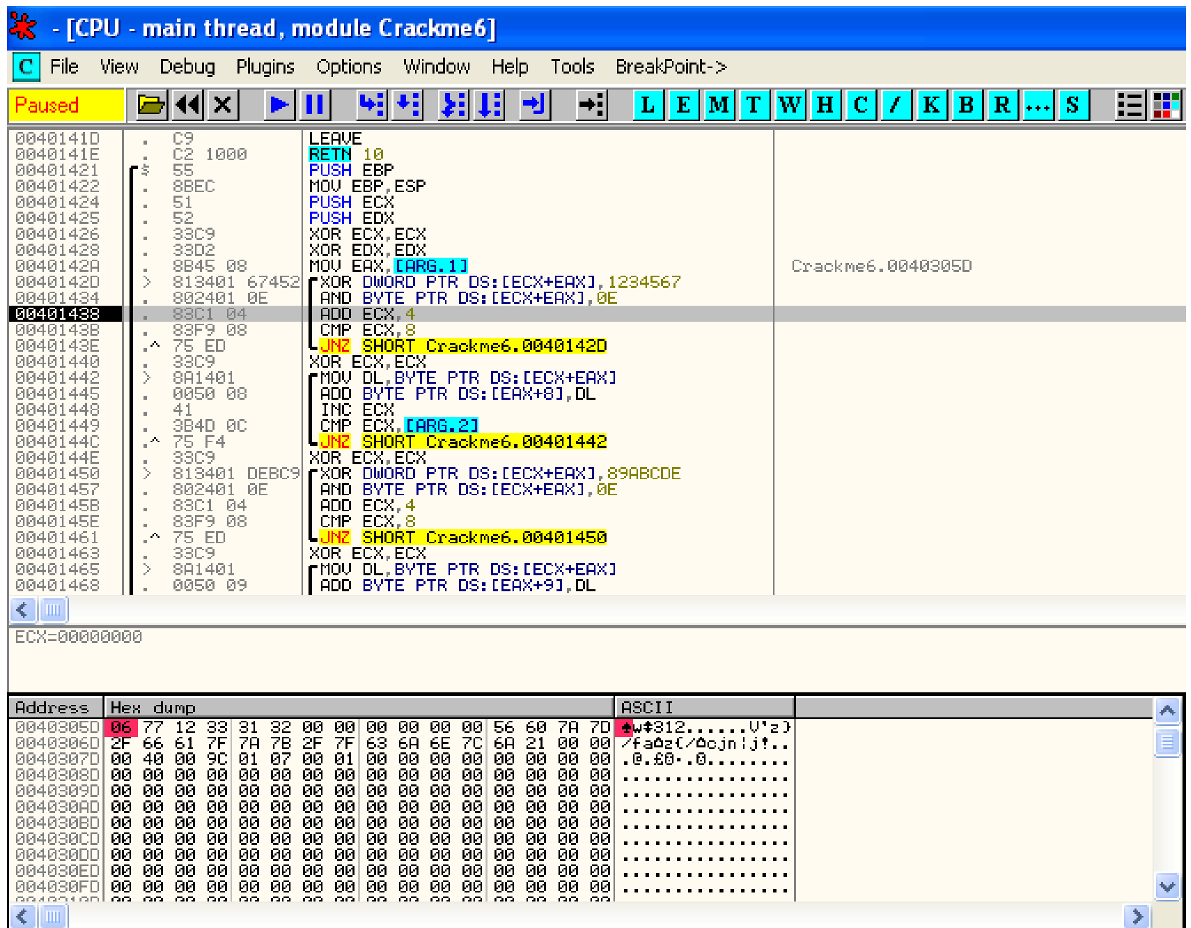
Continuamos estudiando el código en la línea siguiente.

AND BYTE PTR DS:[ECX+EAX], 0E,

Sabemos que [ECX+EAX] corresponden a la dirección 40305D, que es la dirección de nuestra antigua contraseña. Ahora cogemos un byte de esa dirección y le aplicamos la instrucción AND con 0x0E. El resultado volverá almacenarse de vuelta en esa dirección. Veámoslo en la ventana de información:



Vemos que la dirección afectada es 40305D y que el valor actual de esa dirección es 56. Pulsamos F8 y vemos como vuelve a cambiar el primer digito:



Ahora sabemos que el resultado de aplicar la instrucción AND a 0x56 y 0x0E es 0x06.

A continuación ECX se incrementa en 4 (para apuntar a los siguientes 4 bytes) y es comparado con 8. Esto significa que este loop se va ejecutar dos veces – la primera vez ECX va a ser igual a 4 y la segunda vez 8, después salimos del loop. En total vamos a manejar 8 bytes de código. Pulsamos F8 y aparecerá el segundo par de 4 bytes al que se le aplica la instrucción XOR con 1234567:

El quinto byte también cambiará al aplicarle la instrucción AND con 0x0E.

```
- [CPU - main thread, module Crackme6]
00401421  $ 55            PUSH EBP
00401422  . 8BEC          MOV EBP,ESP
00401424  . 51            PUSH ECX
00401425  . 52            PUSH EDX
00401426  . 33C9          XOR ECX,ECX
00401428  . 33D2          XOR EDX,EDX
0040142A  . 8B45 08       MOV EAX,[ARG.1]                        Crackme6.0040305D
0040142D  > 813401 67452  XOR DWORD PTR DS:[ECX+EAX],1234567
00401434  . 802401 0E     AND BYTE PTR DS:[ECX+EAX],0E
00401438  . 83C1 04       ADD ECX,4
0040143B  . 83F9 08       CMP ECX,8
0040143E  .^ 75 ED        JNZ SHORT Crackme6.0040142D
00401440    33C9          XOR ECX,ECX
00401442  > 8A1401        MOV DL,BYTE PTR DS:[ECX+EAX]
00401445  . 0050 08       ADD BYTE PTR DS:[EAX+8],DL
00401448  . 41            INC ECX
00401449  . 3B4D 0C       CMP ECX,[ARG.2]
0040144C  .^ 75 F4        JNZ SHORT Crackme6.00401442
0040144E  . 33C9          XOR ECX,ECX
00401450  > 813401 DEBC9  XOR DWORD PTR DS:[ECX+EAX],89ABCDE
00401457  . 802401 0E     AND BYTE PTR DS:[ECX+EAX],0E
0040145B  . 83C1 04       ADD ECX,4
0040145E  . 83F9 08       CMP ECX,8
00401461  .^ 75 ED        JNZ SHORT Crackme6.00401450
00401463  . 33C9          XOR ECX,ECX
00401465  > 8A1401        MOV DL,BYTE PTR DS:[ECX+EAX]
00401468  . 0050 09       ADD BYTE PTR DS:[EAX+9],DL
0040146B  . 41            INC ECX
0040146C  . 3B4D 0C       CMP ECX,[ARG.2]
ECX=00000008
```

Después de este loop nos encontramos con la siguiente instrucción en la dirección 401440, que resetea ECX a cero.

Pulsamos F8 para situarnos dentro de la siguiente rutina en 401442.

```
00401442  > 8A1401   MOV DL,BYTE PTR DS:[ECX+EAX]
00401445  . 0050 08  ADD BYTE PTR DS:[EAX+8],DL
00401448  . 41       INC ECX
00401449  . 3B4D 0C  CMP ECX,[ARG.2]
0040144C  .^ 75 F4   JNZ SHORT Crackme6.00401442
```

Primero moveremos nuestro primer (nuevo) byte de nuestra (antigua) contraseña en DL (como ECX vuelve a ser cero, sabemos que estamos manejando el primer digito, o lo que es lo mismo hacia donde EAX está apuntando). Si nos fijamos en la ventana de registros, veremos el primer byte (0x06) en el registro EDX:

```
Registers (FPU)
EAX 0040305D Crackme6.0040305D
ECX 00000000
EDX 00000006
EBX 00000000
ESP 0012FB00
EBP 0012FB08
ESI 0040102D Crackme6.0040102D
EDI 0012FB80
EIP 00401445 Crackme6.00401445
```

A continuación añadimos ese dígito a DL con [EAX+8], o lo que es lo mismo el octavo byte contado después del principio de EAX, volviendo ser almacenado en la posición octava.

```
00401445  . 0050 08  ADD BYTE PTR DS:[EAX+8],DL
```

Aquí podemos ver el cambio de ese byte:

A continuación incrementamos ECX en uno y lo comparamos con la longitud de nuestra contraseña. Iremos pasando por todos los digitos de nuestra contraseña, añadiendo el valor de cada digito y almacenando ese valor en la octava posición. Según vayamos pulsando F8 podemos ver como va cambiando la memoria:

Una vez finalizado el loop volvemos a poner ECX a cero y entramos en otro loop similar al primero. Esta vez se aplica la instrucción XOR a cada conjunto de cuatro bytes con 0x89ABCDE.

Volvemos añadir todos los bytes para almacenar el total en la posición del noveno byte. Este proceso continuará hasta que ARG.2 sea igual a cero, donde ARG.2 representa la longitud de nuestra contraseña. Este conjunto de instrucciones va correr 8 veces, una por cada dígito de nuestra contraseña. Una vez pasado por el código podemos observar el resultado final:

Resumimos lo que hicimos hasta ahora:

1. Hemos aplicado XOR a cada conjunto de 4 bytes de nuestra contraseña con el valor hexadecimal 12345678 y almacenado el resultado de vuelta en la parte superior de nuestra contraseña.

2. Aplicamos AND sobre el primer digito con 0x0E, así como el quinto byte.

3. Luego sumamos los valores de todos esos bytes para almacenar el resultado en el octavo byte.

4. A continuación aplicamos XOR a cada conjunto de cuatro bytes del buffer con 0x9ABCDEF, y almacenado el resultados de vuelta al buffer.

5. Volvimos sumar los valores del contenido del buffer para almacenar el resultado en el noveno lugar de la memoria.

El siguiente paso consiste en cargar esos dos valores (el sumatorio de los contenidos en la memoria del buffer), uno en EAX+8 y otro en EAX+9 dentro de DL y DH, haciendo que EDX sea igual a 3F69. Luego comparamos ese valor con 42DE:

```
00401471  .  8A50 09       MOV DL,BYTE PTR DS:[EAX+9]
00401474  .  8A70 08       MOV DH,BYTE PTR DS:[EAX+8]
00401477  .  66:81FA DE42  CMP DX,42DE
```

```
Registers (FPU)
EAX 0040305D Crackme6.0040305D
ECX 00000006
EDX 00003F69
EBX 00000000
ESP 0012FB00
EBP 0012FB08
ESI 0040102D Crackme6.0040102D
EDI 0012FB80
EIP 00401477 Crackme6.00401477
```

Como ambos valores son distintos tomaremos el salto en la dirección 40147C, lo que nos lleva al “bad boy”.

- [CPU - main thread, module Crackme6]

```
00401468  .  0050 09        ADD BYTE PTR DS:[EAX+9],DL
0040146B  .  41             INC ECX
0040146C  .  3B4D 0C        CMP ECX,[ARG.2]
0040146F  .^ 75 F4          JNZ SHORT Crackme6.00401465
00401471  .  8A50 09        MOV DL,BYTE PTR DS:[EAX+9]
00401474  .  8A70 08        MOV DH,BYTE PTR DS:[EAX+8]
00401477  .  66:81FA DE42   CMP DX,42DE
0040147C  .v 0F85 8E000000  JNZ Crackme6.00401510
00401482  .  B9 09000000    MOV ECX,9
00401487  >  8A1401         MOV DL,BYTE PTR DS:[ECX+EAX]
0040148A  .  321401         XOR DL,BYTE PTR DS:[ECX+EAX]
0040148D  .  49             DEC ECX
0040148E  .  67:E3 02       JCXZ SHORT Crackme6.00401493
00401491  .^ EB F4          JMP SHORT Crackme6.00401487
00401493  >  66:8B48 08     MOV CX,WORD PTR DS:[EAX+8]
00401497  .  66:81F1 EEEE   XOR CX,0EEEE
0040149C  .  66:81F9 AC30   CMP CX,30AC
004014A1  .v 75 64          JNZ SHORT Crackme6.00401507
004014A3  .  8A08           MOV CL,BYTE PTR DS:[EAX]
004014A5  .  8A68 01        MOV CH,BYTE PTR DS:[EAX+1]
004014A8  .  66:81C1 9235   ADD CX,3592
004014AD  .  66:81F9 9AE5   CMP CX,0E59A
004014B2  .v 75 49          JNZ SHORT Crackme6.004014FD
004014B4  .  8138 08B0817   CMP DWORD PTR DS:[EAX],7A81B008
004014BA  .v 75 4B          JNZ SHORT Crackme6.00401507
004014BC  .  66:33C9        XOR CX,CX
004014BF  >  80F2 0A        XOR DL,0A
004014C2  .  83C1 04        ADD ECX,4
004014C5  .  83F9 0C        CMP ECX,0C
004014C8  .^ 7E F5          JLE SHORT Crackme6.004014BF
004014CA  .  8178 04 02BF   CMP DWORD PTR DS:[EAX+4],388DBF02
004014D1  .v 75 3D          JNZ SHORT Crackme6.00401510
004014D3  .  8ACA           MOV CL,DL
004014D5  .  32D1           XOR DL,CL
004014D7  .  8AD1           MOV DL,CL
004014D9  .  32CA           XOR CL,DL
004014DB  .  8A48 05        MOV CL,BYTE PTR DS:[EAX+5]
004014DE  .  8A50 06        MOV DL,BYTE PTR DS:[EAX+6]
004014E1  .  86D1           XCHG CL,DL
004014E3  .  80FA BF        CMP DL,0BF
004014E6  .v 75 1F          JNZ SHORT Crackme6.00401507
004014E8  .  80F9 8D        CMP CL,8D
004014EB  .v 75 23          JNZ SHORT Crackme6.00401510
004014ED  .  8078 05 BF     CMP BYTE PTR DS:[EAX+5],0BF
004014F1  .v 75 14          JNZ SHORT Crackme6.00401507
004014F3  .  25 FFFF0000    AND EAX,0FFFF
004014F8  .  66:33C0        XOR AX,AX
```

A no ser que cambiemos el valor de la bandera Z con lo que EAX no va tomar el valor 1y la función se parará inmediatamente:

A continuación cargamos ECX con 9 para acceder al digito noveno de nuestro buffer, mover el contenido de esta novena posición en la memoria dentro de DL, aplicar XOR consigo mismo (para que sea igual a cero), decrementar ECX en uno para situarse en el lugar anterior, y repetir todo esto nueve veces:

```
00401482   .  B9 09000000   MOV ECX,9
00401487   >  8A1401        MOV DL,BYTE PTR DS:[ECX+EAX]
0040148A   .  321401        XOR DL,BYTE PTR DS:[ECX+EAX]
0040148D   .  49            DEC ECX
0040148E   .  67:E3 02      JCXZ SHORT Crackme6.00401493
00401491   .^ EB F4         JMP SHORT Crackme6.00401487
```

Después de salir de este loop lo primero que hace es cargar ECX con el sumatorio que hicimos anteriormente (0x2C en la posición novena de la memoria y 0x84 en la posición octava), hace un XOR con 0xEEEE y lo compara con 30AC:

```
0040148E   .  67:E3 02      JCXZ SHORT Crackme6.00401493
00401491   .^ EB F4         JMP SHORT Crackme6.00401487
00401493   >  66:8B48 08    MOV CX,WORD PTR DS:[EAX+8]
00401497   .  66:81F1 EEEE  XOR CX,0EEEE
0040149C   .  66:81F9 AC30  CMP CX,30AC
004014A1   .v 75 64         JNZ SHORT Crackme6.00401507
```

Como ECX es igual a 87D1 tomaremos el salto en la dirección 4014A1:

```
Registers (FPU)
EAX 0040305D Crackme6.0040305D
ECX 000087D1
EDX 00003F00
EBX 00000000
ESP 0012FB00
EBP 0012FB08
ESI 0040102D Crackme6.0040102D
EDI 0012FB80
EIP 004014A1 Crackme6.004014A1
```

Hasta donde ECX vuelva a ser igual a uno:

```
- [CPU - main thread, module Crackme6]
0040148E  .   67:E3 02       JCXZ SHORT Crackme6.00401493
00401491  .^  EB F4          JMP SHORT Crackme6.00401487
00401493  >   66:8B48 08     MOV CX,WORD PTR DS:[EAX+8]
00401497  .   66:81F1 EEEE   XOR CX,0EEEE
0040149C  .   66:81F9 AC30   CMP CX,30AC
004014A1  .v  75 64          JNZ SHORT Crackme6.00401507
004014A3  .   8A08           MOV CL,BYTE PTR DS:[EAX]
004014A5  .   8A68 01        MOV CH,BYTE PTR DS:[EAX+1]
004014A8  .   66:81C1 9235   ADD CX,3592
004014AD  .   66:81F9 9AE5   CMP CX,0E59A
004014B2  .v  75 49          JNZ SHORT Crackme6.004014FD
004014B4  .   8138 08B0817   CMP DWORD PTR DS:[EAX],7A81B008
004014BA  .v  75 4B          JNZ SHORT Crackme6.00401507
004014BC  .   66:33C9        XOR CX,CX
004014BF  >   80F2 0A        XOR DL,0A
004014C2  .   83C1 04        ADD ECX,4
004014C5  .   83F9 0C        CMP ECX,0C
004014C8  .^  7E F5          JLE SHORT Crackme6.004014BF
004014CA  .   8178 04 02BF   CMP DWORD PTR DS:[EAX+4],388DBF02
004014D1  .v  75 3D          JNZ SHORT Crackme6.00401510
004014D3  .   8ACA           MOV CL,DL
004014D5  .   32D1           XOR DL,CL
004014D7  .   8AD1           MOV DL,CL
004014D9  .   32CA           XOR CL,DL
004014DB  .   8A48 05        MOV CL,BYTE PTR DS:[EAX+5]
004014DE  .   8A50 06        MOV DL,BYTE PTR DS:[EAX+6]
004014E1  .   86D1           XCHG CL,DL
004014E3  .   80FA BF        CMP DL,0BF
004014E6  .v  75 1F          JNZ SHORT Crackme6.00401507
004014E8  .   80F9 8D        CMP CL,8D
004014EB  .v  75 23          JNZ SHORT Crackme6.00401510
004014ED  .   8078 05 BF     CMP BYTE PTR DS:[EAX+5],0BF
004014F1  .v  75 14          JNZ SHORT Crackme6.00401507
004014F3  .   25 FFFF0000    AND EAX,0FFFF
004014F8  .   66:33C0        XOR AX,AX
004014FB  .v  EB 18          JMP SHORT Crackme6.00401515
004014FD  >   8AD1           MOV DL,CL
004014FF  .   32CA           XOR CL,DL
00401501  .   B0 01          MOV AL,1
00401503  .   04 20          ADD AL,20
00401505  .   8AC8           MOV CL,AL
00401507  >   66:B9 9138     MOV CX,3891
0040150B  .   66:81F1 AD0F   XOR CX,0FAD
00401510  >   B8 01000000    MOV EAX,1
00401515  >   5A             POP EDX          ntdll.KiFastSystemCallR
00401516  .   59             POP ECX          ntdll.KiFastSystemCallR
00401517  .   C9             LEAVE
```

Esto es básicamente la segunda comprobación de la contraseña. Y como no queremos saltar cambiamos el valor de la bandera Z para seguir ejecutando el código a partir de la dirección 4014A3.

Las dos siguientes instrucciones mueven el primer y segundo contenido de la memoria correspondiente al buffer de la contraseña dentro de CL y CH, lo que hace que ECX sea igual a 8708. Luego se añade 3592h y se compara con E59A. Ahora ECX es igual a 9A por lo que tomaremos el siguiente salto:

```
0040148E   .   67:E3 02        JCXZ SHORT Crackme6.00401493
00401491   .^  EB F4           JMP SHORT Crackme6.00401487
00401493   >   66:8B48 08      MOV CX,WORD PTR DS:[EAX+8]
00401497   .   66:81F1 EEEE    XOR CX,0EEEE
0040149C   .   66:81F9 AC30    CMP CX,30AC
004014A1   .v  75 64           JNZ SHORT Crackme6.00401507
004014A3   .   8A08            MOV CL,BYTE PTR DS:[EAX]
004014A5   .   8A68 01         MOV CH,BYTE PTR DS:[EAX+1]
004014A8   .   66:81C1 9235    ADD CX,3592
004014AD   .   66:81F9 9AE5    CMP CX,0E59A
004014B2   .v  75 49           JNZ SHORT Crackme6.004014FD
004014B4   .   8138 08B0817    CMP DWORD PTR DS:[EAX],7A81B008
004014BA   .v  75 4B           JNZ SHORT Crackme6.00401507
004014BC   .   66:33C9         XOR CX,CX
004014BF   >   80F2 0A         XOR DL,0A
004014C2   .   83C1 04         ADD ECX,4
004014C5   .   83F9 0C         CMP ECX,0C
004014C8   .^  7E F5           JLE SHORT Crackme6.004014BF
004014CA   .   8178 04 02BF    CMP DWORD PTR DS:[EAX+4],388DBF02
004014D1   .v  75 3D           JNZ SHORT Crackme6.00401510
004014D3   .   8ACA            MOV CL,DL
004014D5   .   32D1            XOR DL,CL
004014D7   .   8AD1            MOV DL,CL
004014D9   .   32CA            XOR CL,DL
004014DB   .   8A48 05         MOV CL,BYTE PTR DS:[EAX+5]
004014DE   .   8A50 06         MOV DL,BYTE PTR DS:[EAX+6]
004014E1   .   86D1            XCHG CL,DL
004014E3   .   80FA BF         CMP DL,0BF
004014E6   .v  75 1F           JNZ SHORT Crackme6.00401507
004014E8   .   80F9 8D         CMP CL,8D
004014EB   .v  75 23           JNZ SHORT Crackme6.00401510
004014ED   .   8078 05 BF      CMP BYTE PTR DS:[EAX+5],0BF
004014F1   .v  75 14           JNZ SHORT Crackme6.00401507
004014F3   .   25 FFFF0000     AND EAX,0FFFF
004014F8   .   66:33C0         XOR AX,AX
004014FB   .v  EB 18           JMP SHORT Crackme6.00401515
004014FD   >   8AD1            MOV DL,CL
004014FF   .   32CA            XOR CL,DL
00401501   .   B0 01           MOV AL,1
00401503   .   04 20           ADD AL,20
00401505   .   8AC8            MOV CL,AL
00401507   >   66:B9 9138      MOV CX,3891
0040150B   .   66:81F1 AD0F    XOR CX,0FAD
00401510   >   B8 01000000     MOV EAX,1
00401515   >   5A              POP EDX
00401516   .   59              POP ECX
00401517   .   C9              LEAVE
```

Como tampoco queremos saltar en esta ocación, volveremos cambiar el valor de la bandera Z.

```
004014B2   .v  75 49           JNZ SHORT Crackme6.004014FD
004014B4   .   8138 08B0817    CMP DWORD PTR DS:[EAX],7A81B008
```

La siguiente línea, CMP DWORD PTR DS:[EAX], 7A81B008, realiza otra comprobación. Después de todas las manipulaciones hechas a nuestra contraseña finalmente los primeros 4 bytes serán igual a 7A81B008 y si no saltaremos hacia nuestro “bad boy”:

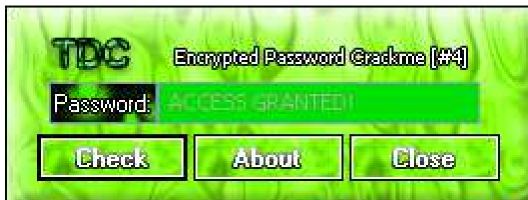
Después de volver a cambiar el valor de la bandera Z para no saltar, entramos otra vez dentro de un conjunto de comprobaciones. Bypaseando todas las siguientes comprobaciones llegaremos finalmente a la instrucción JMP en la dirección 4014FB:

Esta vez saltamos por encima de la instrucción MOV EAX, 1 en la dirección 401510 con lo cual evitamos llegar al RETN con EAX igual a uno.

Salimos de la rutina y regresamos al CALL que nos llevó a realizar todas estas comprobaciones:

```
- [CPU - main thread, module Crackme6]
File  View  Debug  Plugins  Options  Window  Help  Tools  BreakPoint->
Paused
00401298  .   E8 84010000    CALL Crackme6.00401421
0040129D  .   0BC0           OR EAX,EAX
0040129F  .v  75 1F          JNZ SHORT Crackme6.004012C0
004012A1  .   68 0F304000    PUSH Crackme6.0040300F             Text = "ACCESS GRANTED!"
004012A6  .   FF35 8030400   PUSH DWORD PTR DS:[403080]         hWnd = 0005014E (class='Edit',parent=00040116)
004012AC  .   E8 CB020000    CALL <JMP.&user32.SetWindowTextA>  SetWindowTextA
004012B1  .   6A 00          PUSH 0                             Enable = FALSE
004012B3  .   FF35 8030400   PUSH DWORD PTR DS:[403080]         hWnd = 0005014E (class='Edit',parent=00040116)
004012B9  .   E8 88020000    CALL <JMP.&user32.EnableWindow>    EnableWindow
004012BE  .v  EB 10          JMP SHORT Crackme6.004012D0
004012C0  >   68 00304000    PUSH Crackme6.00403000             Text = "ACCESS DENIED!"
004012C5  .   FF35 8030400   PUSH DWORD PTR DS:[403080]         hWnd = 0005014E (class='Edit',parent=00040116)
004012CB  .   E8 AC020000    CALL <JMP.&user32.SetWindowTextA>  SetWindowTextA
004012D0  >   6A 0E          PUSH 0E
004012D2  .   68 00304000    PUSH Crackme6.00403000             ASCII "ACCESS DENIED!"
004012D7  .   E8 3F020000    CALL Crackme6.0040151B
004012DC  .   6A 0F          PUSH 0F
004012DE  .   68 0F304000    PUSH Crackme6.0040300F             ASCII "ACCESS GRANTED!"
004012E3  .   E8 33020000    CALL Crackme6.0040151B
004012E8  .v  EB 22          JMP SHORT Crackme6.0040130C
004012EA  >   837D 10 6A     CMP DWORD PTR SS:[EBP+10],6A
004012EE  .v  75 1C          JNZ SHORT Crackme6.0040130C
004012F0  .   6A 00          PUSH 0                             Result = 0
004012F2  .   FF75 08        PUSH DWORD PTR SS:[EBP+8]          hWnd = 00040116 ('TDC [#4]',class='#32770')
004012F5  .   E8 52020000    CALL <JMP.&user32.EndDialog>       EndDialog
004012FA  .v  EB 10          JMP SHORT Crackme6.0040130C
004012FC  >   837D 0C 10     CMP DWORD PTR SS:[EBP+C],10
00401300  .v  75 0A          JNZ SHORT Crackme6.0040130C
00401302  .   6A 00          PUSH 0                             Result = 0
00401304  .   FF75 08        PUSH DWORD PTR SS:[EBP+8]          hWnd = 00040116 ('TDC [#4]',class='#32770')
00401307  .   E8 40020000    CALL <JMP.&user32.EndDialog>       EndDialog
0040130C  >   33C0           XOR EAX,EAX
0040130E  .   C9             LEAVE
0040130F  .   C2 1000        RETN 10
00401312  .   55             PUSH EBP
00401313  .   8BEC           MOV EBP,ESP
00401315  .   817D 0C 1001   CMP DWORD PTR SS:[EBP+C],110
0040131C  .v  75 31          JNZ SHORT Crackme6.0040134F
0040131E  .   68 45304000    PUSH Crackme6.00403045             Text = "About"
00401323  .   FF75 08        PUSH DWORD PTR SS:[EBP+8]          hWnd = 00040116 ('TDC [#4]',class='#32770')
00401326  .   E8 51020000    CALL <JMP.&user32.SetWindowTextA>  SetWindowTextA
0040132B  .   6A 16          PUSH 16                            RsrcName = 22.
0040132D  .   FF35 7C30400   PUSH DWORD PTR DS:[40307C]         hInst = 00400000
00401333  .   E8 26020000    CALL <JMP.&user32.LoadBitmapA>     LoadBitmapA
00401338  .   50             PUSH EAX                           lParam = 0
00401339  .   6A 00          PUSH 0                             wParam = 0
0040133B  .   68 F7000000    PUSH 0F7                           Message = BM_SETIMAGE
```

Pulsamos F9 y vemos a nuestro “good boy”:

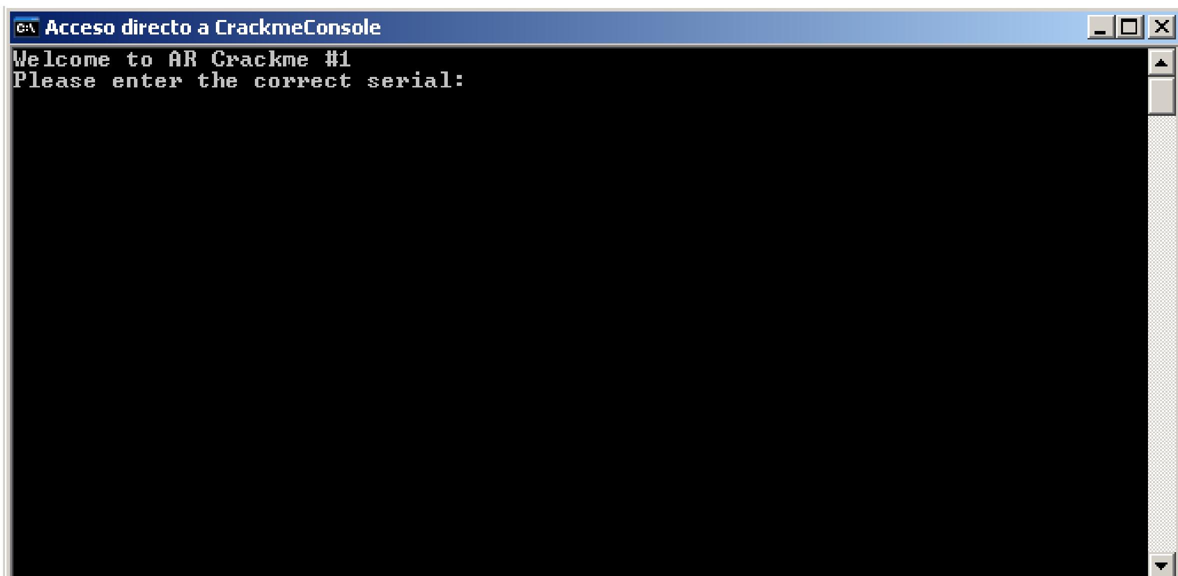


7.8 Caso práctico 8: Introducción al nivel 2 (noob)

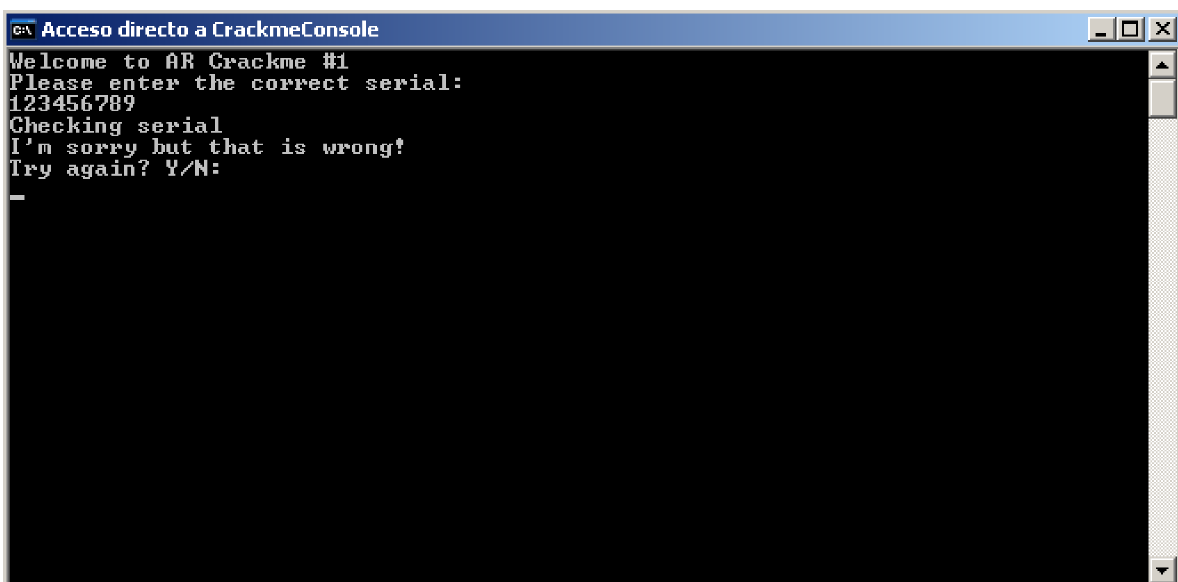
En este ejercicio vamos a profundizar más en el parcheo de programas, tomando como ejecutable un programa de consola.

Las consolas son ventanas de 32-bit, cuyo comportamiento es igual a cualquier otro programa de 32-bit que se ejecuta en Windows. La única diferencia está en que no utilizan una interfaz gráfica. El ejecutable de esta lección se llama CrackmeConsole.exe.

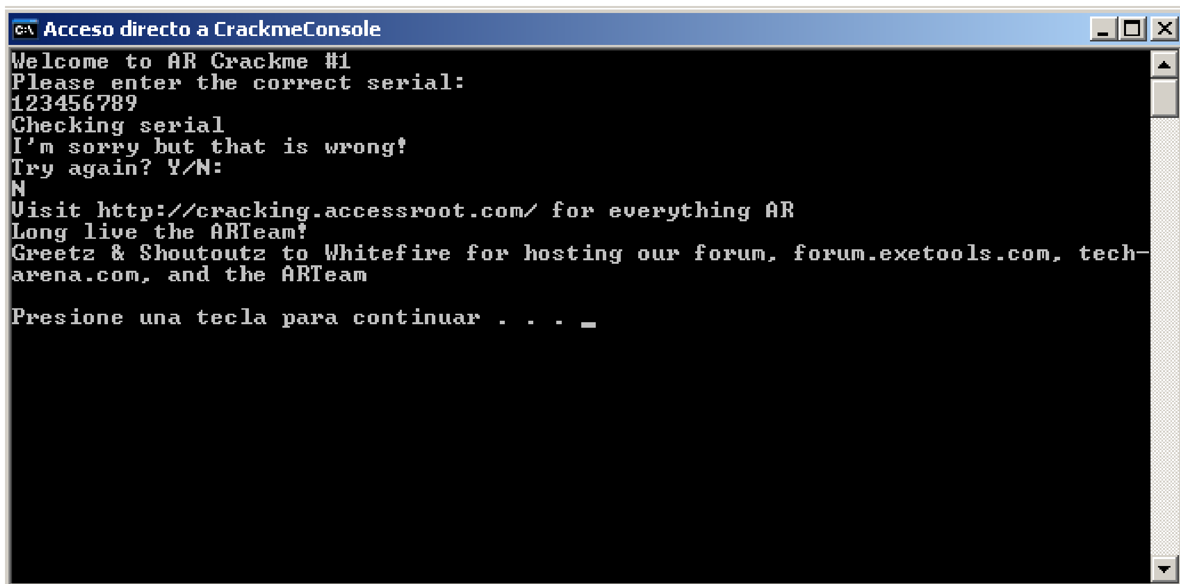
Hacemos doble clic sobre la aplicación para estudiar su comportamiento:



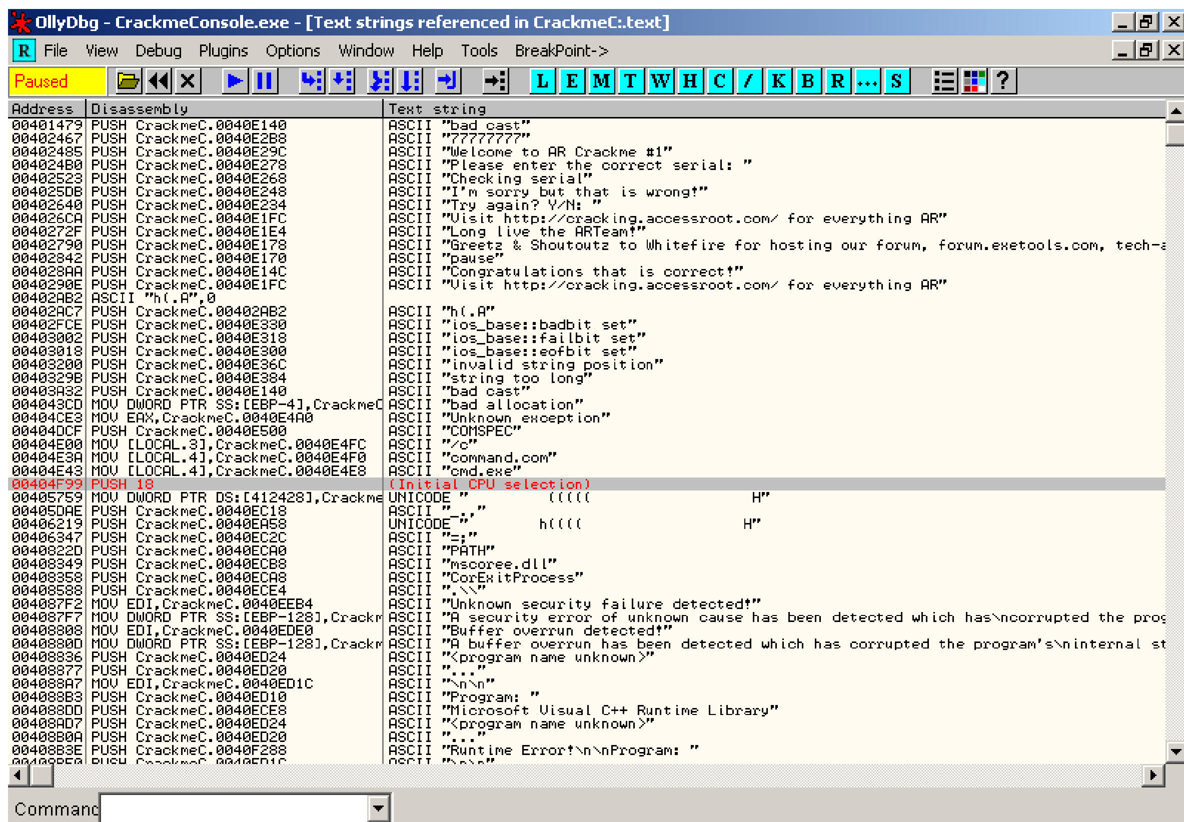
Introducimos un serial cualquiera:



Seleccionamos “N”:



Con esto ya tenemos suficiente para empezar a investigar el programa con Olly. Una vez cargada buscamos las cadenas de texto:



Hacemos doble clic sobre el “bad boy” (I’m sorry but that is wrong) para situarnos en el área que nos interesa.

Vemos que hay un salto desde la dirección 4025C6 que nos lleva al “bad boy” y que está resaltado con una flecha roja.

Vemos también que accedemos al “bad boy” si no tomamos el salto JE en la dirección 4025D5. Averiguemos hacia donde nos lleva este último salto. Hacemos clic sobre la instrucción JE.

Bajamos unas cuantas líneas…

Y llegamos al mensaje bueno.

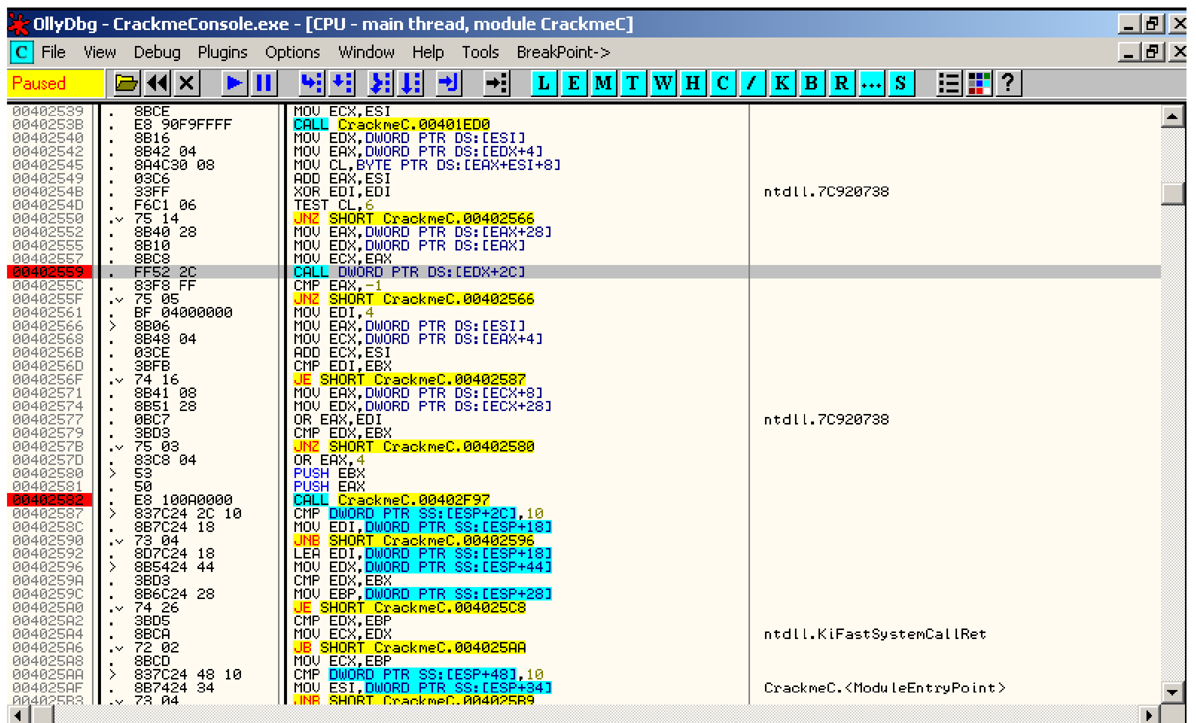
Volvamos a analizar el área correspondiente al inicio del salto.

Sabemos, por la flecha roja delante de los opcodes, que tanto el salto en la dirección 4025C6 como el salto en la direccción 4025CA nos llevan directamente al “bad boy”. Si subimos unas cuantas líneas podemos observar nuestra pirmera pareja de instrucciónes call/compare en la dirección 402582.

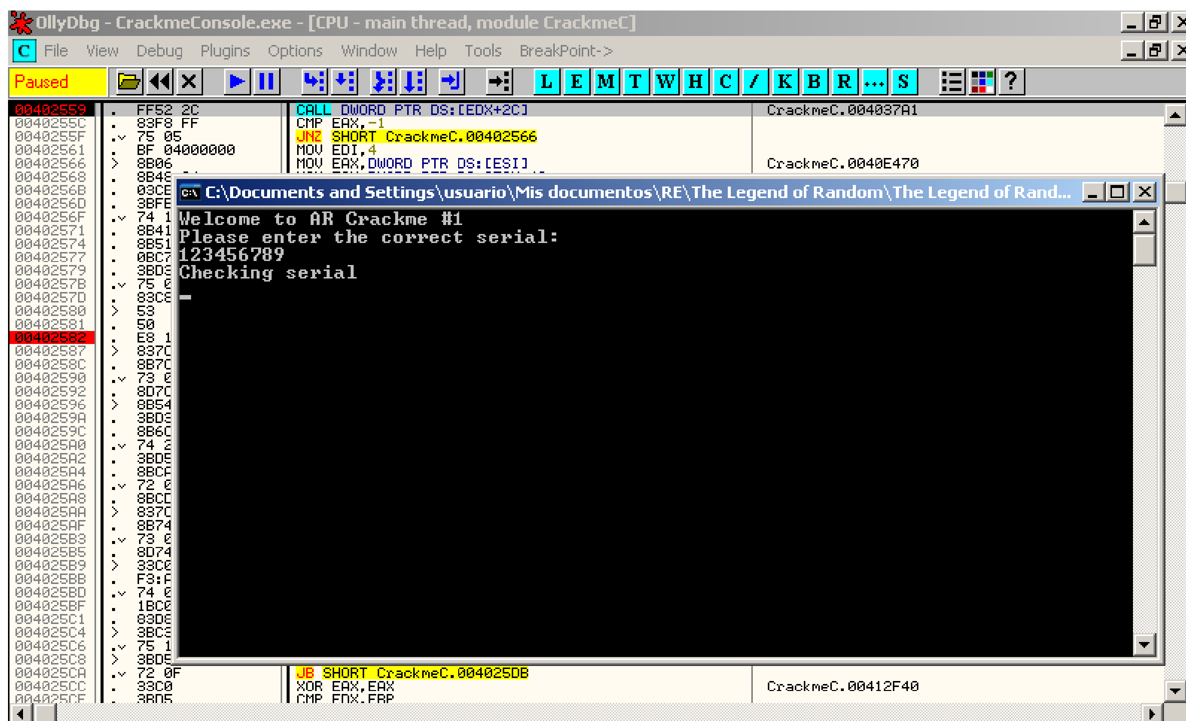
Y subiendo un poco más podemos ver un salto que va directamente a la intrucción CMP sin pasar por la instrucción CALL.

Aunque esto no suele ser un comportamiento normal vamos a poner nuestro primer Breakpoint en el CALL de la dirección 402582.

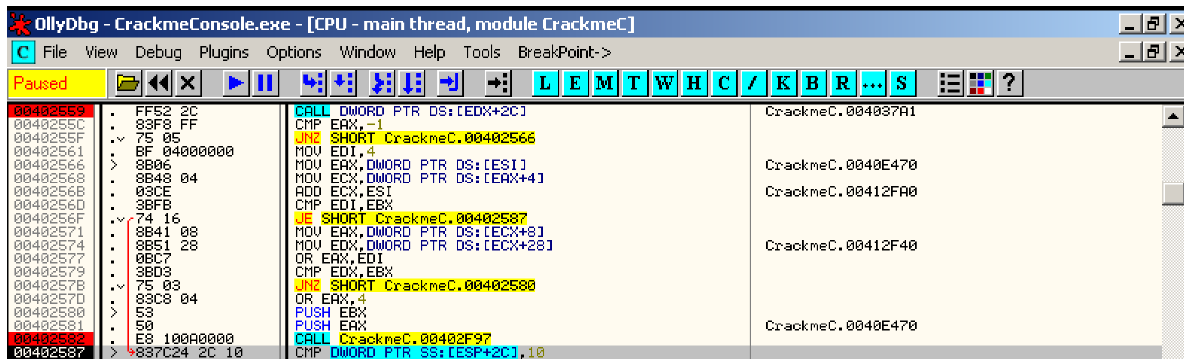
Si volvemos a subir unas líneas, vemos otra pareja de instrucciones CALL/CMP en la dirección 402559. Ponemos nuestro segundo Breakpoint en esta dirección.



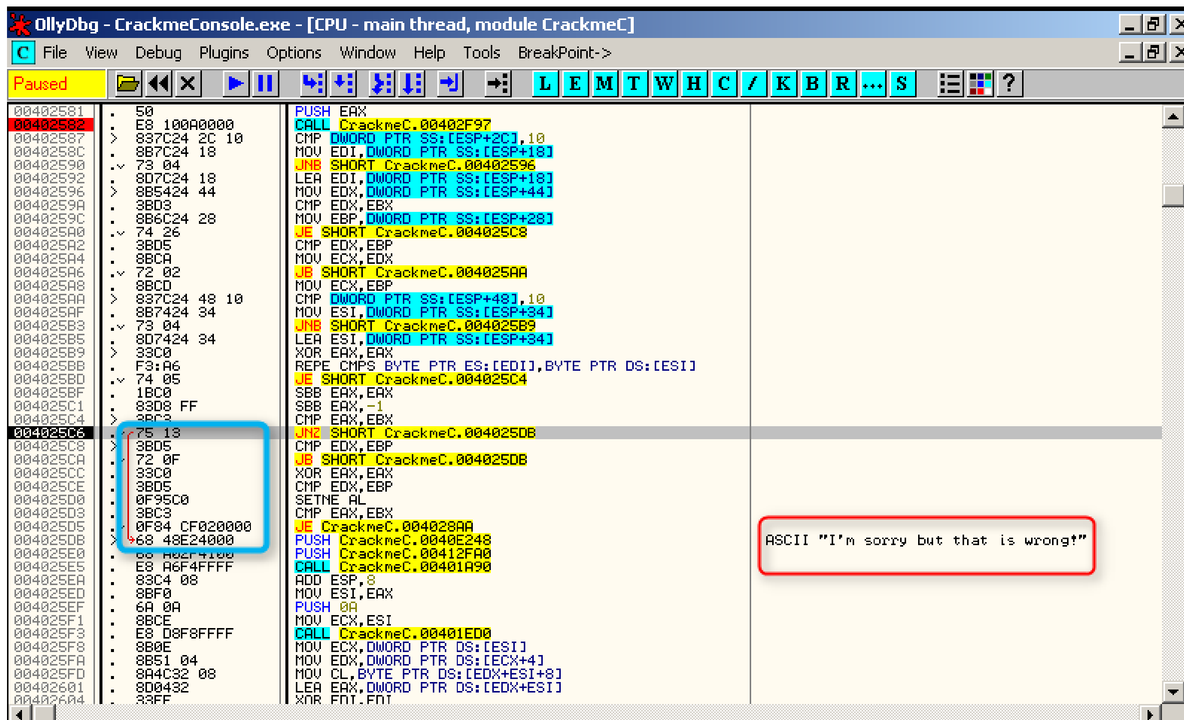
A continuación reiniciamos Olly, pulsamos F9 e introducimos el serial.



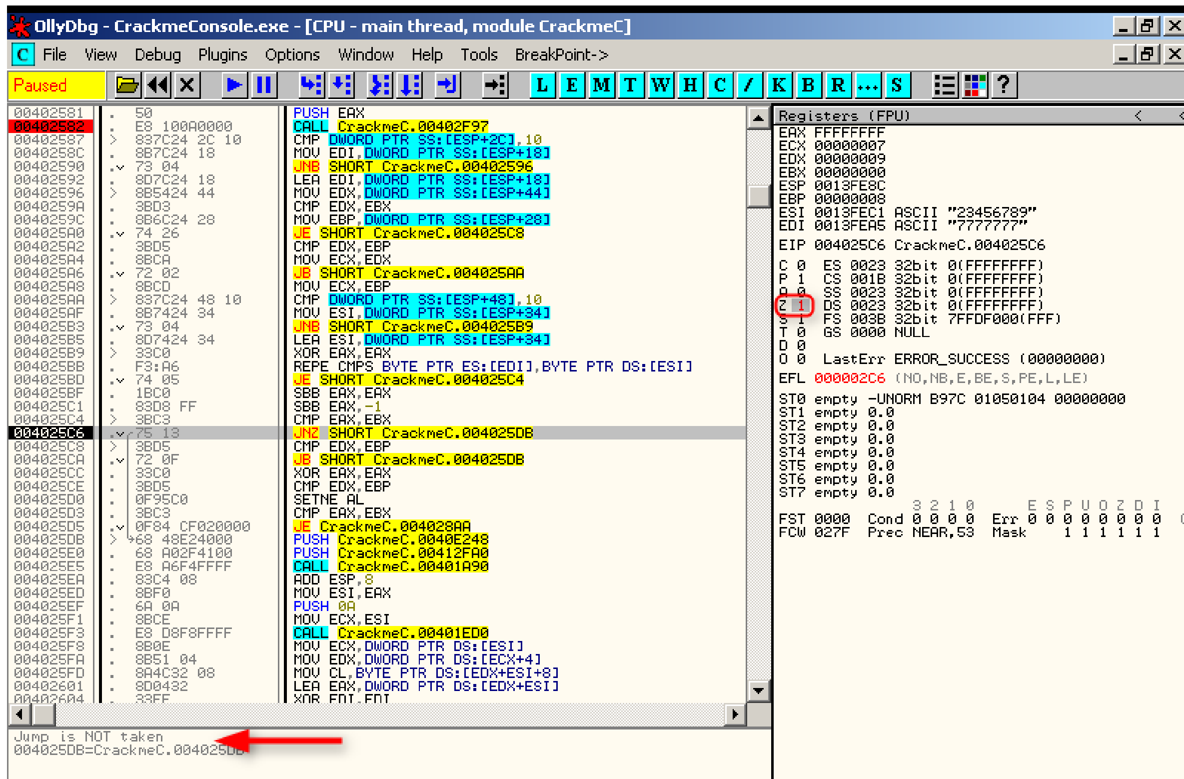
Olly se detiene en nuestro primer Breakpoint. Pulsamos F8 hasta llegar a la dirección 40256F donde saltamos, pasando el segundo CALL. Esto puede ser un indicativo de que se trate de una especie de rutina que sigue el programa para averiguar por ejemplo la longitud del serial.



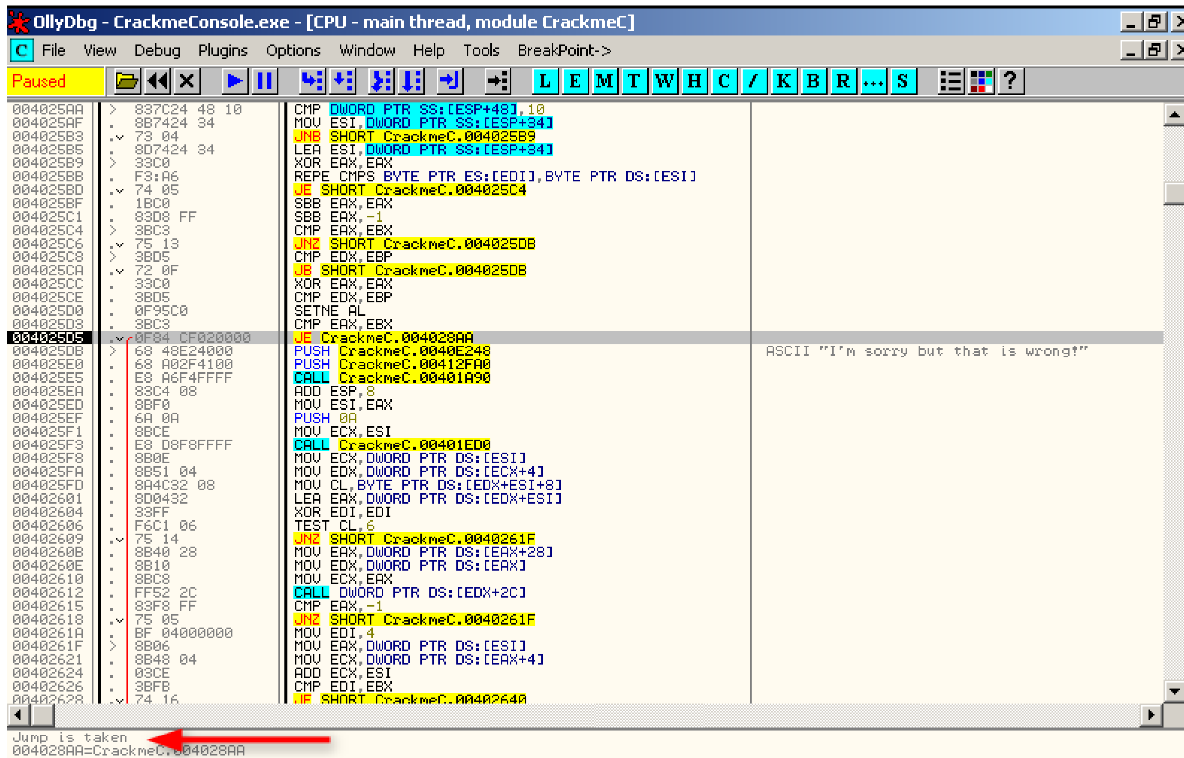
Continuamos pulsando F8 hasta llegar a la dirección 4025C6, donde sabemos que vamos a saltar hacia nuestro “bad boy”.



Hacemos doble clic sobre la bandera Z para no tomar el salto y continuamos pulsando F8.



Llegamos a la dirección 4025D5 y vemos que tomaremos el salto hacia el "good boy".

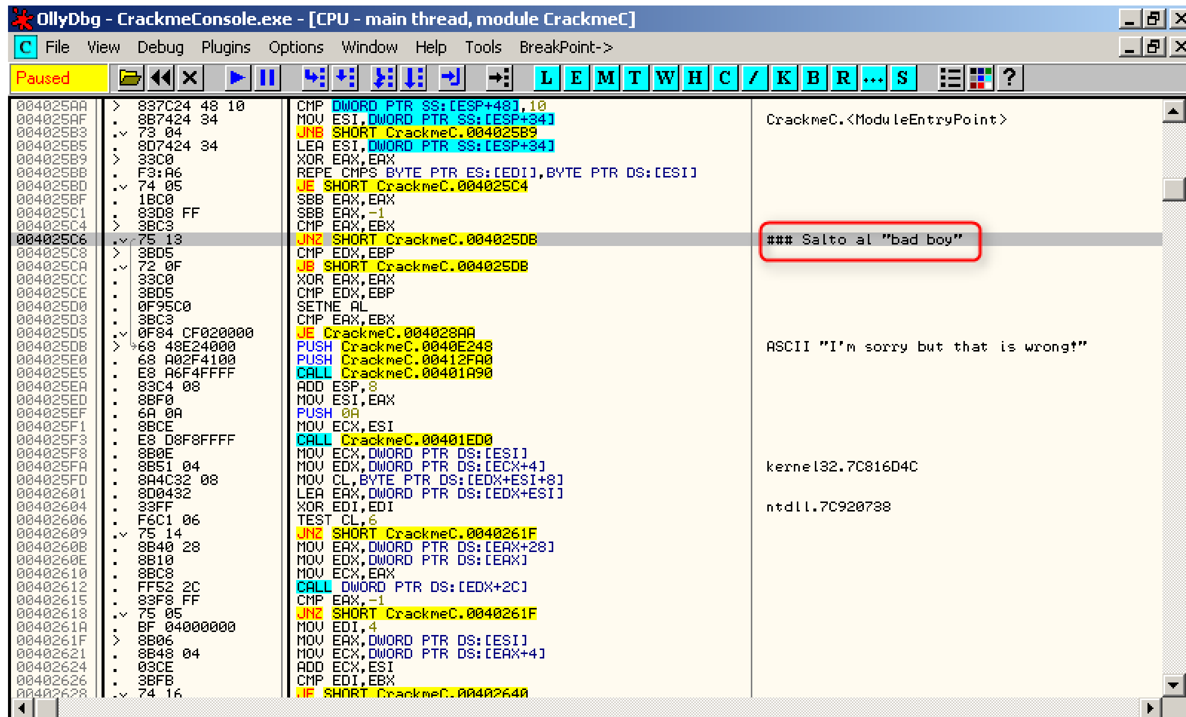


Una vez llegado a este punto, podríamos coger y parchear la aplicación en el punto donde hemos cambiado el valor de la bandera Z. Pero esto nos dejaría con la siguiente duda: ¿Qué sucedería si nuestro serial es demasiado corto o largo? o si es diferente al serial que hemos

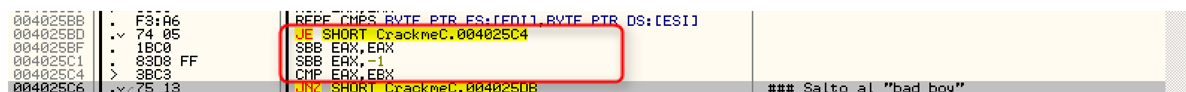
introducido. Realmente este no sería un buen parche ya que no sabemos a ciencia cierta lo que estaríamos parcheando.

Intentemos pues realizar una investigación más profunda de acuerdo al nivel noop visto en el capitulo anterior.

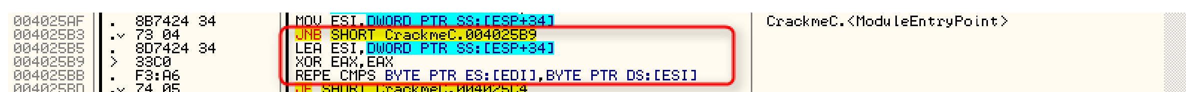
Volvamos a la dirección 4025C6 y analizemos porque hubiesemos tomado el salto si no estaría parcheado (recordemos que hemos cambiado el valor de la bander Z en esta dirección). Como recordatorio podemos poner un comentario en esta línea indicando el estado de la aplicación antes de poner el parche.



Veamos la sección justo antes del salto para intentar responder a nuestro pregunta:



Podemos apreciar dos instrucciónes SBB con una instrucción CMP, lo que no nos aclara mucho. Subamos por lo tanto a ver lo que hay en la sección anterior.



Aquí encontramos la instrucción REPE CMPS. Miremos lo que significa REPE:

Intel x86 Instructions

Archivo Edición Marcador Opciones Ayuda

Contenido Índice Atrás Imprimir

REP/REPE/REPZ/REPNE/REPNZ—Repeat String Operation Prefix

See also

Description

Repeats a string instruction the number of times specified in the count register ((E)CX) or until the indicated condition of the ZF flag is no longer met. The REP (repeat), REPE (repeat while equal), REPNE (repeat while not equal), REPZ (repeat while zero), and REPNZ (repeat while not zero) mnemonics are prefixes that can be added to one of the string instructions. The REP prefix can be added to the INS, OUTS, MOVS, LODS, and STOS instructions, and the REPE, REPNE, REPZ, and REPNZ prefixes can be added to the CMPS and SCAS instructions. (The REPZ and REPNZ prefixes are synonymous forms of the REPE and REPNE prefixes, respectively.) The behavior of the REP prefix is undefined when used with non-string instructions.

The REP prefixes apply only to one string instruction at a time. To repeat a block of instructions, use the LOOP instruction or another looping construct.

All of these repeat prefixes cause the associated instruction to be repeated until the count in register (E)CX is decremented to 0 (see the following table). (If the current address-size attribute is 32, register ECX is used as a counter, and if the address-size attribute is 16, the CX register is used.) The REPE, REPNE, REPZ, and REPNZ prefixes also check the state of the ZF flag after each iteration and terminate the repeat loop if the ZF flag is not in the specified state. When both termination conditions are tested, the cause of a repeat termination can be determined either by testing the (E)CX register with a JECXZ instruction or by testing the ZF flag with a JZ, JNZ, and JNE instruction.

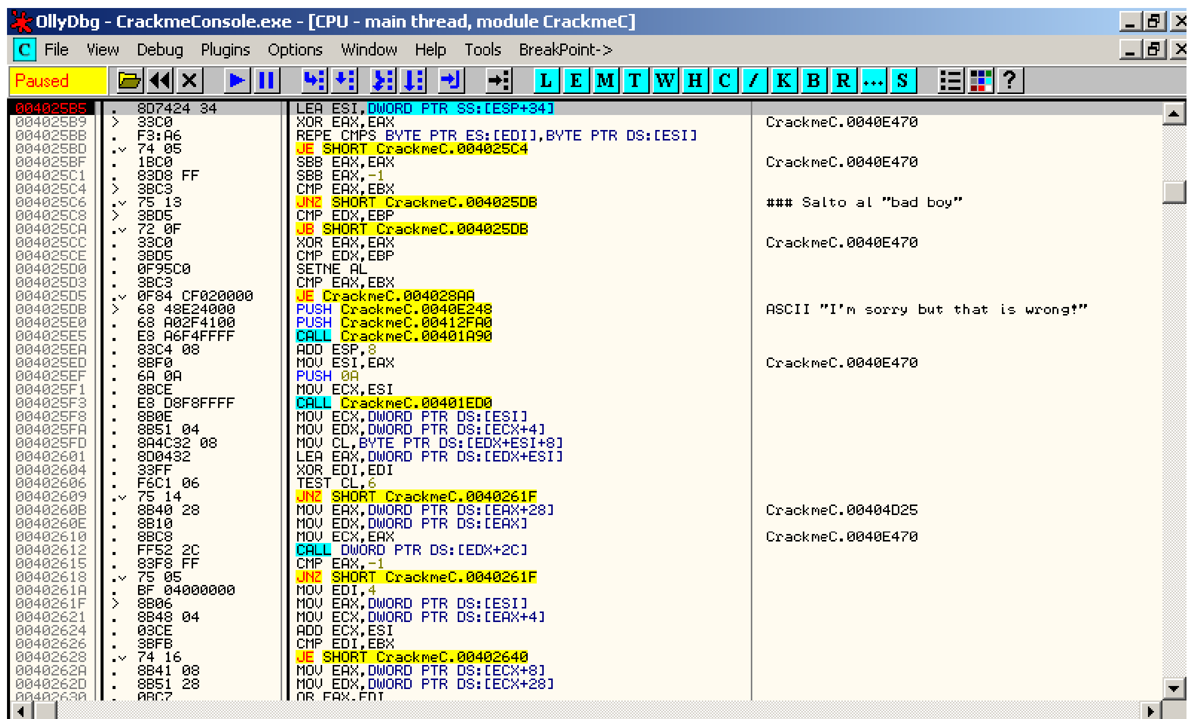
Repeat Conditions

Repeat Prefix	Termination Condition 1	Termination Condition 2
REP	ECX=0	None

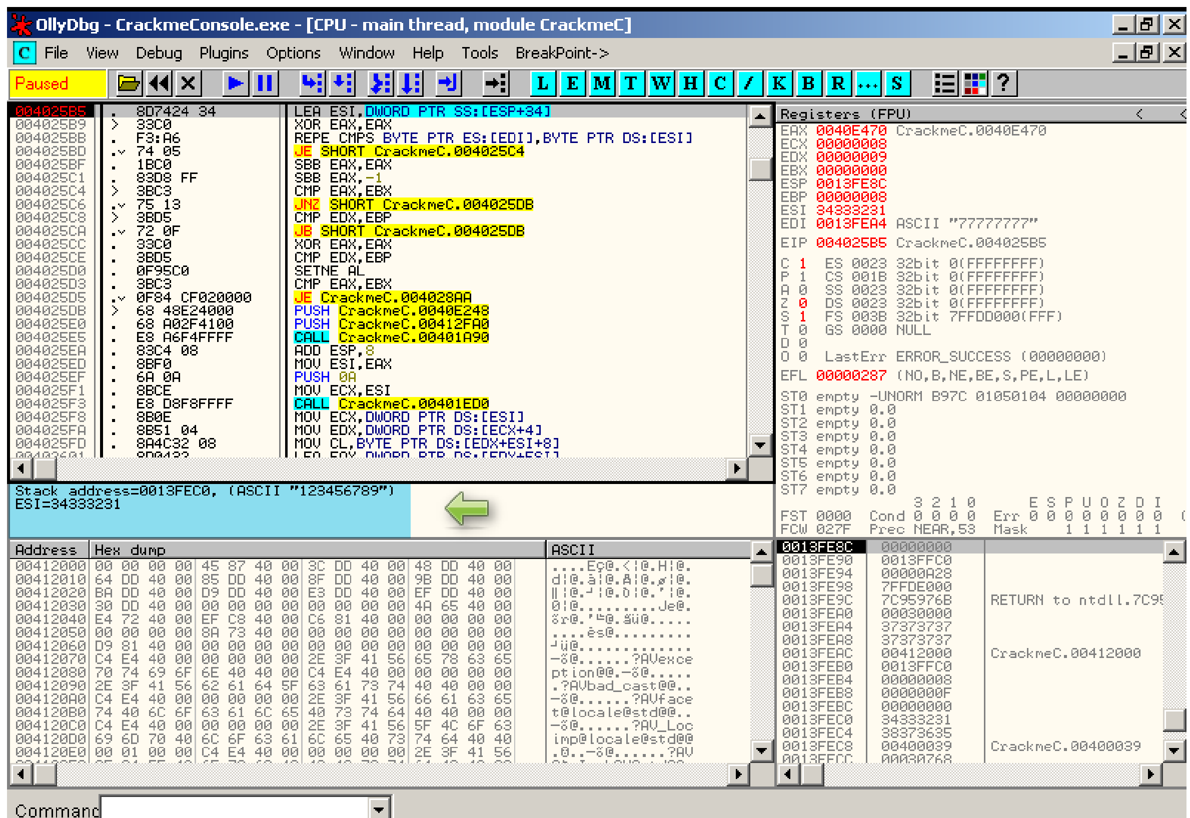
Lo que viene a decir la definición es que la instrucción REPE repite la instrucción CMPS, como si de un loop se tratase hasta que ECX sea igual a cero.

Es decir, "repite la comparación de dos direcciones de la memoria, incrementando esa dirección por cada iteración del loop, mientras que la bandera Z permanezca igual".

Ponemos un Breakpoint en la primera línea de esta sección y que corresponde con la dirección 4025B5. Reiniciamos la aplicación. Introducimos nuestro serial y Olly se detiene en el Breakpoint.

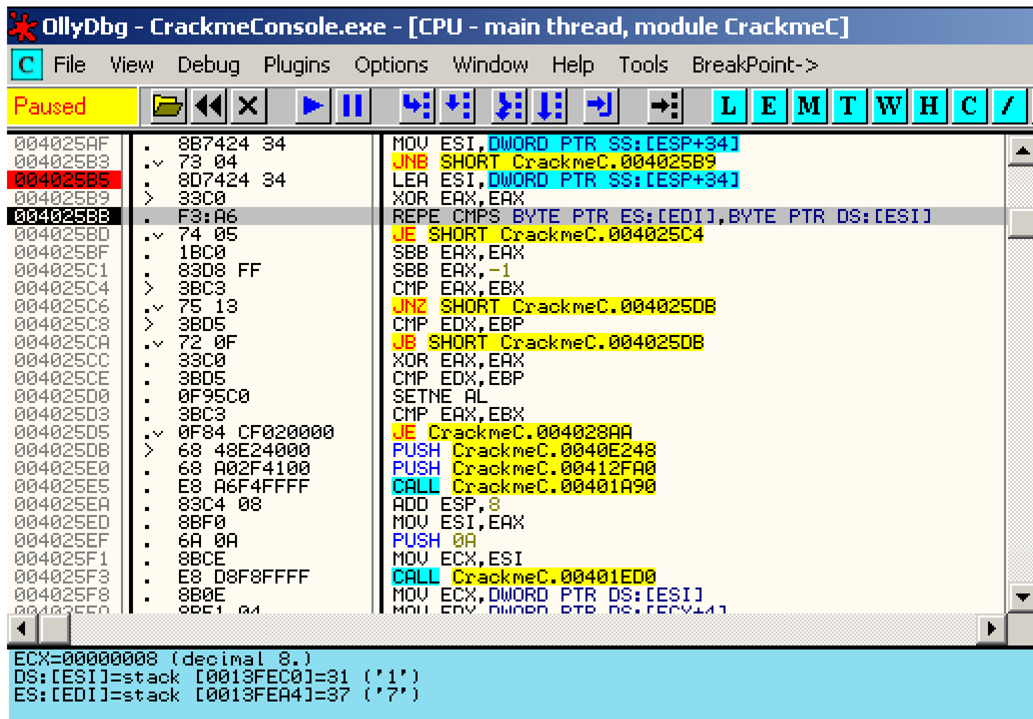


Fijemonos que la primera instrucción **LEA ESI, DWORD PTR SS:[ESP+34]** esta cargando una dirección efectiva de la pila a ESI. La SS significa pila y [ESP+34] denota la posición en la pila. Si miramos el contenido de la ventana de información podemos apreciar que SS:[ESP+34] es igual a la dirección 0013FEC0, y esta almacena nuestro serial en ASCII. Si pulsamos F8 una vez podemos ver como el serial en la pila es pasado al registro ESI.



```
Registers (FPU)
EAX 0040E470 CrackmeC.0040E470
ECX 00000008
EDX 00000009
EBX 00000000
ESP 0013FE8C
EBP 00000008
ESI 0013FEC0 ASCII "123456789"
EDI 0013FEA4 ASCII "77777777"
EIP 004025B9 CrackmeC.004025B9
```

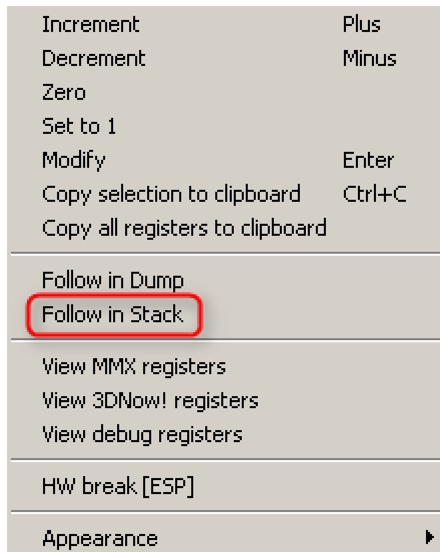
La siguiente instrucción pone EAX a cero y llegamos a la instrucción REPE. En este caso el contenido de la dirección almacenada en ESI es comparada con el contenido de la dirección almacenada en EDI.



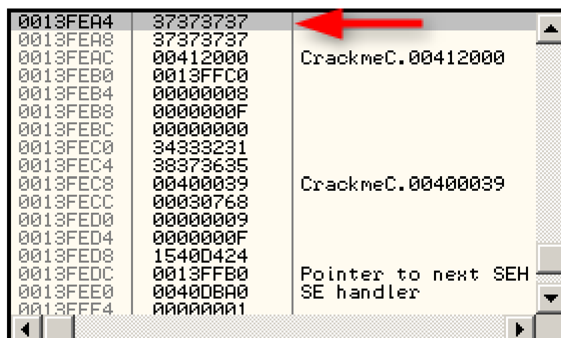
Después de la comparación el registro ECX es disminuido en uno y la comparación seguirá en la siguiente locacilación en memoria de ESI y EDI, finalizando el loop cuando ECX valga cero.

¿Cuantas iteraciones tiene el loop? Si nos fijamos en el gráfico anterior ECX=00000008, lo que significa que el serial tiene una longitud de 8 caracteres y el loop comparará cada dígito entre ambos registros.

Para saber con que es comparado el serial que hemos introducido, vamos al registro donde vemos que EDI apunta hacia una dirección en la pila. Hacemos clic con el botón derecho y seleccionamos “Follow in Stack”.



Vemos como la pila nos muestra la dirección a la que hace referencia el registro EDI, en nuesro caso 0013FEA4. En esta dirección podemos ver una cadena de sucesivos "37"



De la tabla de conversión entre hexadecimales y decimales sabemos que el hexadecimal 37 equivale al 7 decimal. Podemos concluir entonces que nuestro serial va a ser comparado con una cadena ASCII de sietes. En la pila podemos ver que hay ocho caracteres de "37". Estos ocho "7" van a ser comparados, uno por uno con lo que hemos introducido como serial. Si todos los ocho digitos de nuestro serial son iguales a 7 entonces tomaríamos el siguiente salto.

¿ Pero que pasaría si en lugar de nuestro serial introdujesemos ocho sietes ?

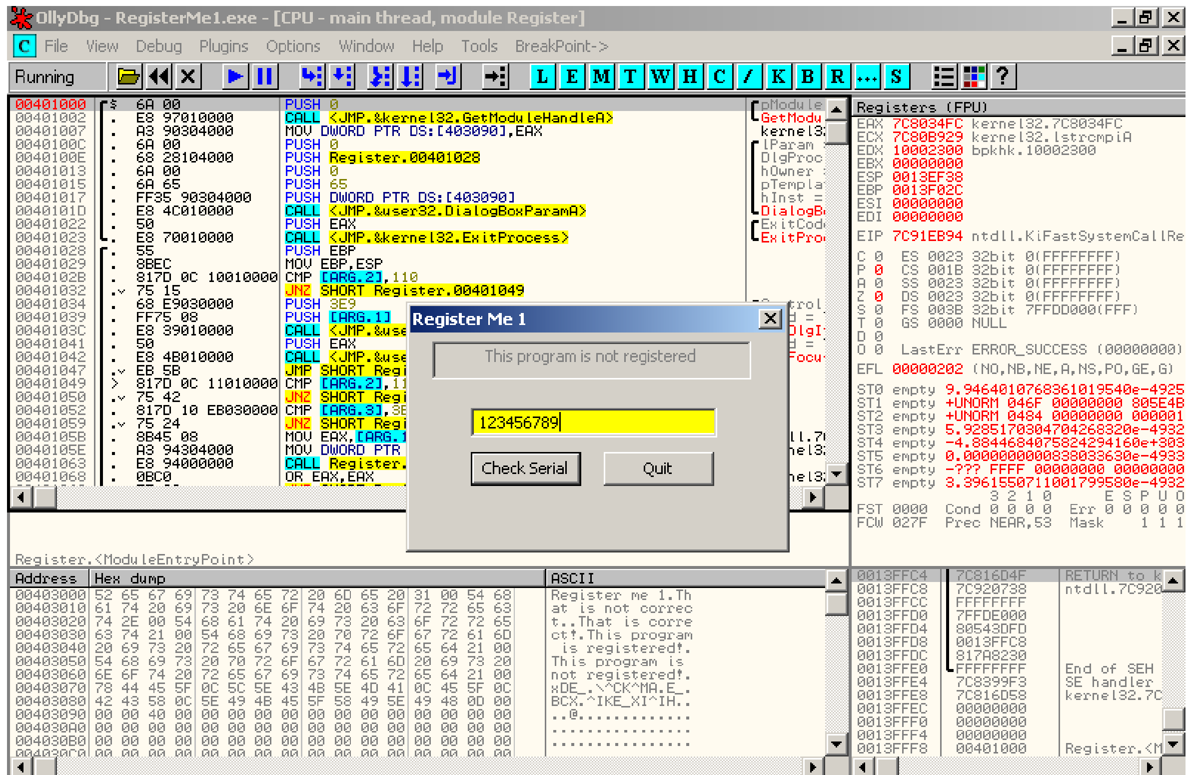
Reiniciemos la aplicación, pulsamos F9 e introducimos el nuevo serial:

```
C:\Documents and Settings\usuario\Mis documentos\RE\The Legend of Random\The Legend of Rand...
Welcome to AR Crackme #1
Please enter the correct serial:
77777777
Checking serial
Congratulations that is correct!
Visit http://cracking.accessroot.com/ for everything AR
Long live the ARTeam!
Greetz & Shoutoutz to Whitefire for hosting our forum, forum.exetools.com, tech-
arena.com, and the ARTeam

Presione una tecla para continuar . . .
```

Conclusión: Si no nos queremos conformar con un simple parche, del cual no sabemos si realmente me va a funcionar o no, podemos siempre profundizar un poco más en nuestra investigación para averiguar la contraseña correcta de la aplicación.

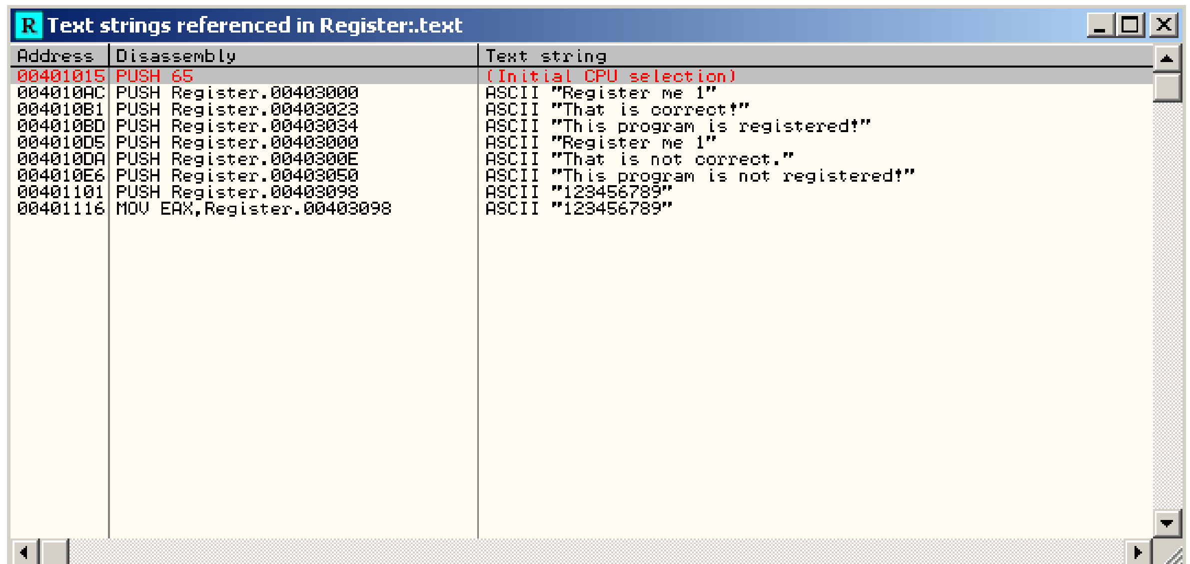
7.9 Caso práctico 9: Ejemplo de Noob Avanzado

Abrimos Olly y cargamos la aplicación RegisterMe1.exe. Pulsamos F9. Vemos en primer lugar un cuadro de texto que pone: “This program is not registered” y debajo otro cuadro de texto vacío en el que tenemos que introducir un serial. Una vez introducido hacemos clic en “Check Serial”.

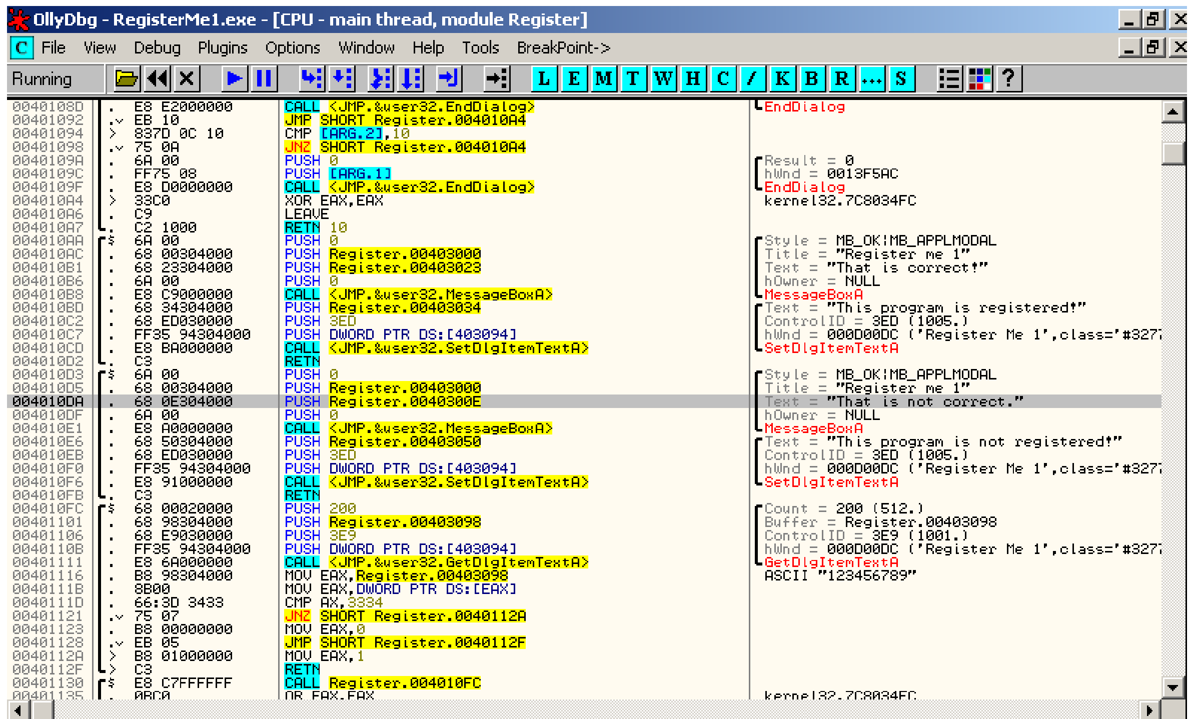


Vemos que nos hemos vuelto a equivocar !!!

Busquemos por cadenas de texto:

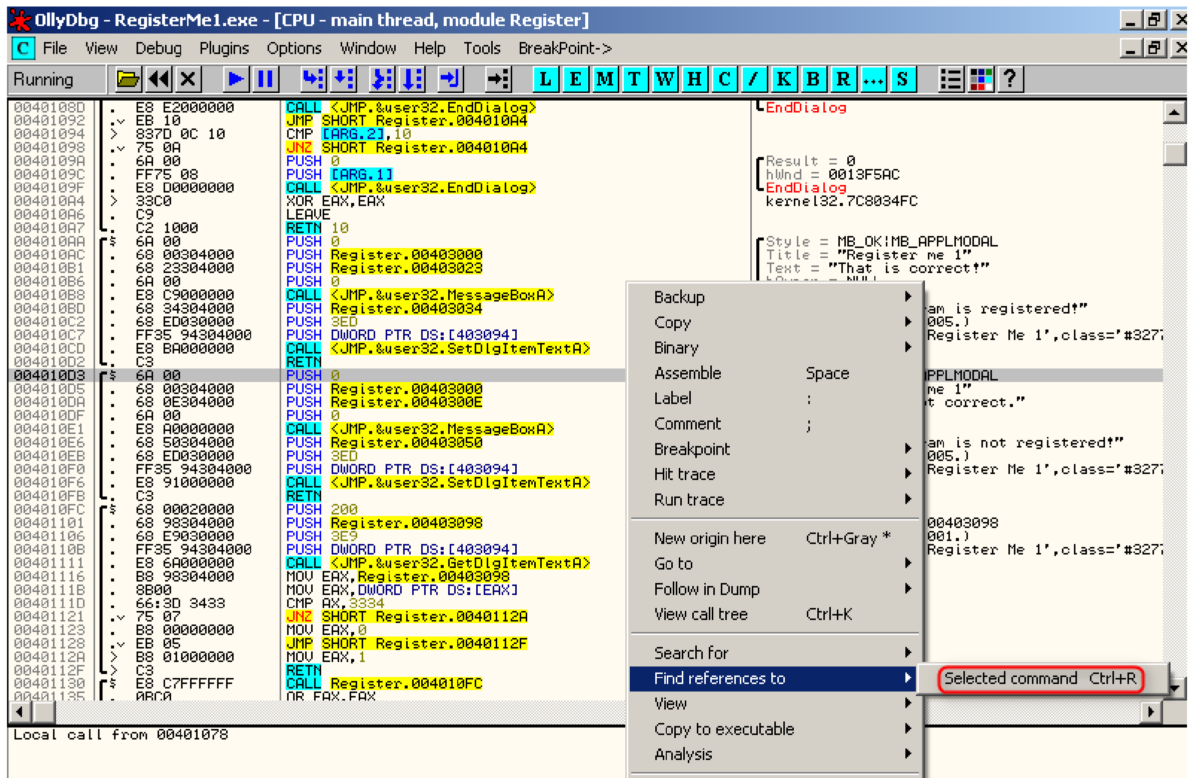


Hacemos doble clic en: “That is not correct.”

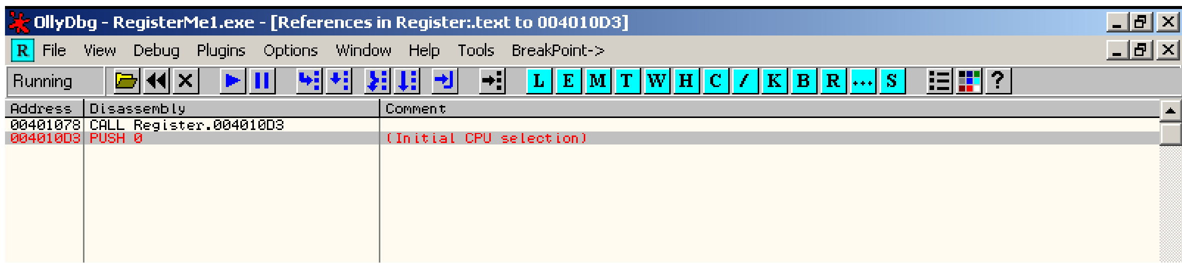


Vemos que Olly nos lleva al corazón del problema, donde podemos observar un conjunto de métodos separados. Comprobemos desde donde son llamados.

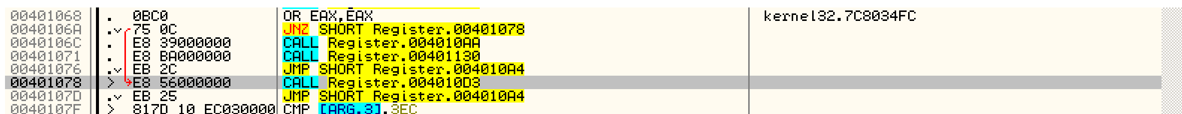
Nos situamos al principio de la función, hacemos clic con el botón derecho y seleccionamos “Find references to” -> “Selected command”.



Y Olly abre la ventana de referencias:

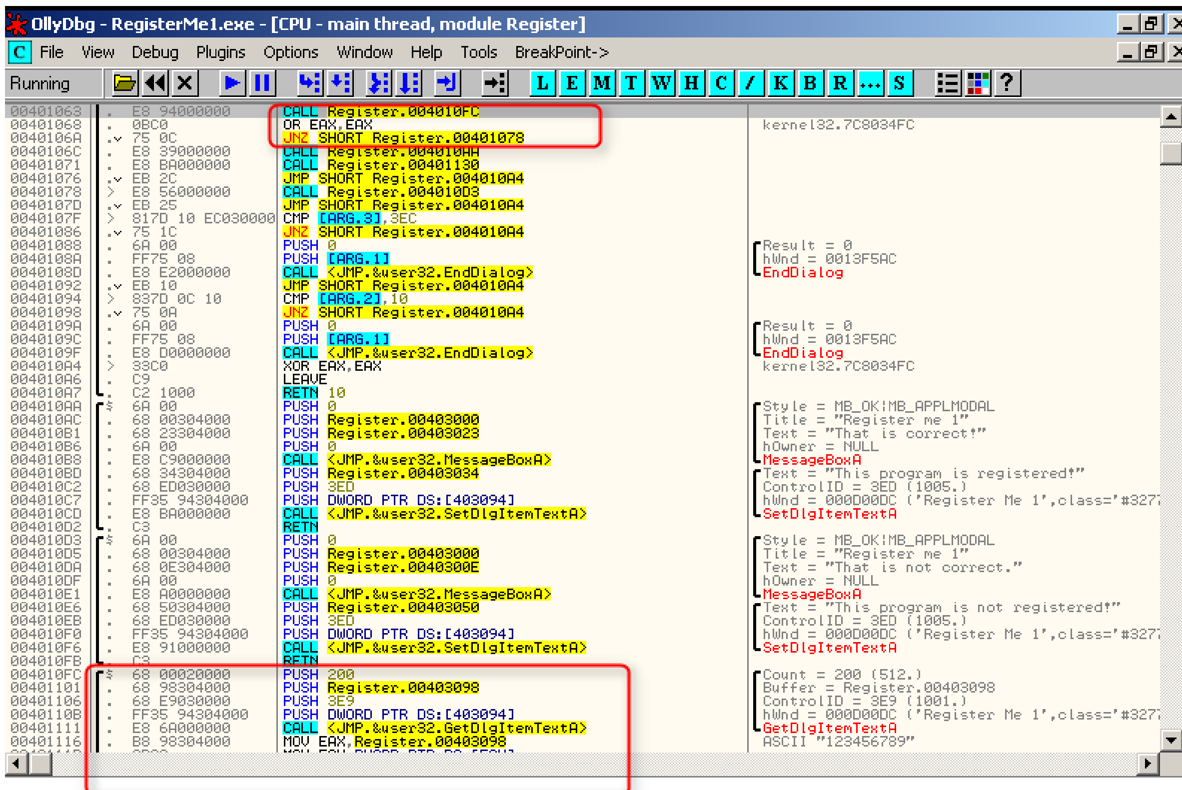


Vemos que hay un call a esa función. Hacemos doble clic y estudiemos su estructura.

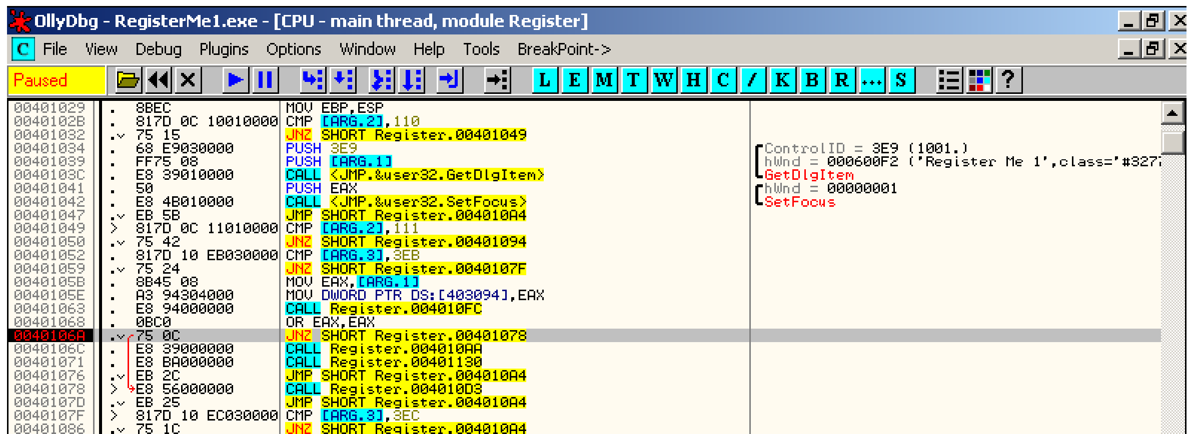


Podemos ver de forma inmediata que el “bad boy” es llamado por la dirección 401078 y que existe una instrucción de salto en 40106A que nos lleva a ese CALL en 401078.

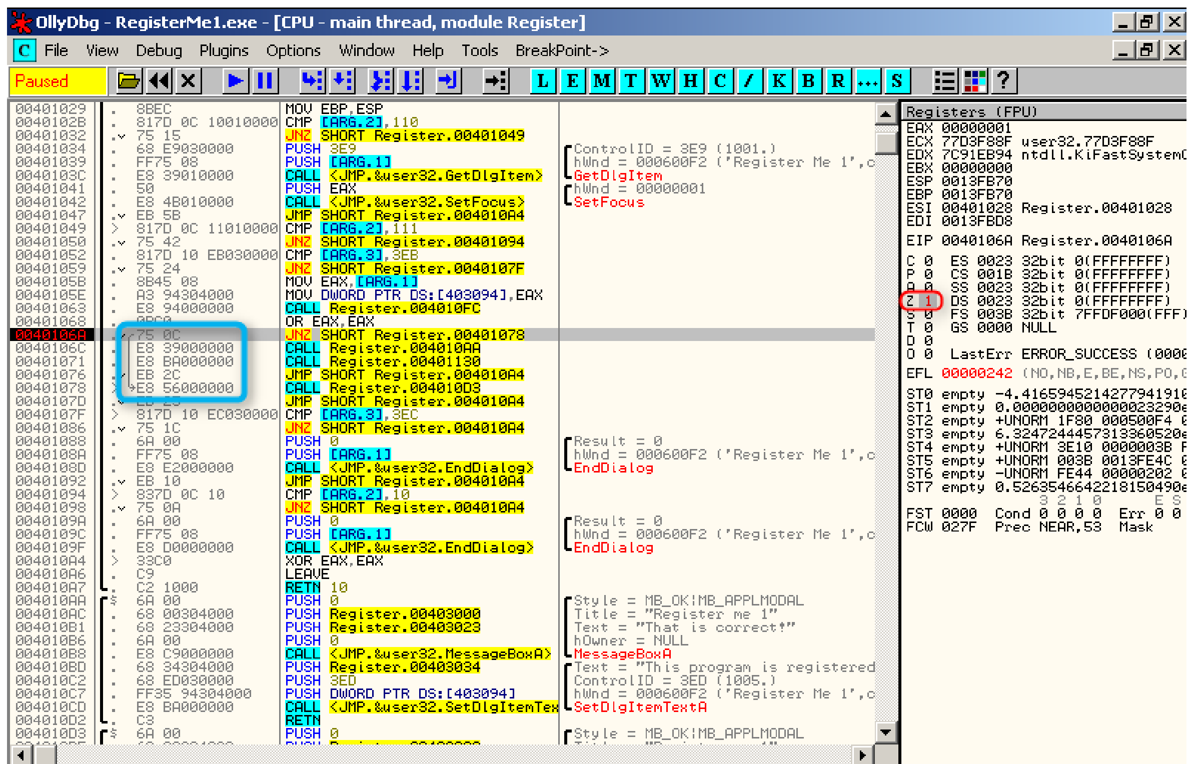
Si subimos unas líneas más arriba podemos ver el CALL que comprueba la rutina salto/comparar que hemos visto anteriormente. Podemos concluir entonces que el CALL en 401063 es el CALL principal que comprueba la rutina en 4010FC, responsable de comprobar si el programa está registrado o no. Después de regresar, el registro EAX comprueba si su valor es cero o no, y si no lo es, tomaremos el salto hacia el “bad boy”.



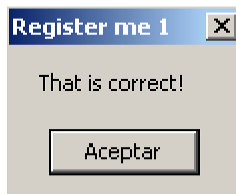
Comprobemos nuestra hipotesis y pongamos un Breakpoint en 40106A. Reiniciamos la aplicación. Introducimos un serial y nos detenemos en el Breakpoint despues del CALL que comprueba la rutina del registro.



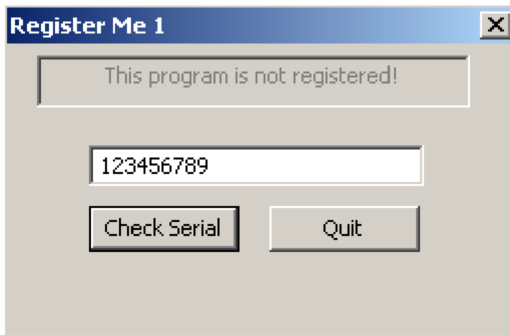
Ayudemos a Olly para que no tome el salto, haciendo doble clic en la bandera Z, y ejecutando el CALL que nos llevará al “good boy”.



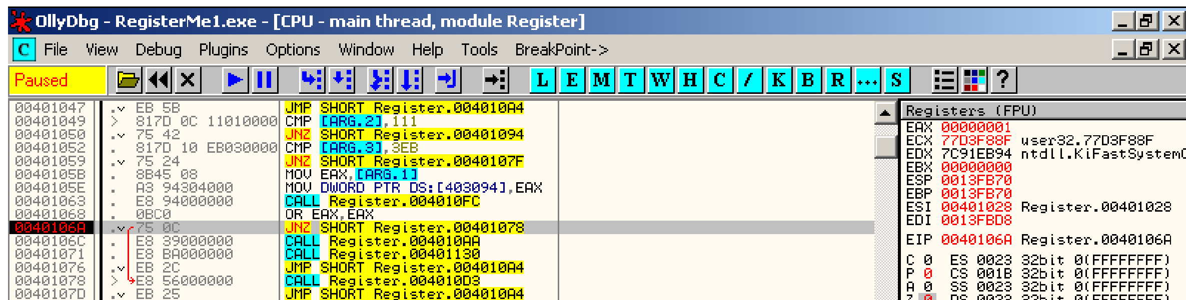
Pulsamos F9.



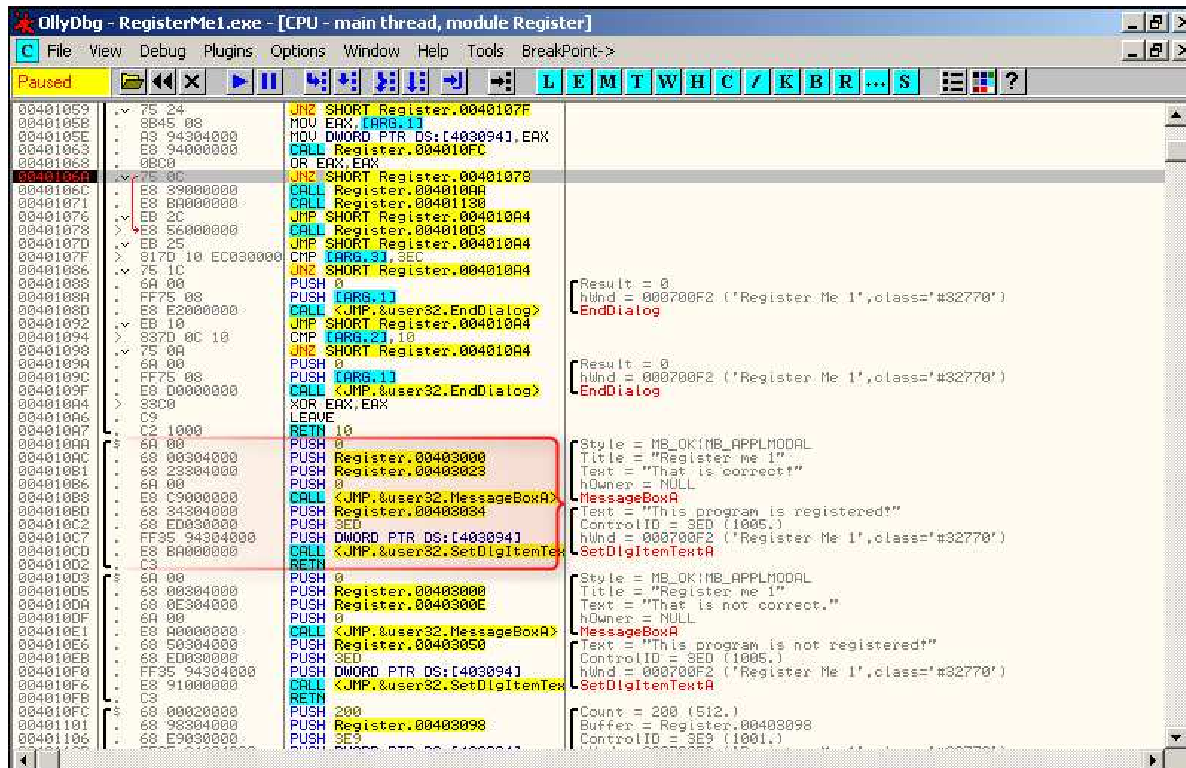
Hacemos clic en “Aceptar” y ….



Bueno, paraece que seguimos estando sin registrar!!! Tiene que haber algo que hemos omitido!!! Volvamos pues, reiniciar la aplicación, introducimos un serial y dejamos que Olly se detenga en el Breakpoint 40106A.



Si obligamos a Olly a no coger el salto hacia el “bad boy”, la aplicación seguirá ejecutandose en la dirección 40106C donde hará una llamada a la dirección 4010AA. Estudiando la rutina 4010AA podemos apreciar como se abrirá un cuadro de dialogo con el mensaje “That is correct” para seguidamente cambiar la etiqueta de la ventan principal a “This program is registered”.



Ahora, una vez que hayamos regresado de ese CALL, hay otro CALL en 401071.

```
0040106A  .v 75 0C        JNZ SHORT Register.00401078
0040106C  .  E8 39000000  CALL Register.004010AA
00401071  .  E8 BA000000  CALL Register.00401130
```

Este CALL a su vez llama al CALL en 401130. Echemos un vistazo a esta última rutina.

Lo primero que vemos es que llama a SetDigItemTextA pero con una cadena un tanto peculiar. Pulsemos F8 y vamos paso a paso. En 401130 se está llamando a la dirección 4010FC. Del analisis anterior sabemos que coincide con la rutina que comprueba el serial. Luego compara, con la instrucción OR, EAX consigo misma para ver si vale cero. Si no vale cero realizará todo un conjunto de movimientos que vienen a continuación:

```
00401130  $  E8 C7FFFFFF   CALL Register.004010FC
00401135  .  0BC0          OR EAX,EAX
00401137  .v 74 33         JE SHORT Register.0040116C
00401139  .  B9 1F000000   MOV ECX,1F
0040113E  .  BE 00000000   MOV ESI,0
00401143  .  33C0          XOR EAX,EAX
00401145  >  8A86 70304000 MOV AL,BYTE PTR DS:[ESI+40307
0040114B  .  83F0 2C       XOR EAX,2C
0040114E  .  8886 70304000 MOV BYTE PTR DS:[ESI+403070],
00401154  .  46            INC ESI                          Register.00401028
00401155  .^ E2 EE         LOOPD SHORT Register.00401145
00401157  .  68 70304000   PUSH Register.00403070           Text = "xDE_\x0C\\^CK^MA\x0CE_\x0CBCX\x0C^IKE_XI^IH\r"
0040115C  .  68 ED030000   PUSH 3ED                         ControlID = 3ED (1005.)
00401161  .  FF35 94304000 PUSH DWORD PTR DS:[403094]       hWnd = 000700F2 ('Register Me 1',class='#32770')
00401167  .  E8 20000000   CALL <JMP.&user32.SetDlgItemTe   SetDlgItemTextA
0040116C  >  C3            RETN
```

Resumiendo, después de parchear la aplicación para que saliera el “good boy”, se invocó otro CALL, y dentro de ese CALL se volvió hacer otra llamada para comprobar otra vez la rutina del serial, realizando de esta manera el mismo analisis sobre el resultado. Esto se conoce como “backup check”. Averiguemos lo que sucede si no pasamos este segundo “backup check” (Sabemos que no lo vamos a pasar porque solo parcheamos el salto).

Primero se carga 1F (31d) en el registro ECX. Después se carga ESI con 0, y EAX se compara a si mismo para ver si vale cero. Acontinuación entramos en un loop. La primera línea del loop mueve un byte de la dirección ESI+403070 al registro AL, y como sabemos que ESI es igual a cero, el valor coincide con la dirección en 403070. Veamos lo que hay en esa dirección. Para ello nos situamos en la ventana Dump y seleccionamos “Go To” -> “Expression” e introducimos 403070.

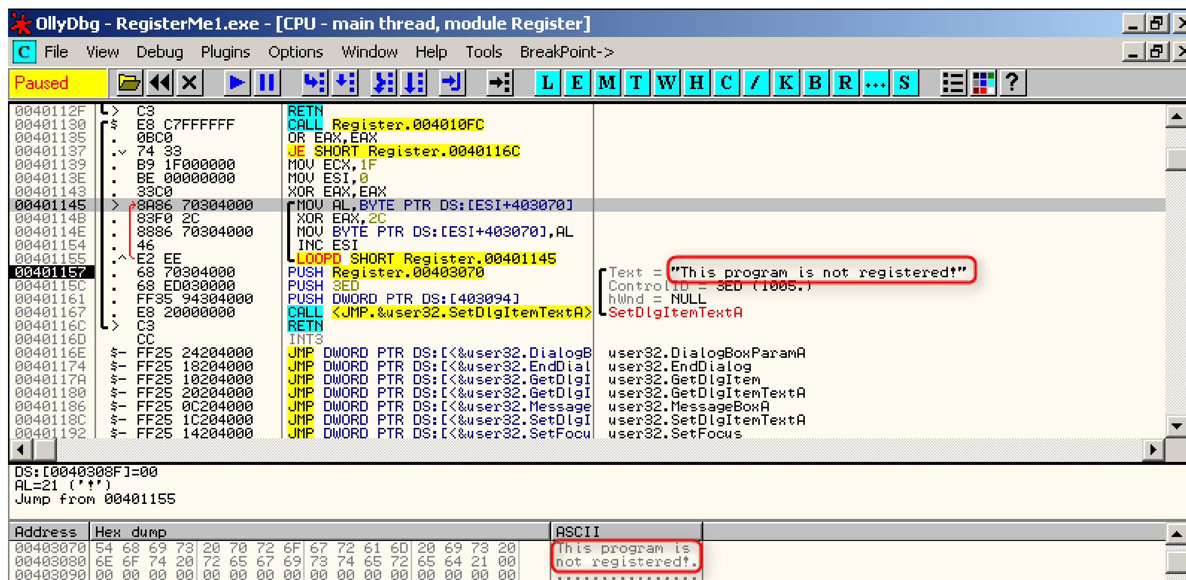
```
Address   Hex dump                                          ASCII
00403070  78 44 45 5F 0C 5C 5E 43 4B 5E 4D 41 0C 45 5F 0C  xDE_.\^CK^MA.E_.
00403080  42 43 58 0C 5E 49 4B 45 5F 58 49 5E 49 48 0D 00  BCX.^IKE_XI^IH..
00403090  00 00 40 00 F2 00 07 00 31 32 33 34 35 36 37 38  ..@.=.•.12345678
004030A0  39 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  9...............
004030B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403100  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403110  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403120  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403140  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403150  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403160  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403170  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Si nos fijamos bien, podemos ver que coincide con la cadena que se pasó como argumento a SetDigItemTextA. Así que carga el primer carácter de esta cadena en el registro AL.

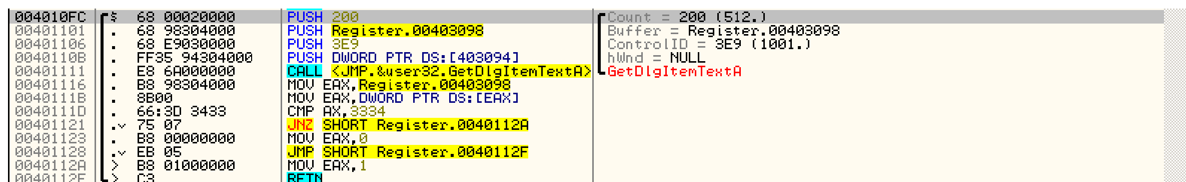
Siguiendo con el loop vemos que se compara ese caracter con 2C para guardar el resultado de vuelta en la misma dirección de la memoria: MOV BYTE PTR DS:[ESI+403070], AL.

Por último incrementamos el registro ESI y hacemos un LOOPD. LOOPD significa disminuir el registro ECX en uno y continuar con el loop hasta que ECX sea igual a cero. Podemos concluir con esto que el valor que ha sido cargado originariamente en ECX, 31d, coincide con la longitud del loop.

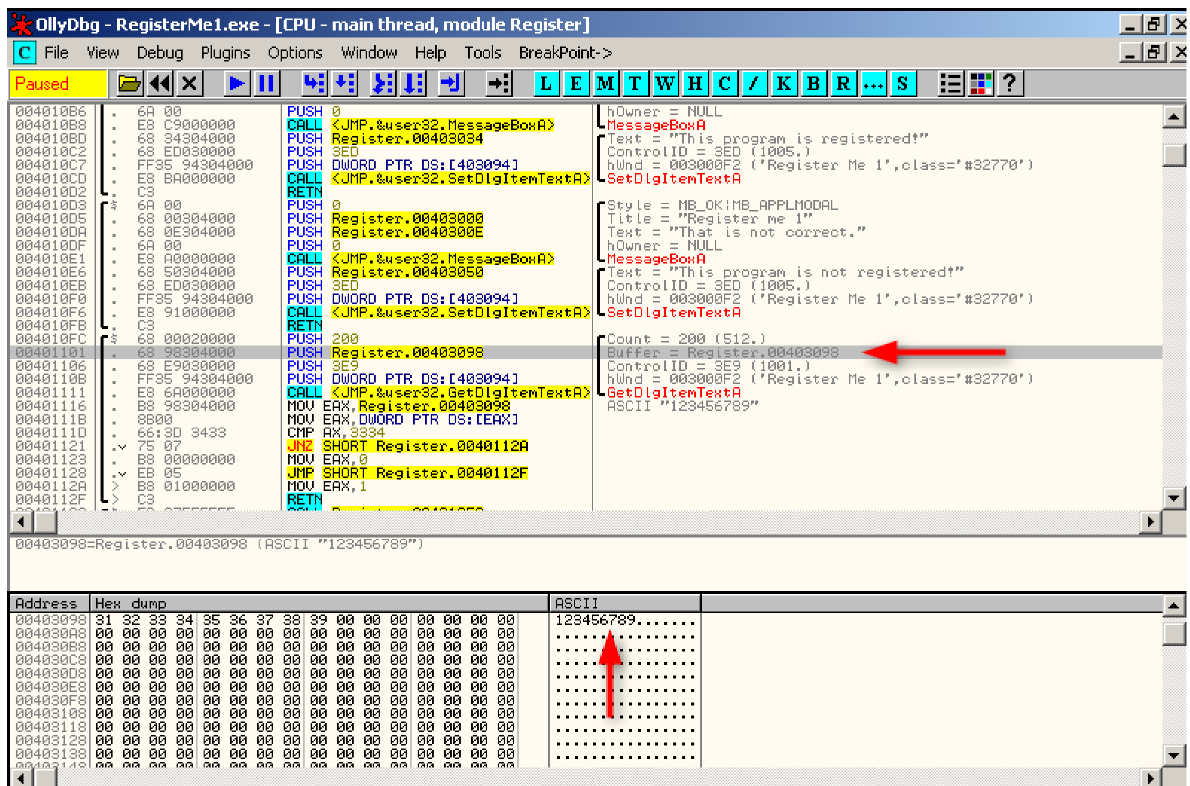
Después de completar le loop obtendremos el siguiente resultado:



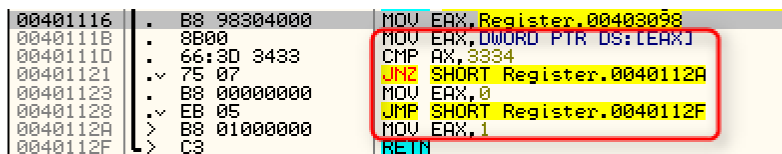
Volvamos a la rutina que verifica el serial:



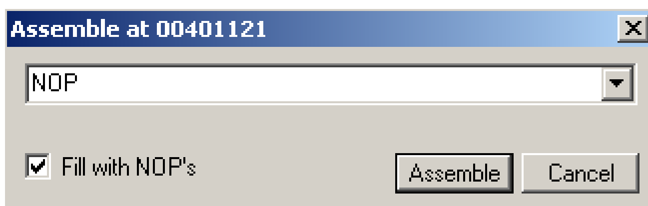
Al principio de la rutina tenemos un CALL a GetDigItemTextA. Hacemos clic con el botón derecho en la instrucción de la dirección 401101 (vemos que apunta al buffer donde está almacenado el serial) y seleccionamos "Follow in Dump". Después de pasar la instrucción GetDigItemTextA, podemos ver nuestro serial en el buffer:

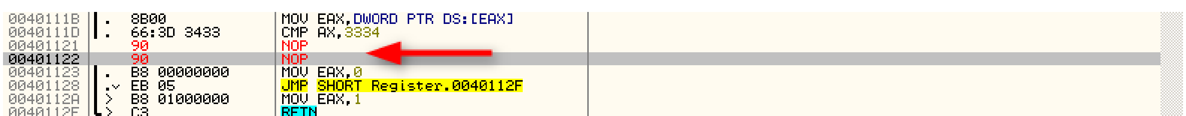


Una vez almacenado en el buffer, la dirección del principio del buffer es movido a EAX seguido del contenido de esa dirección. Esto basicamente moverá los cuatro primero bytes de nuestro serial a EAX. Estos bytes se compararán despues con 3334, y si no coincide, EAX se cargará con un uno (lo que es malo), en caso contrario, si coinciden EAX almazenará un cero (lo que es bueno).



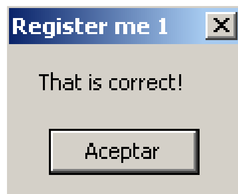
Podemos ver que la principal toma de decisión se hará en 401121. Esta línea determinará si EAX va tomar el valor cero o uno, justo antes del regreso. Así que lo que haremos es garantizar que el valor de EAX sea siempre cero:



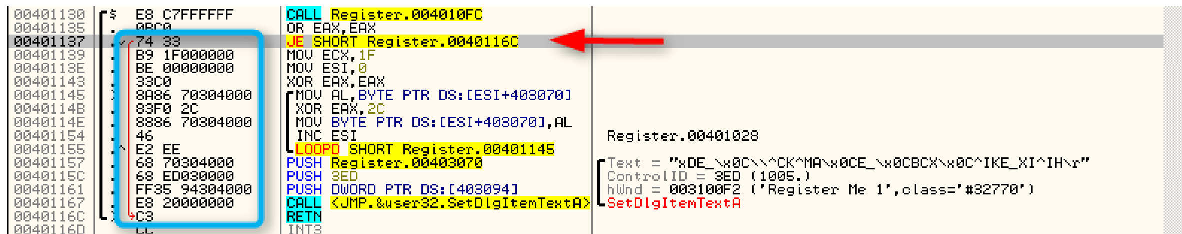


De esta forma el código siempre caerá en la instrucción MOV EAX, 0 para saltar a RETN.

Ejecutamos el programa:



Vemos que después del CALL que realiza la comprobación del serial, saltamos a nuestro “good boy”.



Y después de la segunda comprobación tambíen saltamos al “good boy”.

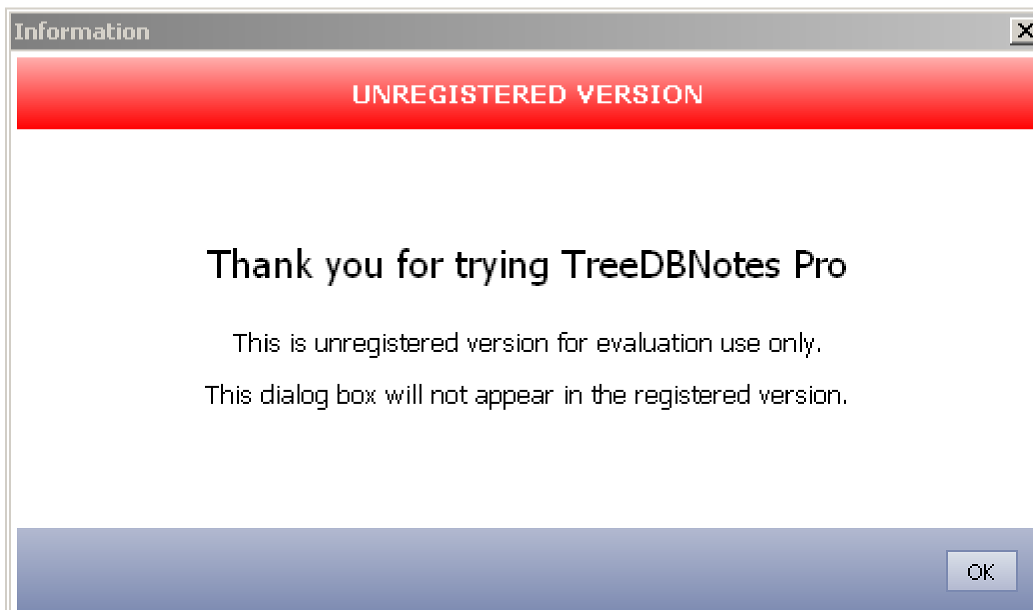
Conclusión: hemos encontrado un parche que registra nuestro serial independientemente de su valor.

7.10 Caso práctico 10: Crackeando un programa real

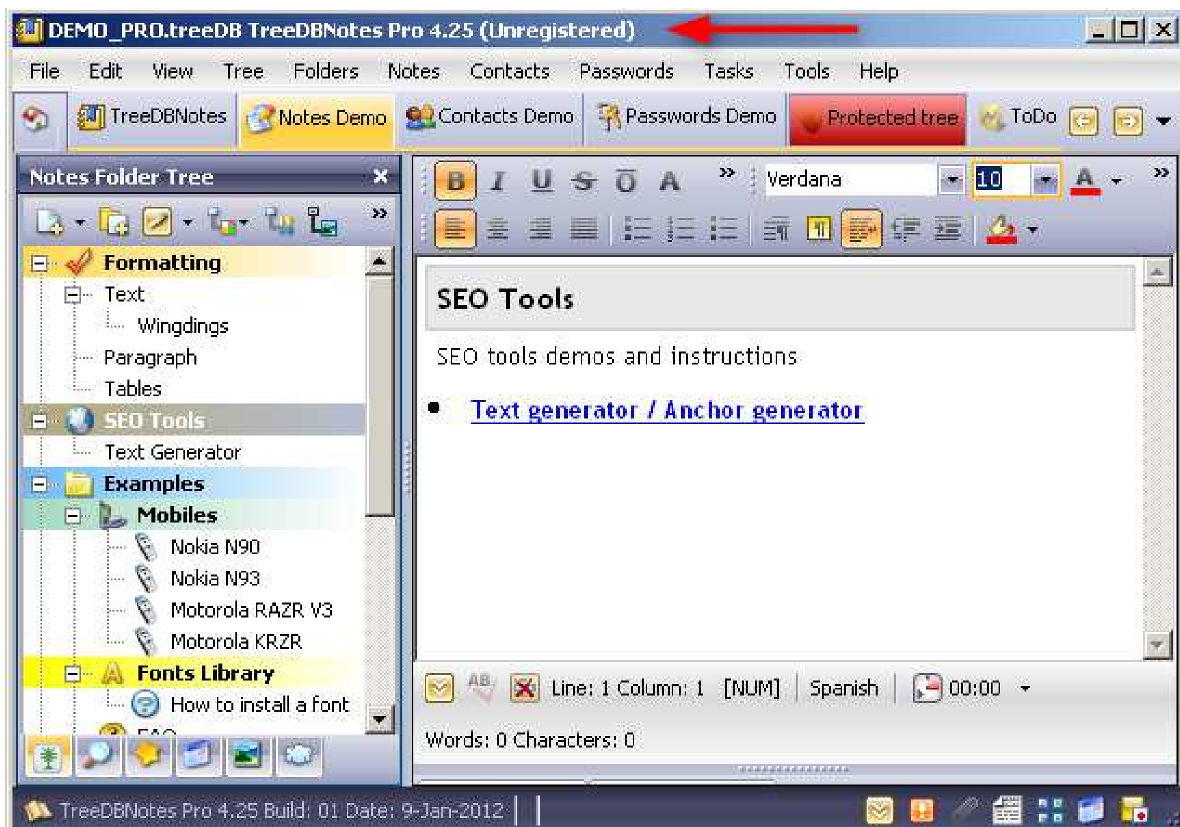
***Nota**: el trabajo y esfuerzo que un desarrollador de software pone en su proyecto bien merece la pena ser compensado de alguna manera. Este ejercicio no pretende ser un llamamiento a la piratería, pero sí un desafío a la inteligencia humana. Si realmente consideramos que debemos poseer una aplicación para su uso y disfrute, seamos honestos con nosotros mismos y demos al programador la remuneración que se merece.*

Instalemos la aplicación en nuestro disco duro. Una vez finalizada, nos aparece la siguiente ventana:

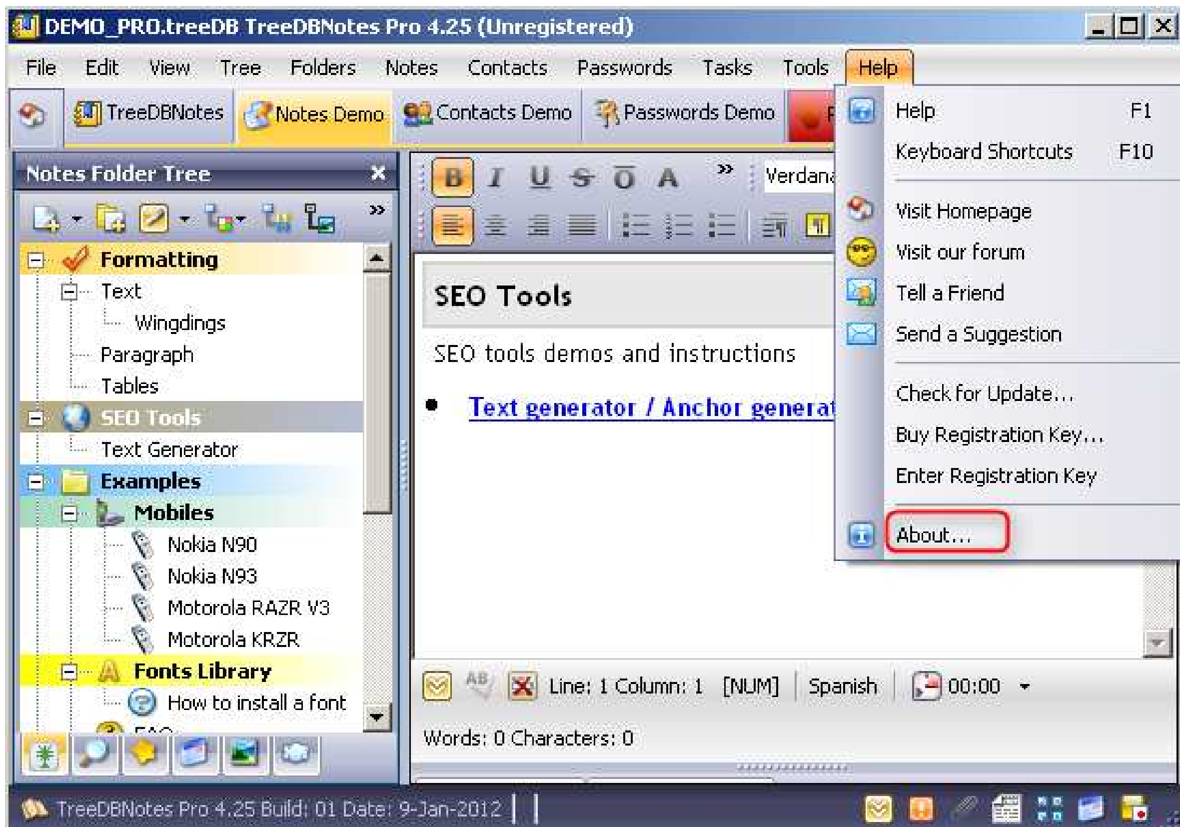
Dejamos la pestaña puesta en “Run TreeDBNotes” para ver a que nos enfrentamos y hacemos clic en “Finish”.

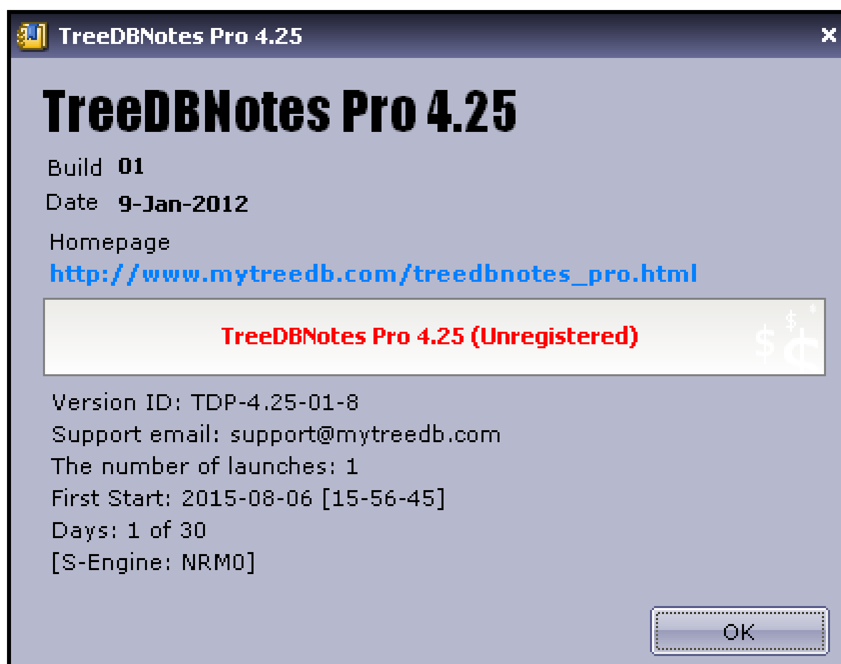


Vemos algunas cadenas de texto de interes como pueden ser: “unregistered”, “evaluation”,”registered”, etc. Hacemos clic en “OK” y se abre la ventana principal:

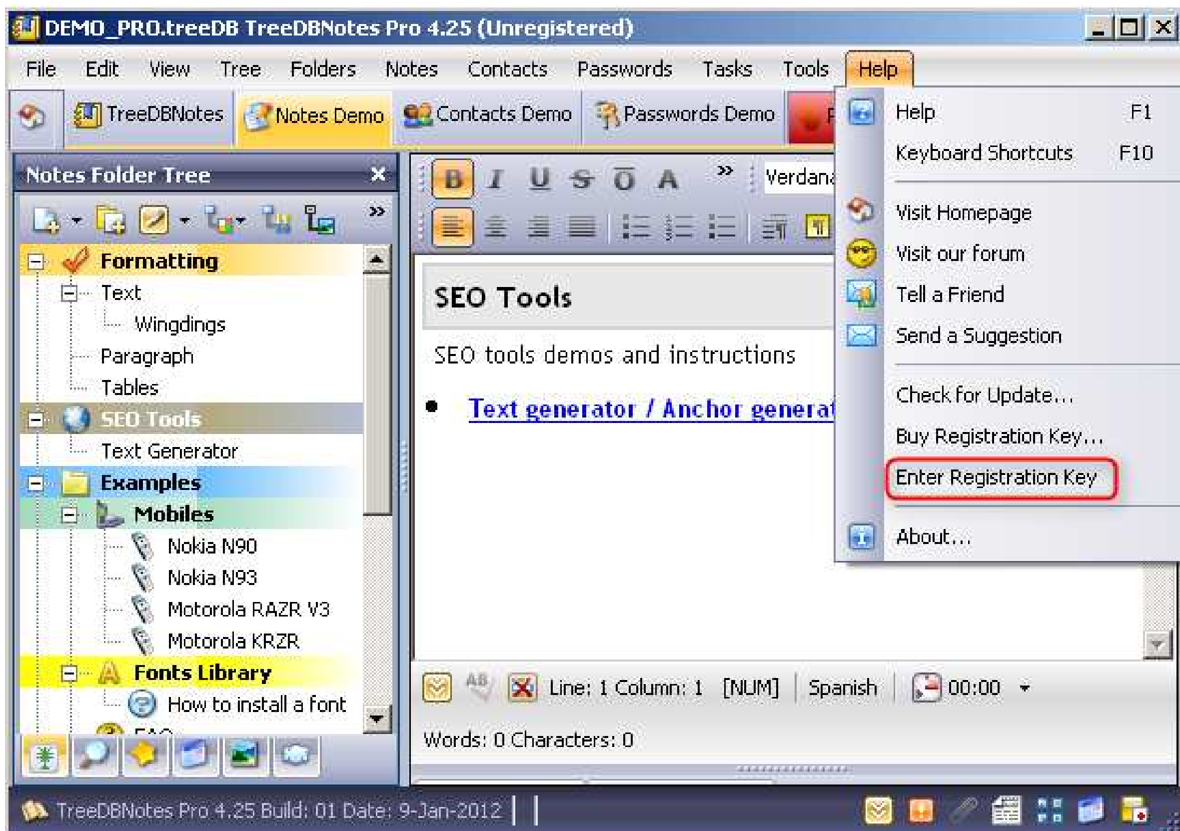


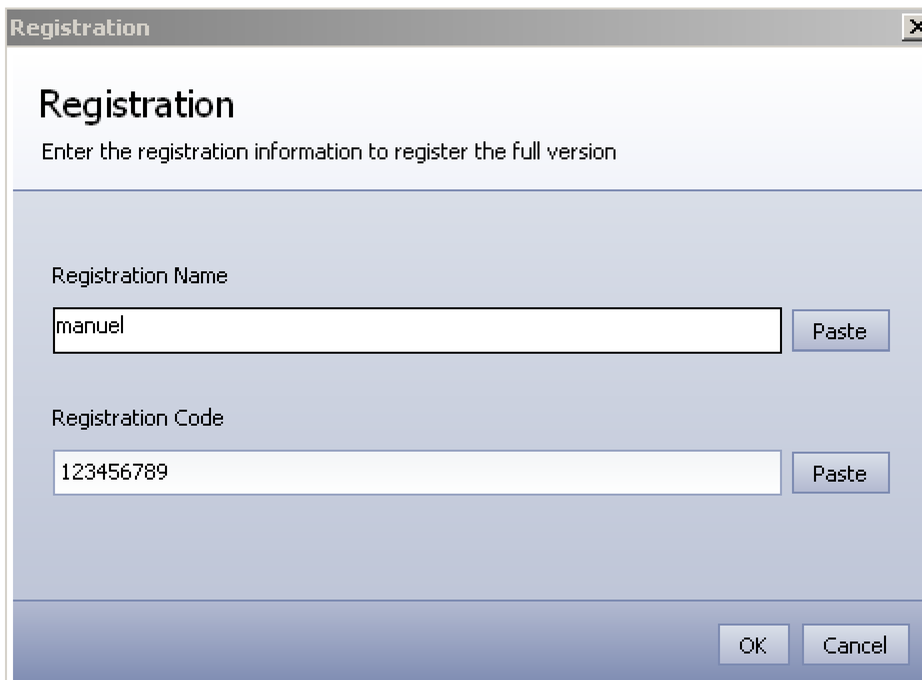
Vemos que pone “Unregistered” en la barra superior. Otro sitio donde poder recabar información acerca de la aplicación es pulsando el botón “Help” -> “About”:



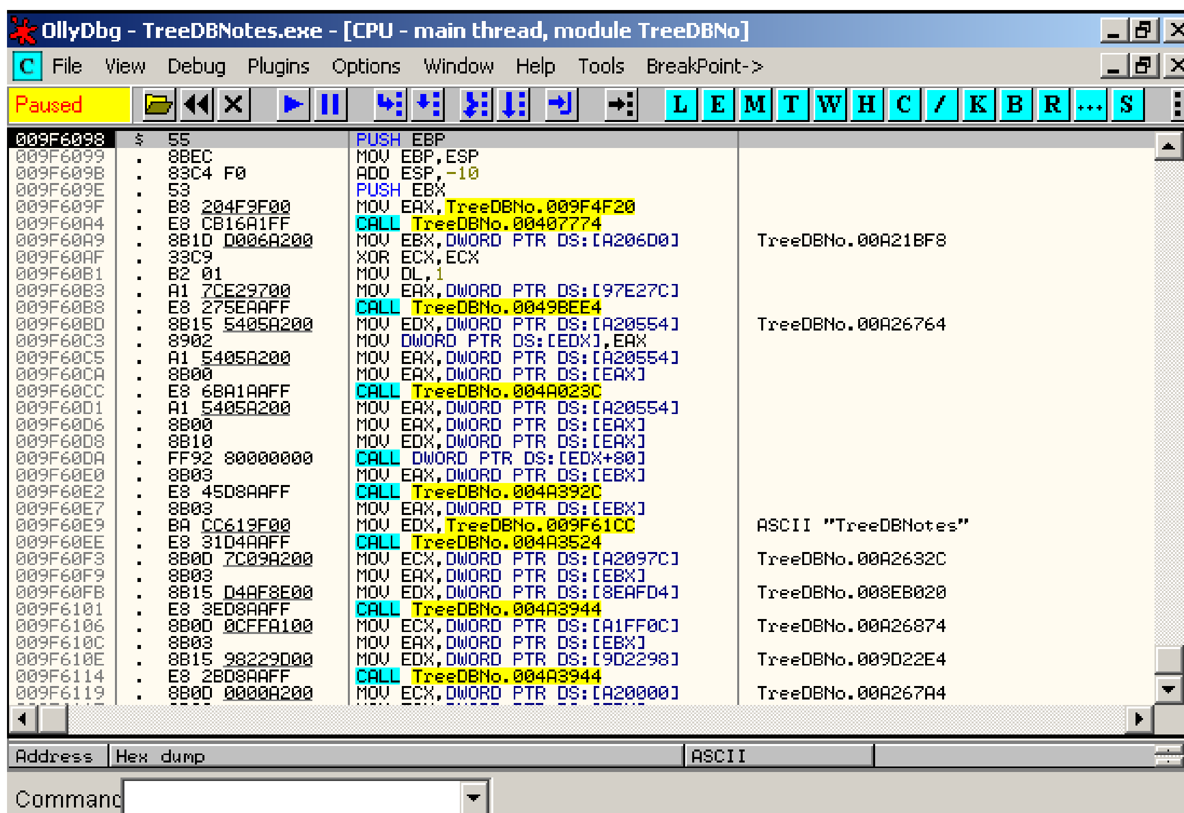


Vemos que vuelve a poner “Unregistered”. Para el caso en que Olly no detecte ningúna cadena de texto podemos ir a “Enter Registration Key” e introducir valores aleatorios:


DEMO_PRO.treeDB TreeDBNotes Pro 4.25 (Unregistered)
File Edit View Tree Folders Notes Contacts Passwords Tasks Tools Help
TreeDBNotes Notes Demo Contacts Demo Passwords Demo
Notes Folder Tree
Formatting
Text
Wingdings
Paragraph
Tables
SEO Tools
Text Generator
Examples
Mobiles
Nokia N90
Nokia N93
Motorola RAZR V3
Motorola KRZR
Fonts Library
How to install a font
SEO Tools
SEO tools demos and instructions
Text generator / Anchor generat
Help F1
Keyboard Shortcuts F10
Visit Homepage
Visit our forum
Tell a Friend
Send a Suggestion
Check for Update...
Buy Registration Key...
Enter Registration Key
About...
Line: 1 Column: 1 [NUM] Spanish 00:00
Words: 0 Characters: 0
TreeDBNotes Pro 4.25 Build: 01 Date: 9-Jan-2012

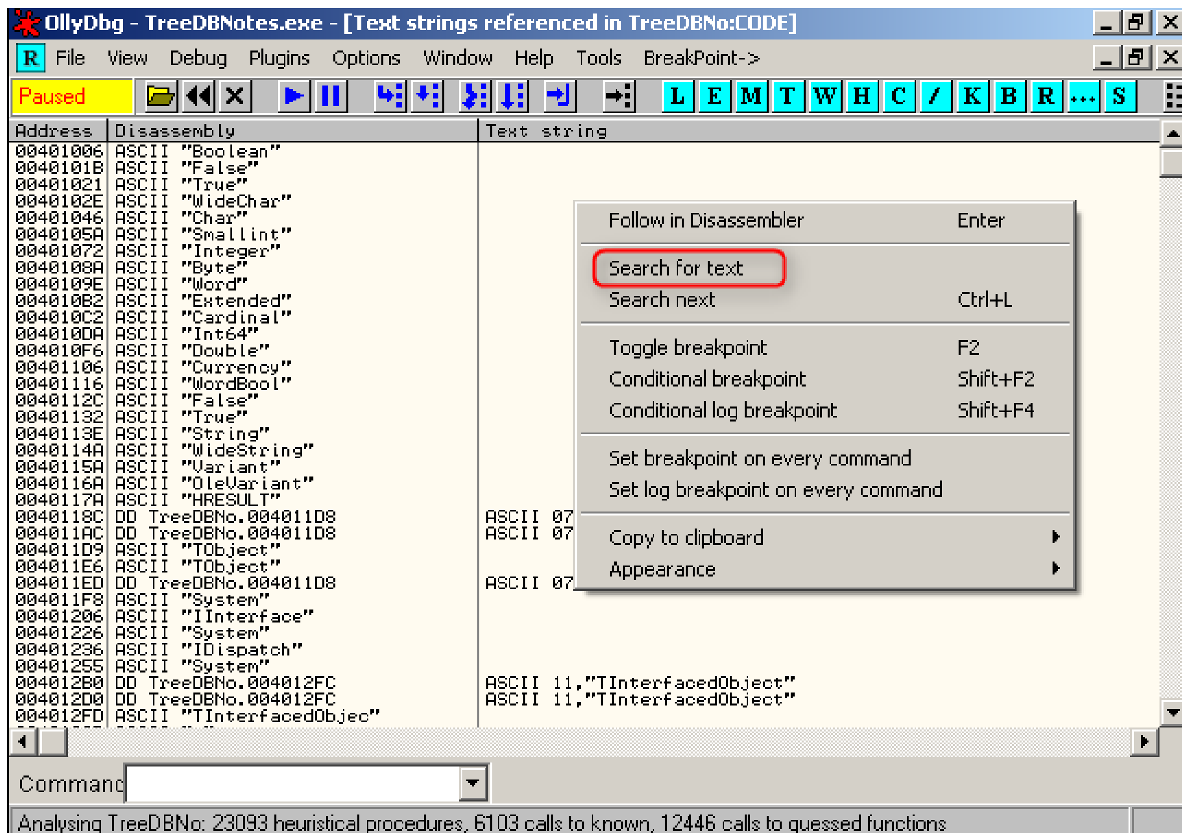

Registration
Registration
Enter the registration information to register the full version
Registration Name
manuel
Paste
Registration Code
123456789
Paste
OK
Cancel

Hasta aquí ya tenemos una idea más o menos clara de lo que tenemos a nuestra disposición. Cargemos pues la aplicación en Olly:

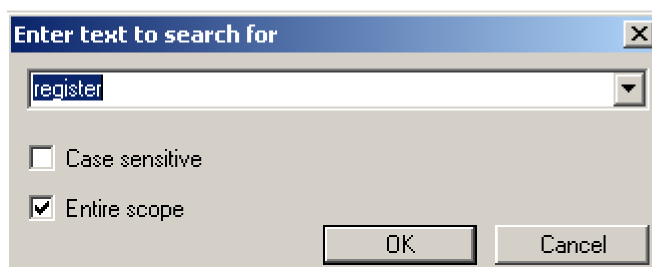


Lo primero que notamos es que hay muchas instrucciónes CALL y ningúna API de Windows. Esto es una buena señal de que el programa podría estar escrito en Delphi.

Hacemos clic con el botón derecho en la ventana de desensamblaje y seleccionamos “Search for” -> “All referenced text strings”.Dentro de la ventana (R) hacemos clic con el botón derecho y seleccionamos “Search for text”.

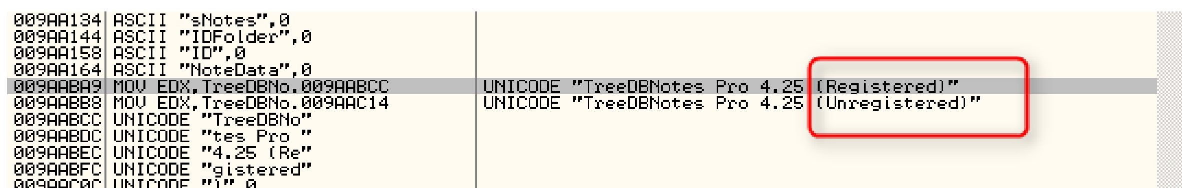


Hemos visto como las palabras “registration” y “registered” aparecieron unas cuantas veces. Hagamos pues una busqueda que contenga los caraceteres “regist”, deshabilitamos la casilla de “Case sensitive” y marcamso la de “Entire Scope”.



La primera cadena de texto que encuentra no parece decir gran cosa, asi que pulsamso Ctrl+Alt+L para buscar la siguiente:

Las próximas cadenas tampoco son prometedoras, por lo que seguiremos pulsando Ctrl+Alt+L hasta llegar a las siguientes líneas:



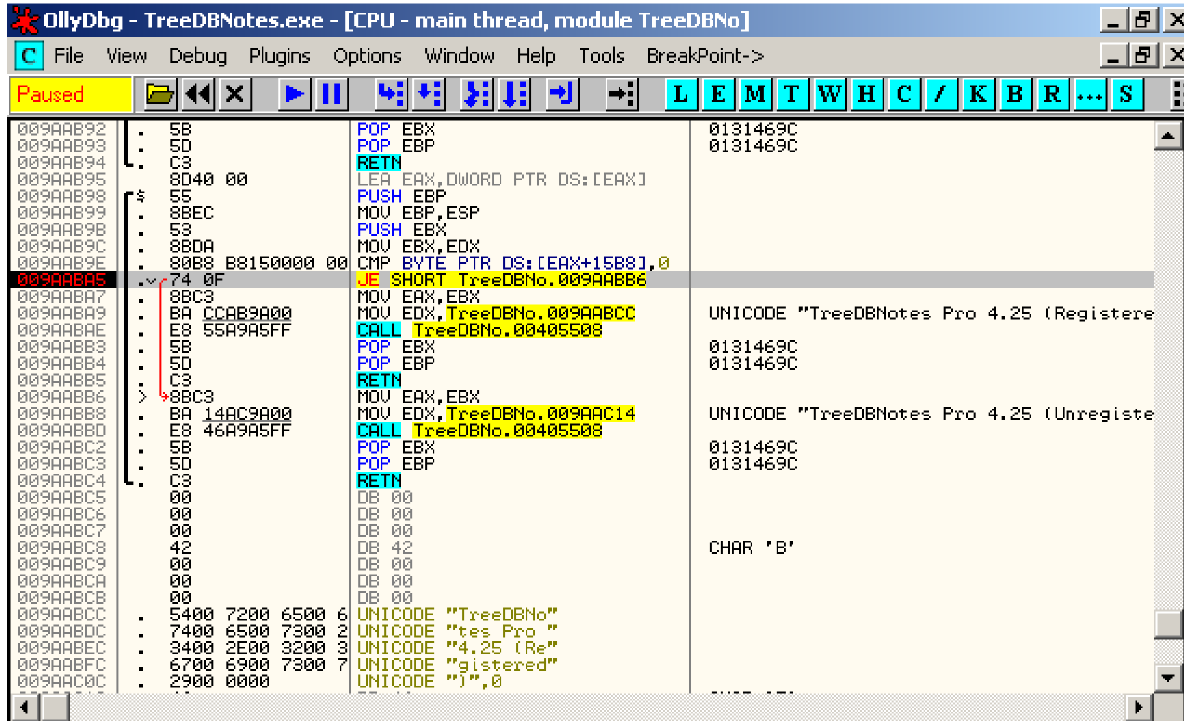
Hacemos doble clic sobre (Registered) lo que nos lleva al código siguiente:

```
OllyDbg - TreeDBNotes.exe - [CPU - main thread, module TreeDBNo]
C File View Debug Plugins Options Window Help Tools BreakPoint->
Paused                                    L E M T W H C / K B R ... S

009AAB92  .  5B              POP EBX                                  kernel32.7C816D4F
009AAB93  .  5D              POP EBP                                  kernel32.7C816D4F
009AAB94  .  C3              RETN
009AAB95     8D40 00         LEA EAX,DWORD PTR DS:[EAX]
009AAB98  $  55              PUSH EBP
009AAB99  .  8BEC            MOV EBP,ESP
009AAB9B  .  53              PUSH EBX
009AAB9C  .  8BDA            MOV EBX,EDX                              ntdll.KiFastSystemCallRet
009AAB9E  .  80B8 B8150000 00 CMP BYTE PTR DS:[EAX+15B8],0
009AABA5  .v 74 0F           JE SHORT TreeDBNo.009AABB6
009AABA7  .  8BC3            MOV EAX,EBX
009AABA9  .  BA CCAB9A00     MOV EDX,TreeDBNo.009AABCC                UNICODE "TreeDBNotes Pro 4.25 (Regis
009AABAE  .  E8 55A9A5FF     CALL TreeDBNo.00405508
009AABB3  .  5B              POP EBX                                  kernel32.7C816D4F
009AABB4  .  5D              POP EBP                                  kernel32.7C816D4F
009AABB5  .  C3              RETN
009AABB6  >  8BC3            MOV EAX,EBX
009AABB8  .  BA 14AC9A00     MOV EDX,TreeDBNo.009AAC14                UNICODE "TreeDBNotes Pro 4.25 (Unreg
009AABBD  .  E8 46A9A5FF     CALL TreeDBNo.00405508
009AABC2  .  5B              POP EBX                                  kernel32.7C816D4F
009AABC3  .  5D              POP EBP                                  kernel32.7C816D4F
009AABC4  .  C3              RETN
009AABC5     00              DB 00
009AABC6     00              DB 00
009AABC7     00              DB 00
009AABC8     42              DB 42                                    CHAR 'B'
009AABC9     00              DB 00
009AABCA     00              DB 00
009AABCB     00              DB 00
009AABCC  .  5400 7200 6500 6 UNICODE "TreeDBNo"
009AABDC  .  7400 6500 7300 2 UNICODE "tes Pro "
009AABEC  .  3400 2E00 3200 3 UNICODE "4.25 (Re"
009AABFC  .  6700 6900 7300 7 UNICODE "gistered"
009AAC0C  .  2900 0000       UNICODE ")",0
```

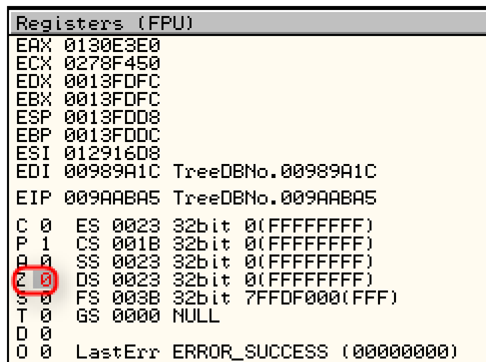
En primer lugar podemos ver la dirección en la que se utiliza la cadena de texto (9AABA9) y también podemos ver donde se almacena en la memoria (9AABBC). En segundo lugar vemos que ambas cadenas de texto ("Registered" y "Unregistered") pertenecen al mismo método (delimitado por la línea negra al lado de los opcodes). También podemos apreciar un salto condicional en la dirección 9AABA5.

```
OllyDbg - TreeDBNotes.exe - [CPU - main thread, module TreeDBNo]
C File View Debug Plugins Options Window Help Tools BreakPoint->
Paused                                    L E M T W H C / K B R ... S

009AAB92  .  5B              POP EBX                                  kernel32.7C816D4F
009AAB93  .  5D              POP EBP                                  kernel32.7C816D4F
009AAB94  .  C3              RETN
009AAB95     8D40 00         LEA EAX,DWORD PTR DS:[EAX]
009AAB98  $  55              PUSH EBP
009AAB99  .  8BEC            MOV EBP,ESP
009AAB9B  .  53              PUSH EBX
009AAB9C  .  8BDA            MOV EBX,EDX                              ntdll.KiFastSystemCallRet
009AAB9E  .  80B8 B8150000 00 CMP BYTE PTR DS:[EAX+15B8],0
009AABA5  .v 74 0F           JE SHORT TreeDBNo.009AABB6
009AABA7  .  8BC3            MOV EAX,EBX
009AABA9  .  BA CCAB9A00     MOV EDX,TreeDBNo.009AABCC                UNICODE "TreeDBNotes Pro 4.25 (Registered
009AABAE  .  E8 55A9A5FF     CALL TreeDBNo.00405508
009AABB3  .  5B              POP EBX                                  kernel32.7C816D4F
009AABB4  .  5D              POP EBP                                  kernel32.7C816D4F
009AABB5  .  C3              RETN
009AABB6  >  8BC3            MOV EAX,EBX
009AABB8  .  BA 14AC9A00     MOV EDX,TreeDBNo.009AAC14                UNICODE "TreeDBNotes Pro 4.25 (Unregister
009AABBD  .  E8 46A9A5FF     CALL TreeDBNo.00405508
009AABC2  .  5B              POP EBX                                  kernel32.7C816D4F
009AABC3  .  5D              POP EBP                                  kernel32.7C816D4F
009AABC4  .  C3              RETN
009AABC5     00              DB 00
009AABC6     00              DB 00
009AABC7     00              DB 00
009AABC8     42              DB 42                                    CHAR 'B'
009AABC9     00              DB 00
009AABCA     00              DB 00
009AABCB     00              DB 00
009AABCC  .  5400 7200 6500 6 UNICODE "TreeDBNo"
009AABDC  .  7400 6500 7300 2 UNICODE "tes Pro "
009AABEC  .  3400 2E00 3200 3 UNICODE "4.25 (Re"
009AABFC  .  6700 6900 7300 7 UNICODE "gistered"
009AAC0C  .  2900 0000       UNICODE ")",0
```

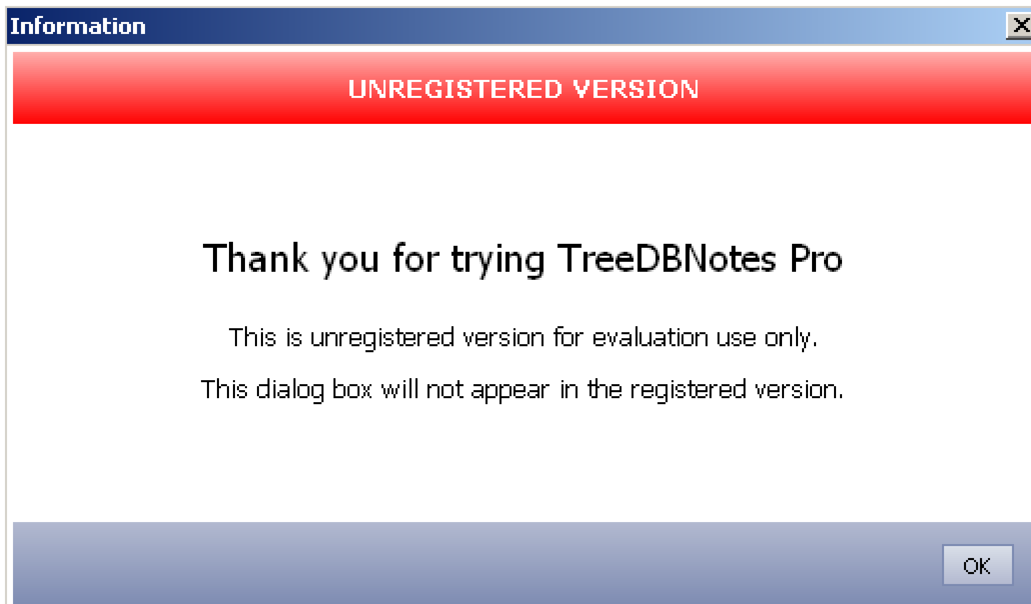
Si nos situamos encima de la instrucción JE podemos ver que si el resultado de la comparación anterior es igual entonces saltaremos a la versión “Unregistered” del programa. Pongamos un Breakpoint en esa dirección y pulsemos F9:



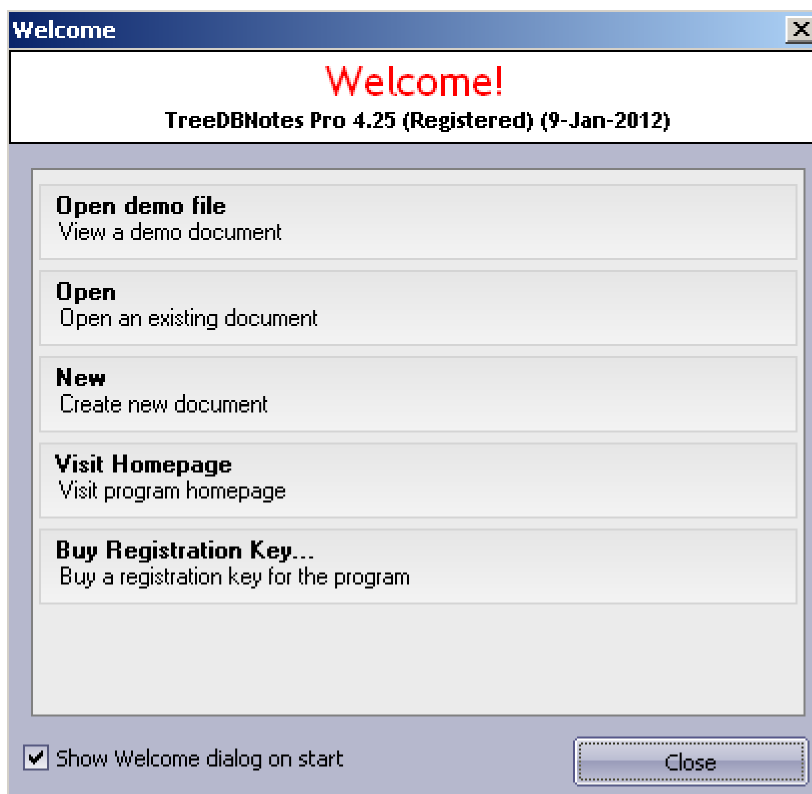
Olly se detiene en el Breakpoint y cogerá el salto hacia el “bad boy”. Para evitarlo cambiamos el valor de la bandera Z:



Volvemos a ejecutar el programa y Olly se detiene en nuestro Breakpoint por lo que volveremos a cambiar el valor de la bandera Z. Este proceso habrá que repetirlo hasta que finalmente aparezca la ventana siguiente:



Vemos que parcheando esta primera comprobación no nos ha llevado a ningún sitio. Sin embargo si hacemos clic en OK, y volvemos a cambiar el valor de la bandera Z para volver a ejecutar el programa, el "Unregistered" de la parte superior de la ventana ha desaparecido.



Ahora sí sabemos que vamos por el buen camino. Subamos pues un nivel en nuestra investigación e analizemos el código con más detalle. Reiniciamos la aplicación para detenernos en nuestro Breakpoint:

```
009AAB98  $  55              PUSH EBP
009AAB99  .  8BEC            MOV EBP,ESP
009AAB9B  .  53              PUSH EBX
009AAB9C  .  8BDA            MOV EBX,EDX
009AAB9E  .  80B8 B8150000 00 CMP BYTE PTR DS:[EAX+15B8],0
009AABA5  .v 74 0F           JE SHORT TreeDBNo.009AABB6
009AABA7  .  8BC3            MOV EAX,EBX
009AABA9  .  BA CCAB9A00     MOV EDX,TreeDBNo.009AABCC          UNICODE "TreeDBNotes Pro 4.25 (Registere
009AABAE  .  E8 55A9A5FF     CALL TreeDBNo.00405508
009AABB3  .  5B              POP EBX                             0131469C
009AABB4  .  5D              POP EBP                             0131469C
009AABB5  .  C3              RETN
009AABB6  >  8BC3            MOV EAX,EBX
009AABB8  .  BA 14AC9A00     MOV EDX,TreeDBNo.009AAC14          UNICODE "TreeDBNotes Pro 4.25 (Unregiste
009AABBD  .  E8 46A9A5FF     CALL TreeDBNo.00405508
009AABC2  .  5B              POP EBX                             0131469C
009AABC3  .  5D              POP EBP                             0131469C
009AABC4  .  C3              RETN
```

No hay ninguna instrucción CALL delante de JE, pero sí un CMP en la dirección 9AAB9E:

CMP BYTE PTR DS:[EAX+15B8],0

Dependiendo del resultado de esta comparación estaremos registrados o no. EAX+15B8 es una dirección en memoria, que al comenzar por DS nos indica que se trata de una variable global. Haciendo clic en la instrucción CMP podemos ver el valor de EAX+15B8:

```
DS:[0130F998]=00
```

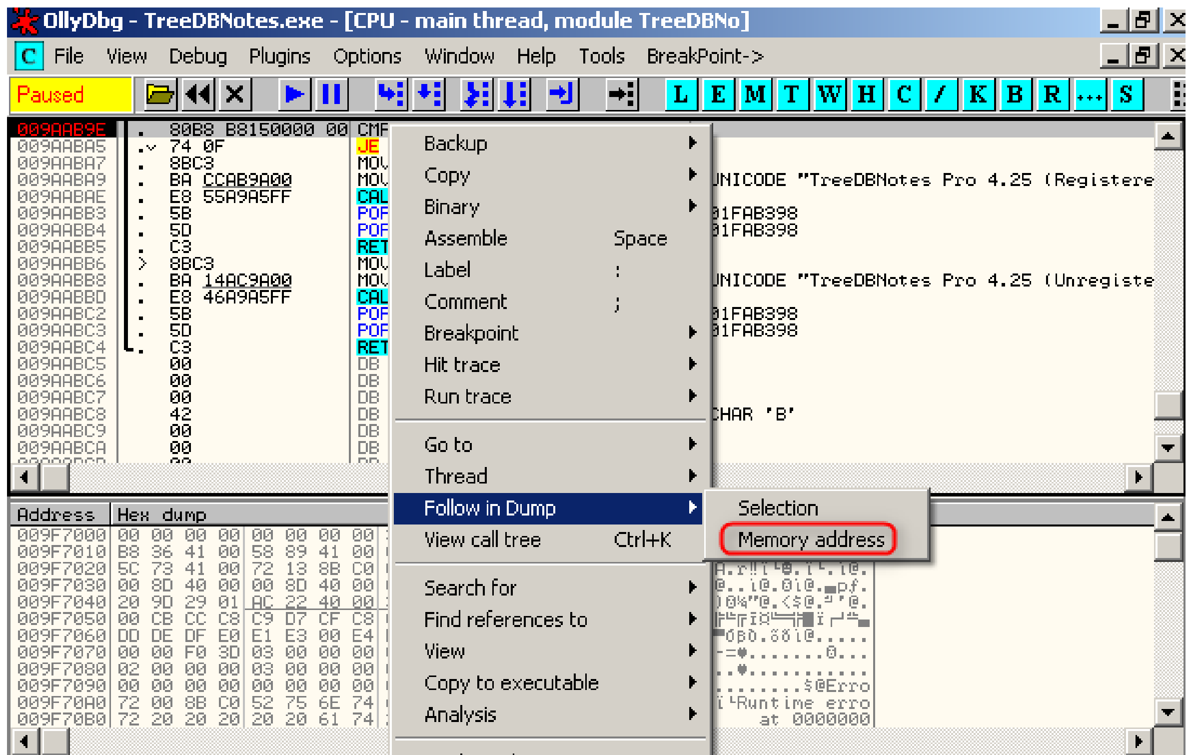
Hacemos clic con le botón derecho y seleccionamos “Follow address in Dump”.

```
Address   Hex dump
0130F998  00 00 00 00 FF FF FF FF 00 00 00 00 00 00 00 00
0130F9A8  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0130F9B8  00 00 00 00 00 00 00 00 01 00 00 00 01 00 00 00
0130F9C8  0A 00 00 00 00 00 00 00 9C 46 31 01 00 00 00 00
0130F9D8  00 00 00 00 84 EC 78 02 00 00 00 00 00 00 00 00
0130F9E8  00 00 00 00 00 00 00 00 08 F5 FC 01 20 F8 FC 01
0130F9F8  90 FA FC 01 0C D7 F9 01 3C 87 F8 01 00 00 00 00
0130FA08  26 00 00 00 28 AB 88 00 00 00 00 00 00 00 00 00
0130FA18  08 00 00 FF 00 00 00 00 04 9B FE 01 08 00 00 00
0130FA28  00 00 00 00 12 00 00 00 01 00 00 00 01 00 00 00
0130FA38  3D 00 00 00 1E 01 00 00 34 95 50 00 E0 E3 30 01
```

La primera dirección que va a ser comprobada para averiguar si estamos registrados o no es el primer 00 correspondiente a la dirección 130F998. Esto significa que si el contenido de este espacio en memoria es distinto a cero, la rutina asumirá que estamos registrados. Esto también significa que probablemente haya más rutinas en la aplicación que comprobarán ese espacio en memoria. Por eso en la ventana principal pone que estamos registrados mientras que un lugar diferente de la aplicación sabe que no lo estamos.

Lo primero que haremos es asignarle a esta rutina un valor distinto a cero para asegurarnos el “correcto” funcionamiento en esta parte de la memoria.

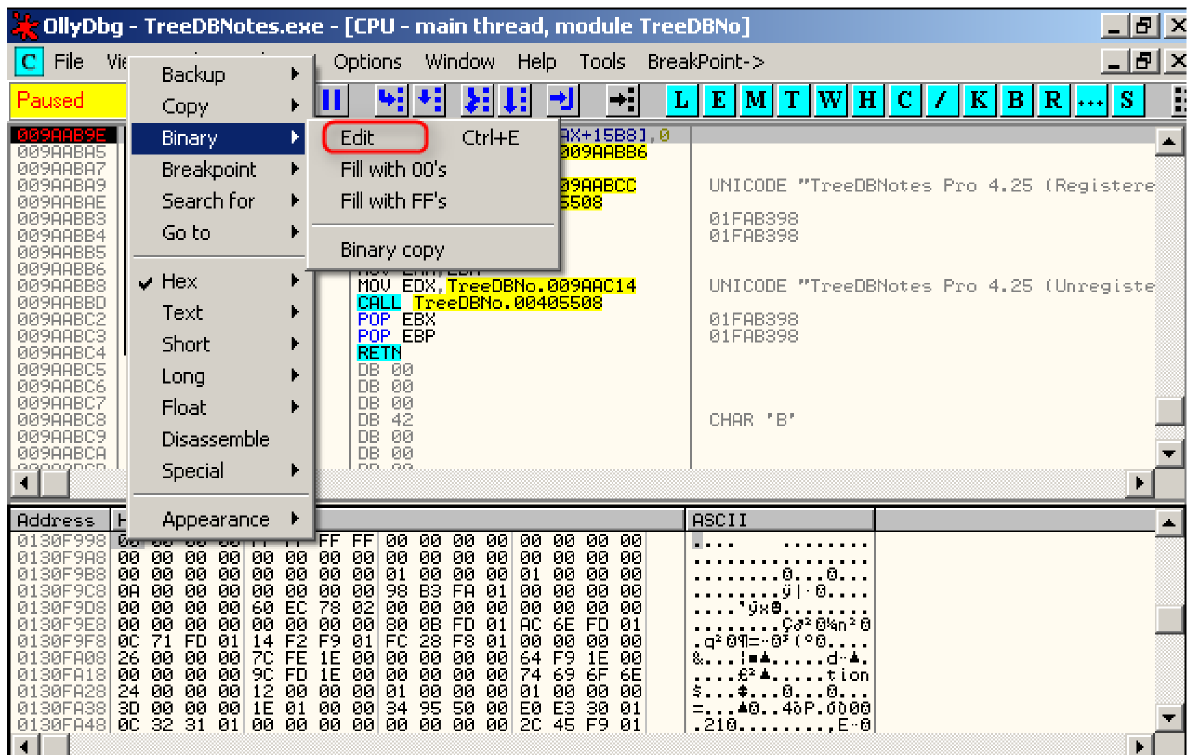
Ponemos un Breakpoint en la instrucción CMP e eliminamos el Breakpoint en JE. Reiniciamos la aplicación e Olly se detiene en nuestro nuevo Breakpoint. Hacemos clic derecho sobre la instrucción y seleccionamos “Follow in Dump” -> “Memory address”.



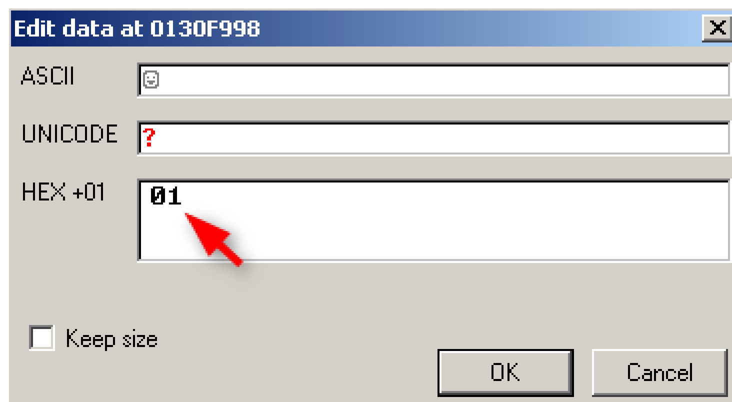
Lo primero que vemos, es que la dirección en memoria que almacena la bandera que determina si estamos registrados o no, es diferente (130F998)

```
Address  Hex dump                                         ASCII
0130F998 00 00 00 00 FF FF FF FF 00 00 00 00 00 00 00 00 ....    ........
0130F9A8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0130F9B8 00 00 00 00 00 00 00 00 01 00 00 00 01 00 00 00 ........0...0...
0130F9C8 0A 00 00 00 00 00 00 00 98 B3 FA 01 00 00 00 00 ........ÿ|·0....
0130F9D8 00 00 00 00 60 EC 78 02 00 00 00 00 00 00 00 00 ....'ÿx☻........
0130F9E8 00 00 00 00 00 00 00 00 80 0B FD 01 AC 6E FD 01 ........Çσ²0¼n²0
0130F9F8 0C 71 FD 01 14 F2 F9 01 FC 28 F8 01 00 00 00 00 .q²0¶=·0³(°0....
0130FA08 26 00 00 00 7C FE 1E 00 00 00 00 00 64 F9 1E 00 &...¦■▲.....d·▲.
0130FA18 00 00 00 00 9C FD 1E 00 00 00 00 00 74 69 6F 6E ....£²▲.....tion
0130FA28 24 00 00 00 12 00 00 00 01 00 00 00 01 00 00 00 $...↕...0...0...
0130FA38 3D 00 00 00 1E 01 00 00 34 95 50 00 E0 E3 30 01 =...▲0..4òP.α π00
0130FA48 0C 32 31 01 00 00 00 00 00 00 00 00 2C 45 F9 01 .210........,E·0
```

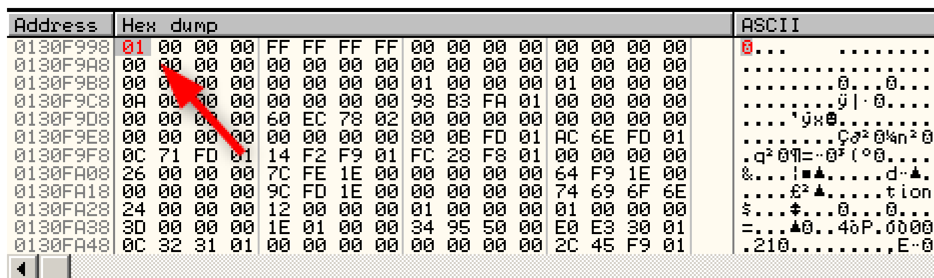
Hacemos clic con el botón derecho en “00” y seleccionamso “Binary” -> “Edit”:



Introducimos un “01”

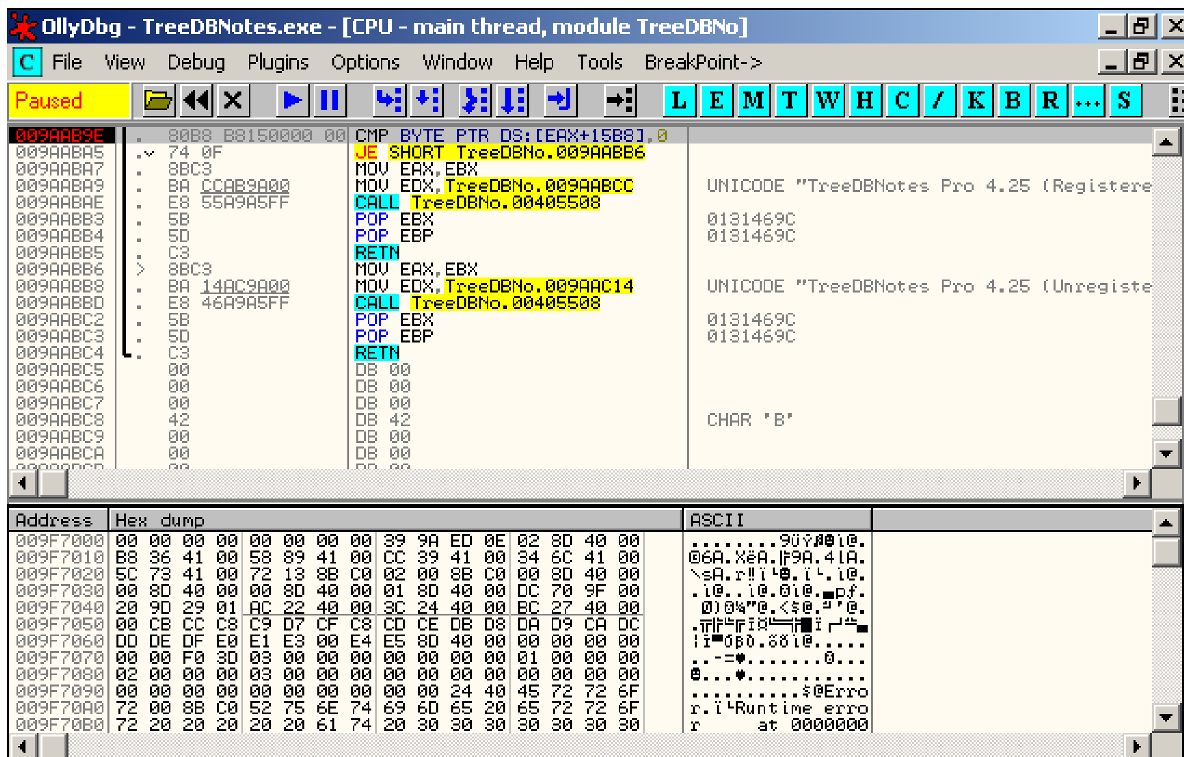


Vemo que la ventana Dump se ha actualizado.



Volvemos ejecutar la aplicación y Olly se detiene en el breakpoint.

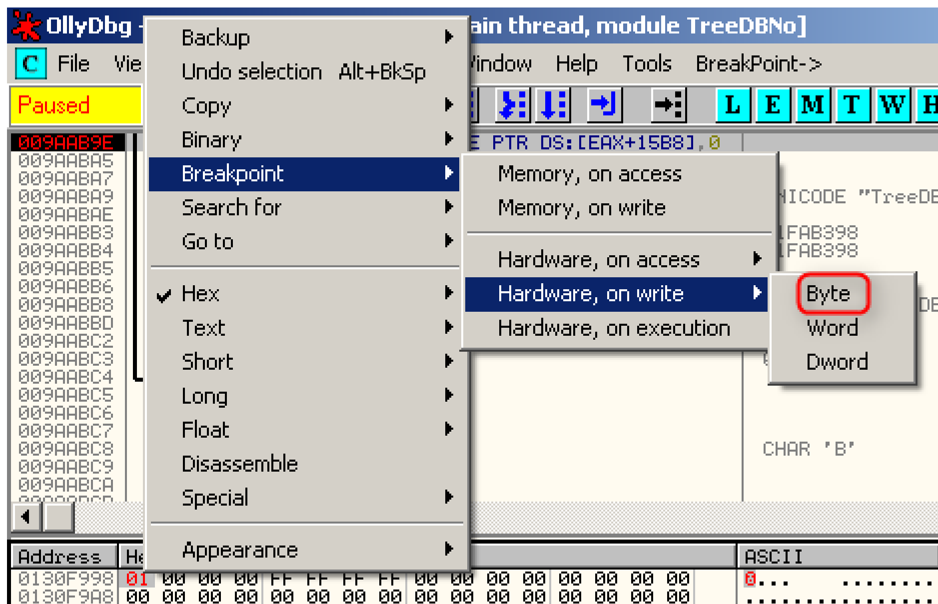
Vemos que en la ventana Dump desapareció el “01” y vuelve estar el “00”:



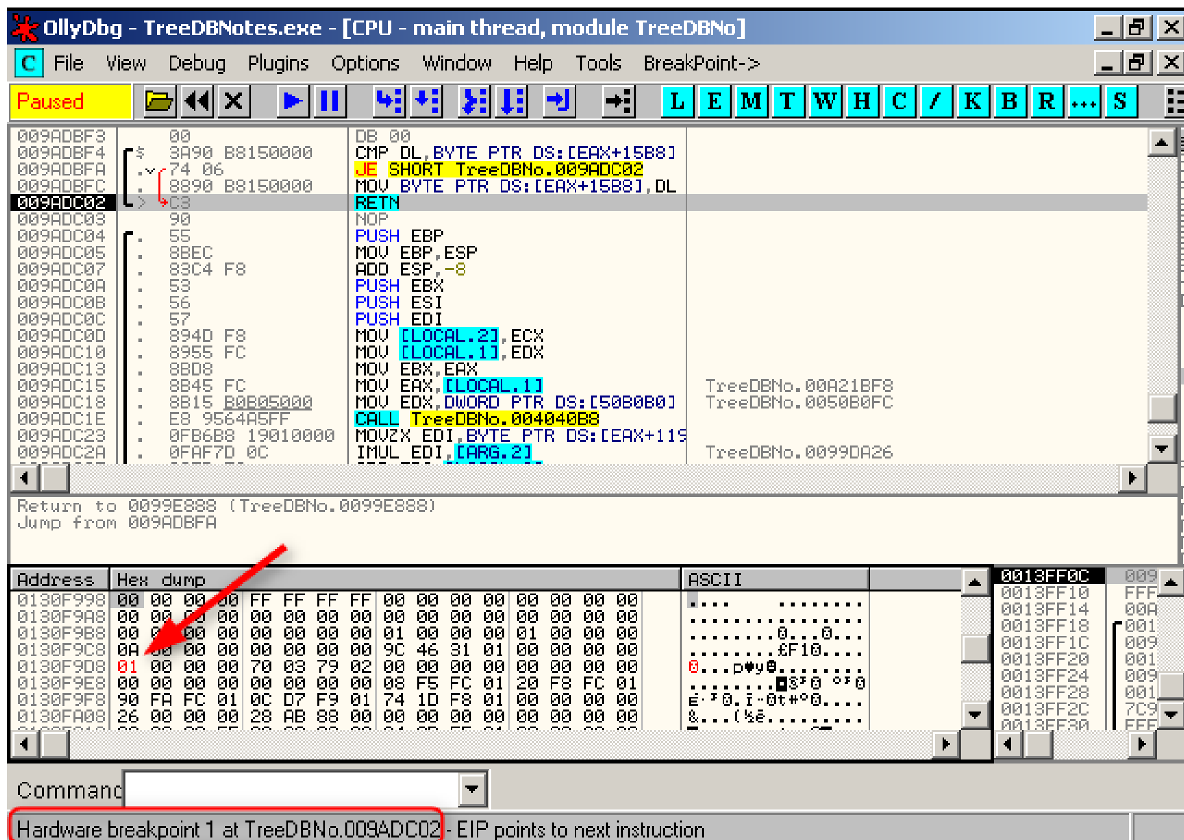
Esto significa que en algún lugar de la aplicación hay una segunda comprobación que resetea el valor de nuestra bandera otra vez a cero. El siguiente paso es pues localizar el lugar de esa comprobación. Para ello vamos a poner un “Hardware Breakpoint” en esta dirección de la memoria para decirle a Olly que pare siempre que la aplicación intente escribir algo en esa dirección. Elegiremos “write”, porque en algún lugar la aplicación escribe un cero en esa dirección.

Reiniciamos la aplicación, pulsamos F9 y Olly se detiene. Hacemos clic con el botón derecho sobre la instrucción CMP y volvemos a seleccionar “Follow in Dump” y cambiamos el primer binario de “00” a “01” Vemos que volvio a cambiar la dirección de la memoria.

Hacemos clic con el botón derecho en el primer valor de la ventana Dump y seleccionamos “Breakpoint” -> “Hardware, on wirte” -> “Byte”.



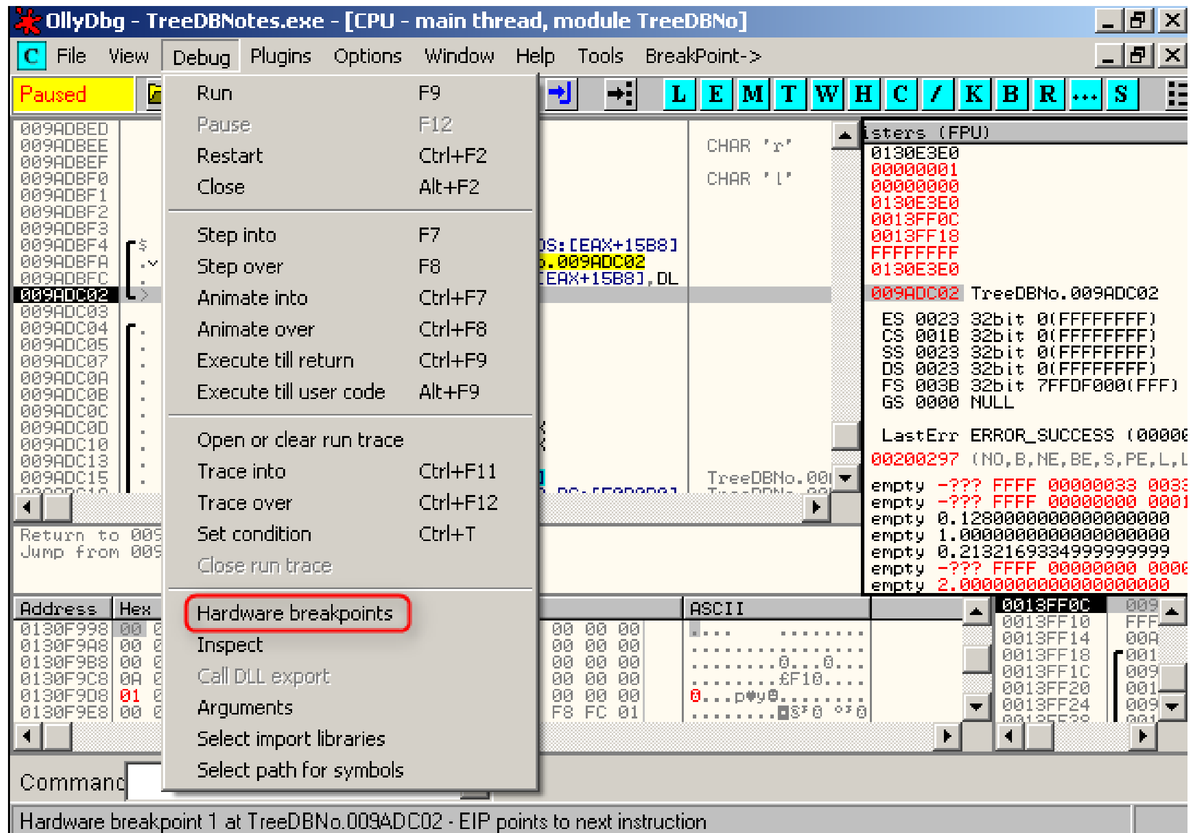
Pulsamos F9 y Olly se detiene en nuestro primer Breakpoint. Vemos que el valor "01" introducido anteriormente sigue ahí. Volvemos a ejecutar la aplicaión y Olly se detiene en una nueva sección. Si nos fijamos en la esquina inferior izquierda vemos que Olly se detuvo en nuestro "Hardware Breakpoint".

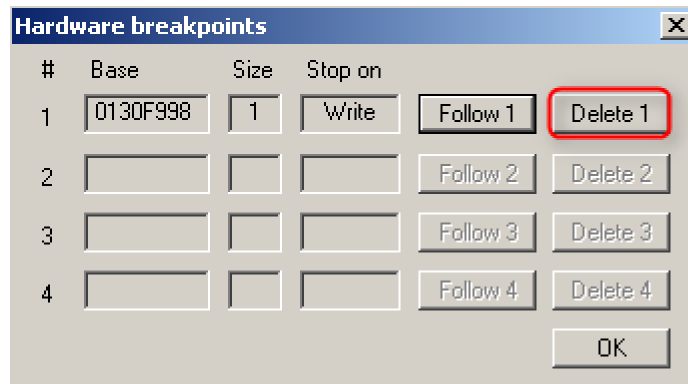


Continuemso estudiando el código. La primera instrucción compara DL con el contenido que hemos editado en la dirección de la memoria. Si ambos coinciden saltaremos hacia la dirección 9ADC02, donde tenemos una instrucción de retorno. Si no coincide, almacenaremos el contenido de DL en nuestra memoria. Ya sabemos que DL es igual a cero, ya

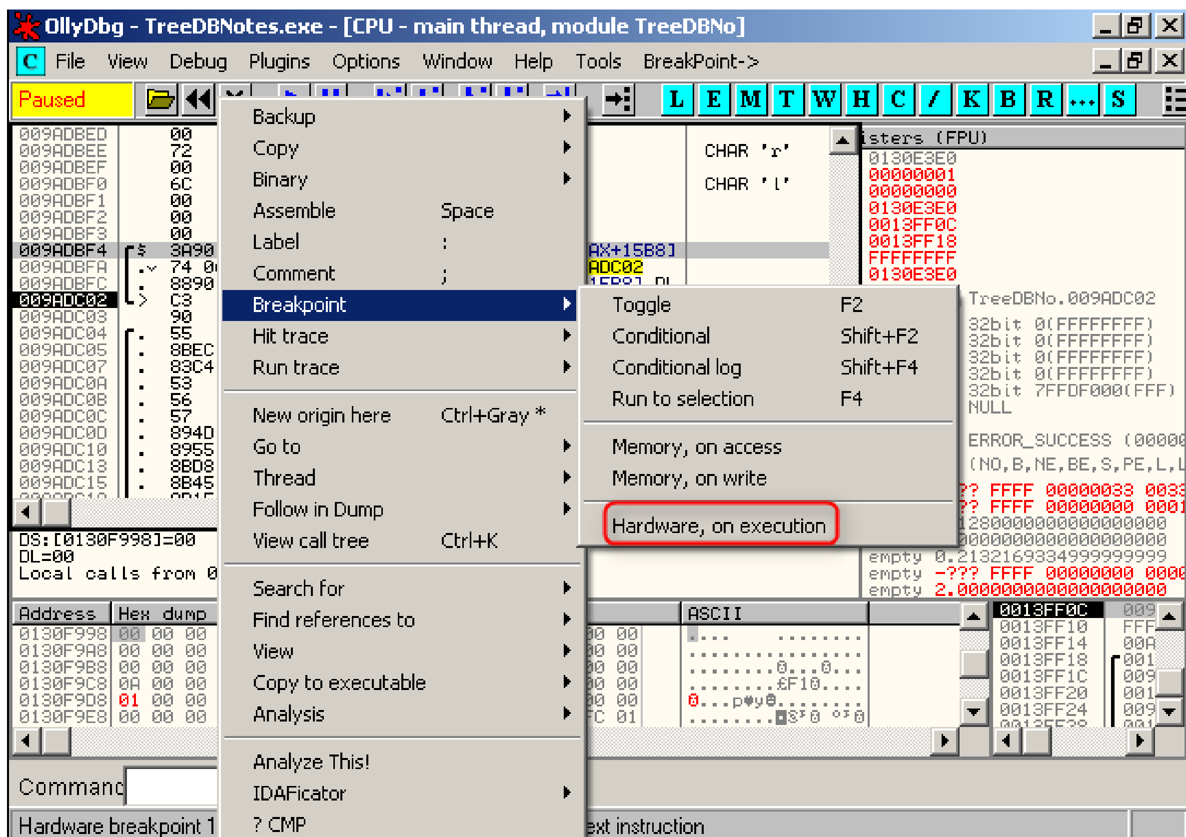
que vimos en nuestra dirección de la memoria cambiar el valor 01 de vuelta a 00. Así que esto es básicamente otra comprobación. Si la comparación es FALSE pondrá un 0 en la bandera de registrado / no registrado. Si el resultado de la comparación no falla, entonces todo queda como estaba.

Vamos a eliminar ahora nuestro hardware breakpoint. Para ello desde el menú, seleccionamos “Debug” -> “Hardware Breakpoints” -> “Delete 1”.

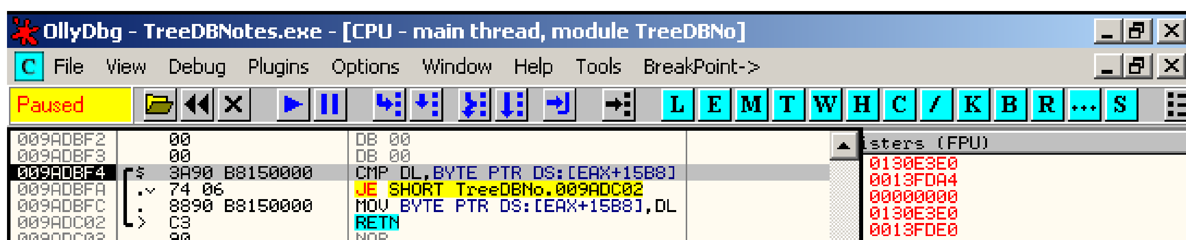




A continuación pondremos otro hardware breakpoint en la dirección 9ADBF4 para poder detenernos antes de que se ejecute la rutina. (Hemos seleccionado un hardware breakpoint y no otro por ser más robusto. Todos los demás breakpoints son eleminados por la aplicación.)



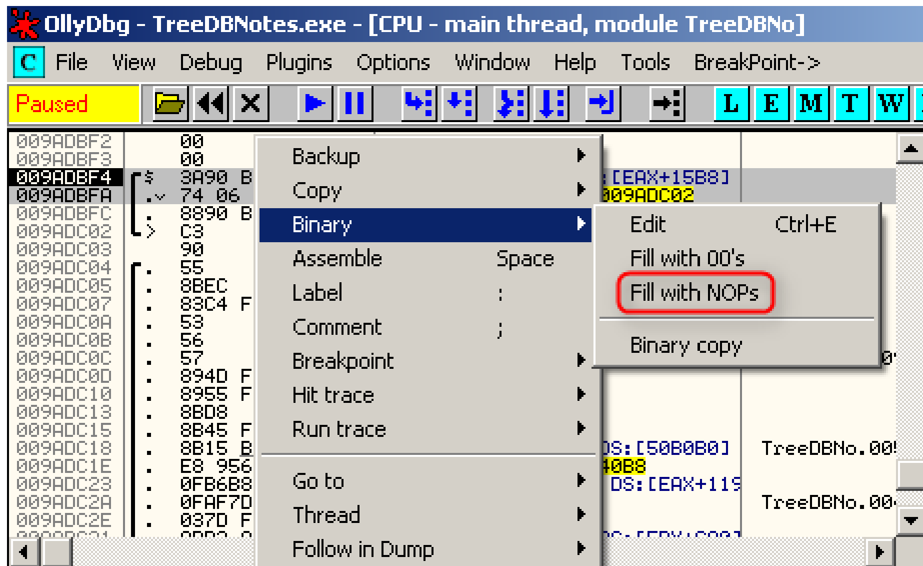
Reiniciamos la aplicación, pulsamos F9 y nos detenemos en nuestro nuevo Breakpoint.



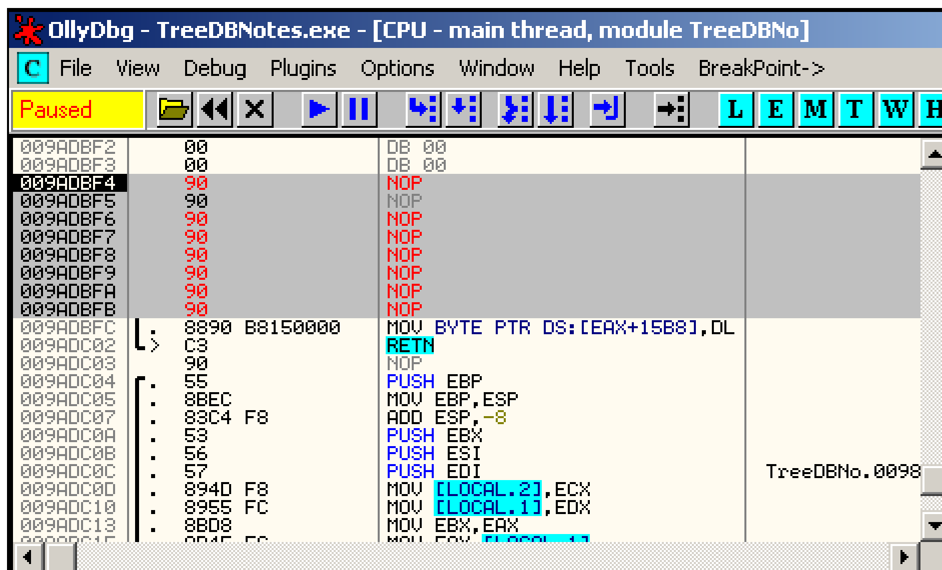
Analizemos lo que tenemos hasta ahora: Esta rutina es llamada antes de ser ejecutado el primer Breakpoint. Es una rutina que comprueba si estamos registrados o no. Si no lo estamos pone un cero en la dirección de la memoria que viene dada por [EAX+15B8], y si estamos registrado pondrá un 01 (o cualquier otro carácter distinto de cero). A continuación se llama a nuestra vieja rutina, la que imprime en la parte superior de la ventana si estamos registrados o no dependiendo si en ese lugar de la memoria hay un cero o un uno. Si nos aseguramos que se va a asignar un uno en ese lugar de la memoria, cada vez que se ejecute, todas las demas rutinas que vengan a continuación van a comprobar ese lugar en la memoria, verán que el valor es uno y pensarán que estamos registrados. Intentemos pues cambiar la rutina para que siempre ponga un uno en el lugar apropiado de la memoria.

Sabemos por la dirección 9ADBFC que DL ya tiene "algo" asignado en la memoria. Asi que podríamos cambiar ese "algo" a un uno. El problema de cambiar la DL por un uno es que esto cambiaría la longitud de la instrucción añadiendole un byte y esto sobreescribiría la instrucción RETN. Asi que lo que haremos es reemplazar las instrucciones CMP y JE en lugar de cargar un 01 en DL.

Seleccionamos por lo tanto ambas instrucciones, hacemos clic con el botón derecho y seleccionamos “Binary” -> “Fill with NOPs”.



El resultado que obtendremos es el siguiente:



A continuación hacemos clic en el primer NOP (9ADBF4) y pulsamos la barra espaciadora. Introducimos MOV DL, 1.

Hacemos clic en “Assemble” y “Cancel”.

```
OllyDbg - TreeDBNotes.exe - [CPU - main thread, module TreeDBNo]
C File View Debug Plugins Options Window Help Tools BreakPoint->
Paused                                      L E M T W H
009ADBF2      00               DB 00
009ADBF3      00               DB 00
009ADBF4      B2 01            MOV DL,1
009ADBF6      90               NOP
009ADBF7      90               NOP
009ADBF8      90               NOP
009ADBF9      90               NOP
009ADBFA      90               NOP
009ADBFB      90               NOP
009ADBFC    |.  8890 B8150000  MOV BYTE PTR DS:[EAX+15B8],DL
009ADC02    \>  C3             RETN
```

Siempre que se llame a esta rutina, se va a cargar un uno en la memoria en lugar de un cero. Pulsamos F9 y Olly se detiene en nuestro Breakpoint original.

```
OllyDbg - TreeDBNotes.exe - [CPU - main thread, module TreeDBNo]
C File View Debug Plugins Options Window Help Tools BreakPoint->
Paused                                      L E M T W H C / K B R ... S
009AAB98  /$  55               PUSH EBP
009AAB99  |.  8BEC             MOV EBP,ESP
009AAB9B  |.  53               PUSH EBX
009AAB9C  |.  8BDA             MOV EBX,EDX
009AAB9E  |.  80B8 B8150000 00 CMP BYTE PTR DS:[EAX+15B8],0
009AABA5  |.v 74 0F            JE SHORT TreeDBNo.009AABB6
009AABA7  |.  8BC3             MOV EAX,EBX
009AABA9  |.  BA CCAB9A00      MOV EDX,TreeDBNo.009AABCC      UNICODE "TreeDBNotes Pro 4.25 (Registere
009AABAE  |.  E8 55A9A5FF      CALL TreeDBNo.00405508
009AABB3  |.  5B               POP EBX                        0131469C
009AABB4  |.  5D               POP EBP                        0131469C
009AABB5  |.  C3               RETN
009AABB6  |>  8BC3             MOV EAX,EBX
009AABB8  |.  BA 14AC9A00      MOV EDX,TreeDBNo.009AAC14      UNICODE "TreeDBNotes Pro 4.25 (Unregiste
009AABBD  |.  E8 46A9A5FF      CALL TreeDBNo.00405508
009AABC2  |.  5B               POP EBX                        0131469C
009AABC3  |.  5D               POP EBP                        0131469C
009AABC4  \.  C3               RETN
009AABC5      00               DB 00
009AABC6      00               DB 00
009AABC7      00               DB 00
009AABC8      42               DB 42                          CHAR 'B'
009AABC9      00               DB 00
009AABCA      00               DB 00
009AABCB      00               DB 00
009AABCC   .  5400 7200 6500 6 UNICODE "TreeDBNo"
009AABDC   .  7400 6500 7300 2 UNICODE "tes Pro "
009AABEC   .  3400 2E00 3200 3 UNICODE "4.25 (Re"
009AABFC   .  6700 6900 7300 7 UNICODE "gistered"
009AAC0C   .  2900 0000        UNICODE ")",0
009AAC10      46               DB 46                          CHAR 'F'
009AAC11      00               DB 00
009AAC12      00               DB 00
009AAC13      00               DB 00
009AAC14   .  5400 7200 6500 6 UNICODE "TreeDBNo"
```

Pulsamos F8 y vemos que pasaremos por nuestro “good boy”.

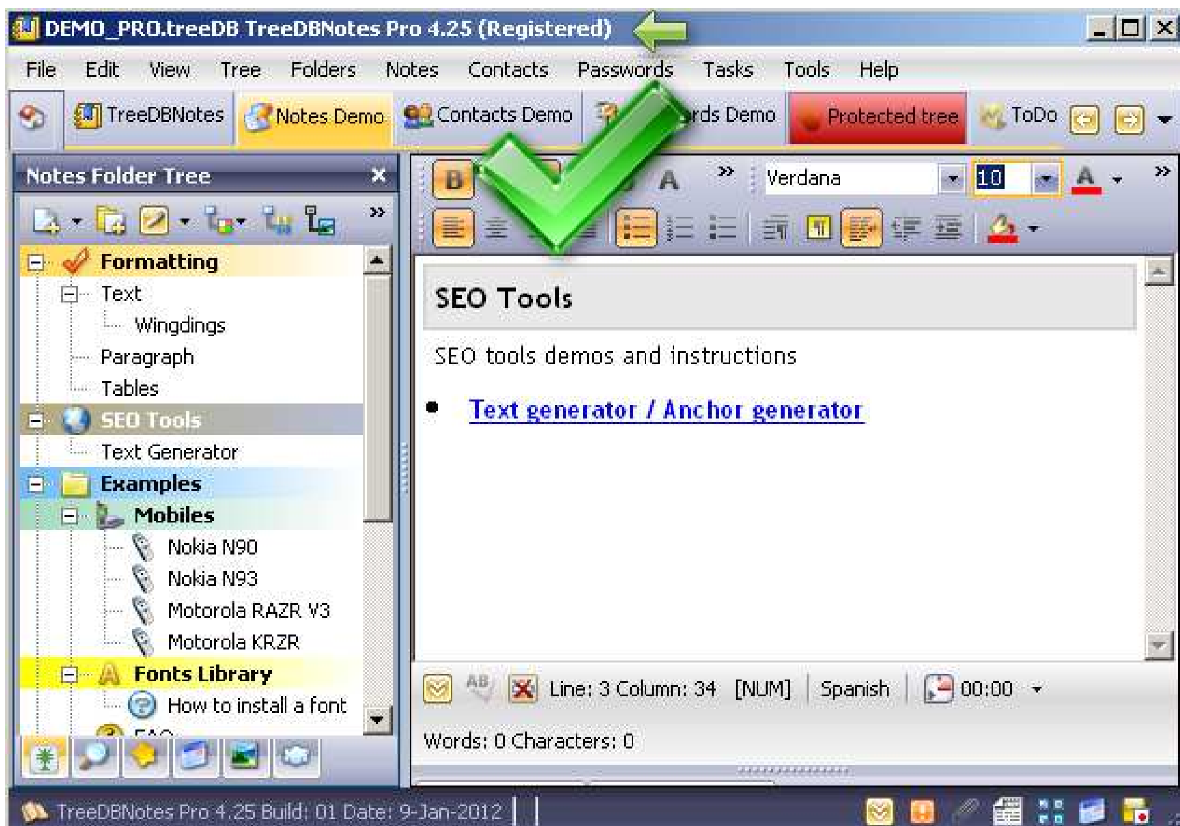
```
OllyDbg - TreeDBNotes.exe - [CPU - main thread, module TreeDBNo]
C File View Debug Plugins Options Window Help Tools BreakPoint->
Paused                                    L E M T W H C / K B R ... S
009AAB98  $ 55              PUSH EBP
009AAB99  . 8BEC            MOV EBP,ESP
009AAB9B  . 53              PUSH EBX
009AAB9C  . 8BDA            MOV EBX,EDX
009AAB9E  . 80B8 B8150000 00 CMP BYTE PTR DS:[EAX+15B8],0
009AABA5  . 74 0F           JE SHORT TreeDBNo.009AABB6
009AABA7  . 8BC3            MOV EAX,EBX
009AABA9  . BA CCAB9A00     MOV EDX,TreeDBNo.009AABCC        UNICODE "TreeDBNotes Pro 4.25 (Registere
009AABAE  . E8 55A9A5FF     CALL TreeDBNo.00405508
009AABB3  . 5B              POP EBX                          0131469C
009AABB4  . 5D              POP EBP                          0131469C
009AABB5  . C3              RETN
009AABB6  > 8BC3            MOV EAX,EBX
009AABB8  . BA 14AC9A00     MOV EDX,TreeDBNo.009AAC14        UNICODE "TreeDBNotes Pro 4.25 (Unregiste
009AABBD  . E8 46A9A5FF     CALL TreeDBNo.00405508
009AABC2  . 5B              POP EBX                          0131469C
009AABC3  . 5D              POP EBP                          0131469C
009AABC4  . C3              RETN
009AABC5    00              DB 00
009AABC6    00              DB 00
009AABC7    00              DB 00
009AABC8    42              DB 42                            CHAR 'B'
009AABC9    00              DB 00
009AABCA    00              DB 00
009AABCB    00              DB 00
009AABCC  . 5400 7200 6500 6 UNICODE "TreeDBNo"
009AABDC  . 7400 6500 7300 2 UNICODE "tes Pro "
009AABEC  . 3400 2E00 3200 3 UNICODE "4.25 (Re"
009AABFC  . 6700 6900 7300 7 UNICODE "gistered"
009AAC0C  . 2900 0000       UNICODE ")",0
009AAC10    46              DB 46                            CHAR 'F'
009AAC11    00              DB 00
009AAC12    00              DB 00
009AAC13    00              DB 00
009AAC14  . 5400 7200 6500 6 UNICODE "TreeDBNo"
```

Pulsamos F9 y nos detendremos en la rutina que comprueba si estamos registrados o no.

```
OllyDbg - TreeDBNotes.exe - [CPU - main thread, module TreeDBNo]
C File View Debug Plugins Options Window Help Tools BreakPoint->
Paused                                    L E M T W H C / K B R ... S
009ADBF4    B2 01           MOV DL,1
009ADBF6    90              NOP
009ADBF7    90              NOP
009ADBF8    90              NOP
009ADBF9    90              NOP
009ADBFA    90              NOP
009ADBFB    90              NOP
009ADBFC  . 8890 B8150000   MOV BYTE PTR DS:[EAX+15B8],DL
009ADC02  > C3              RETN
009ADC03    90              NOP
009ADC04  . 55              PUSH EBP
009ADC05  . 8BEC            MOV EBP,ESP
009ADC07  . 83C4 F8         ADD ESP,-8
009ADC0A  . 53              PUSH EBX
009ADC0B  . 56              PUSH ESI
009ADC0C  . 57              PUSH EDI
009ADC0D  . 894D F8         MOV [LOCAL.2],ECX
009ADC10  . 8955 FC         MOV [LOCAL.1],EDX
009ADC13  . 8BD8            MOV EBX,EAX
009ADC15  . 8B45 FC         MOV EAX,[LOCAL.1]                TreeDBNo.00A21BF8
009ADC18  . 8B15 B0B05000   MOV EDX,DWORD PTR DS:[50B0B0]    TreeDBNo.0050B0FC
009ADC1E  . E8 9564A5FF     CALL TreeDBNo.004040B8
009ADC23  . 0FB6B8 19010000 MOVZX EDI,BYTE PTR DS:[EAX+119
009ADC2A  . 0FAF7D 0C       IMUL EDI,[ARG.2]                 TreeDBNo.0099DA26
009ADC2E  . 037D F8         ADD EDI,[LOCAL.2]
009ADC31  . 8BB3 A00C0000   MOV ESI,DWORD PTR DS:[EBX+CA0]
009ADC37  . 897E 0C         MOV DWORD PTR DS:[ESI+C],EDI
009ADC3A  . 8BD7            MOV EDX,EDI
009ADC3C  . 8BC6            MOV EAX,ESI
009ADC3E  . E8 6D6DB3FF     CALL TreeDBNo.004E49B0
009ADC43  . 8BC3            MOV EAX,EBX
```

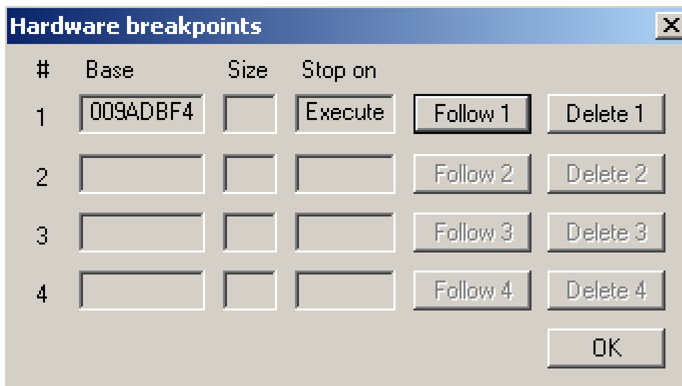
Pulsamos F9 unas cuantas veces más hasta que finalmente aparezca la siguiente ventana:

Hacemos clic en “Open demo file” y vemos que estamos registrados.

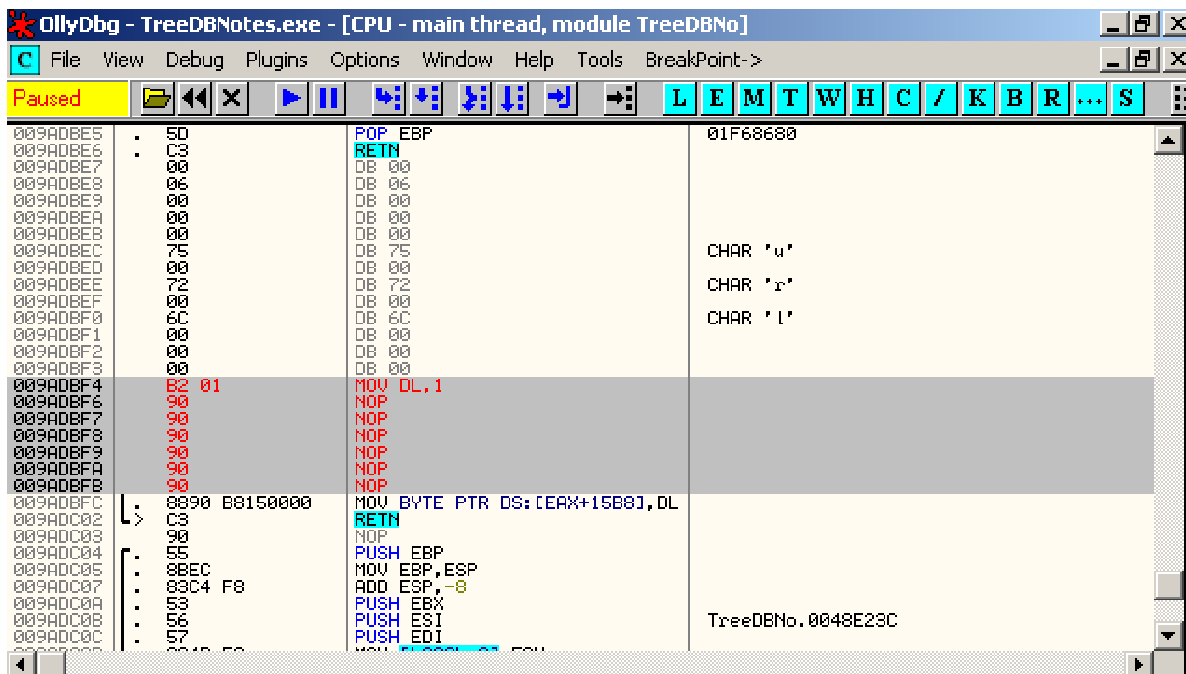


Finalmente guardamos la aplicación parcheada en nuestro disco duro.

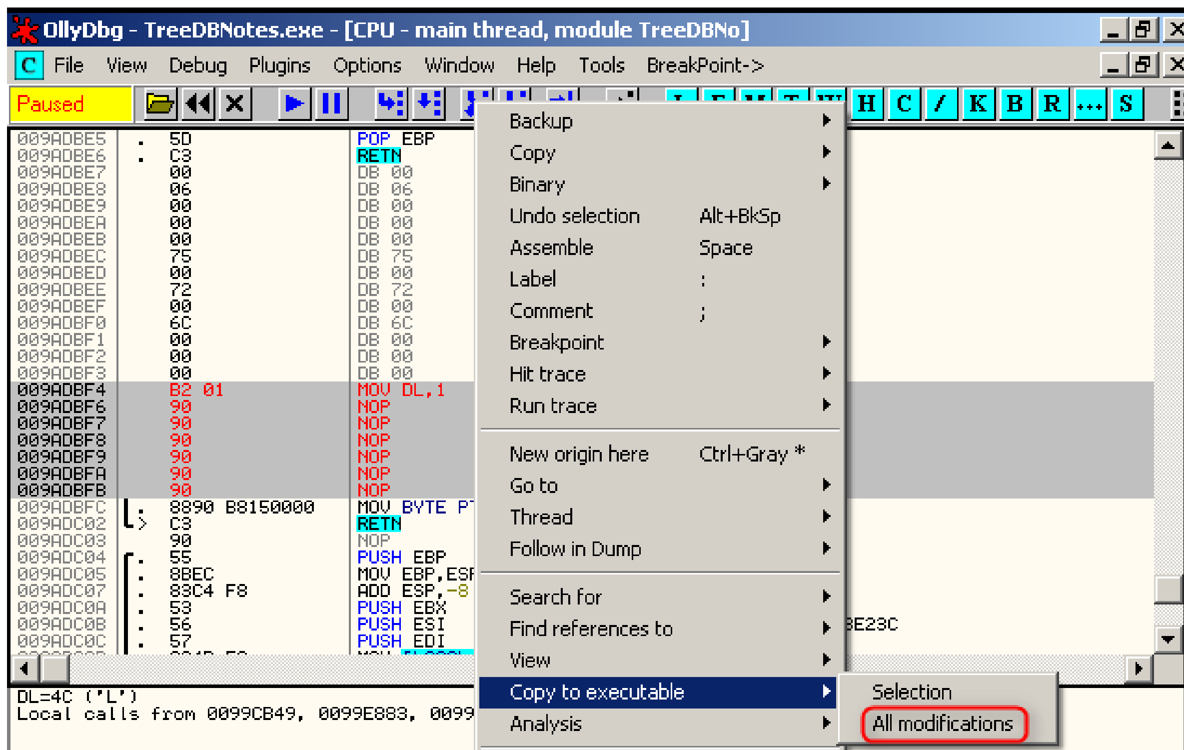
Seleccionamos “Debug” -> “Hardware Breakpoints” y hacemos clic en el botón “Follow 1”, lo que nos llevará a nuestro parche.



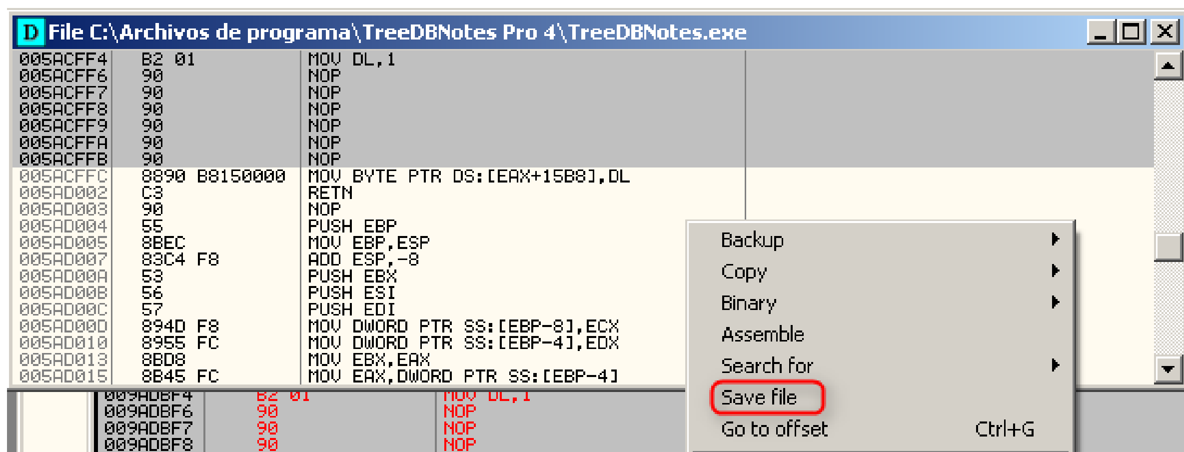
Seleccionamos todos los cambios que hizimos.



Hacemos clic con el botón derecho: “Copy to executable” -> “All modifications” -> “Copy all”.

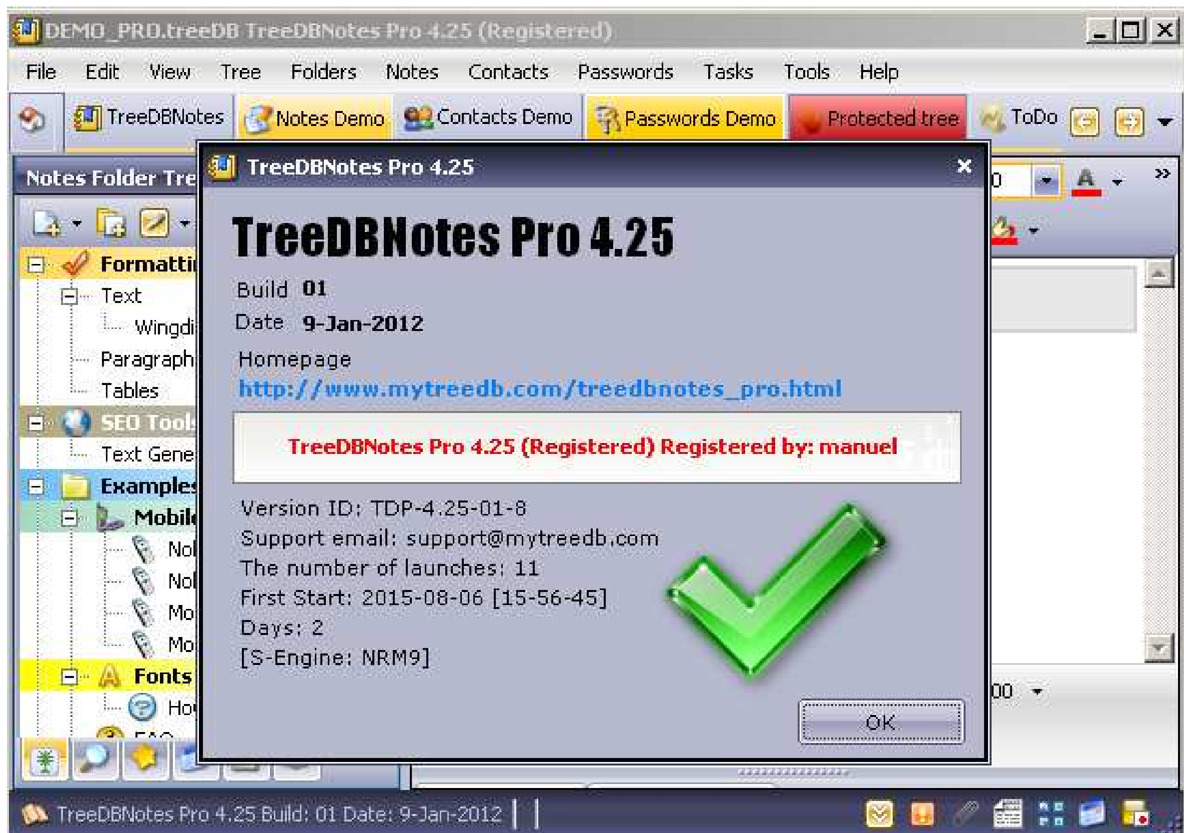


En la ventana que se abre hacemos clic con el botón derecho y seleccionamso “Save file”.



Cerramos Olly y ejecutamos la aplicación recien creada:

Papelera de reciclaje
Mis documentos
OLLYDBG
ResHacker
TreeDBNotes Pro 4
TreeDBNotes



DEMO_PRO.treeDB TreeDBNotes Pro 4.25 (Registered)
File Edit View Tree Folders Notes Contacts Passwords Tasks Tools Help
TreeDBNotes Notes Demo Contacts Demo Passwords Demo Protected tree ToDo
TreeDBNotes Pro 4.25
TreeDBNotes Pro 4.25
Build 01
Date 9-Jan-2012
Homepage
http://www.mytreedb.com/treedbnotes_pro.html
TreeDBNotes Pro 4.25 (Registered) Registered by: manuel
Version ID: TDP-4.25-01-8
Support email: support@mytreedb.com
The number of launches: 11
First Start: 2015-08-06 [15-56-45]
Days: 2
[S-Engine: NRM9]
OK
TreeDBNotes Pro 4.25 Build: 01 Date: 9-Jan-2012

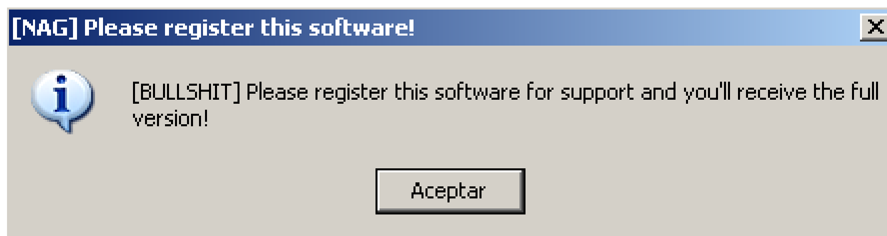
7.11 Caso práctico 11: NAGS

Los Nags, o “nag screens” son cuadros de mensajes que aparecen de vez en cuando para recordarnos que el periodo de prueba está a punto de acabar y que tenemos que registrarnos si deseamos continuar disfrutando de la aplicación. Deshacerse de estos molestos nags es un tema central en ingeniería inversa por lo que se estudiaran en este apartado dos aplicaciones con nags.

Se va añadir también un nuevo plugin a Olly, llamado IDAFicator.

7.11.1 Nag1

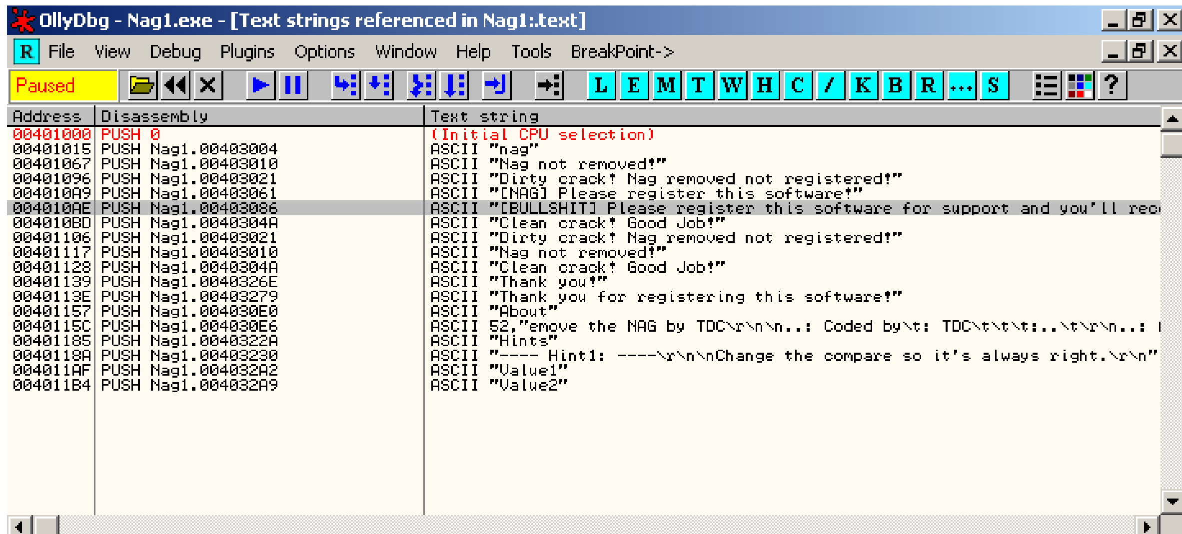
Ejecutamos el programa haciendo doble clic sobre el ejecutable.



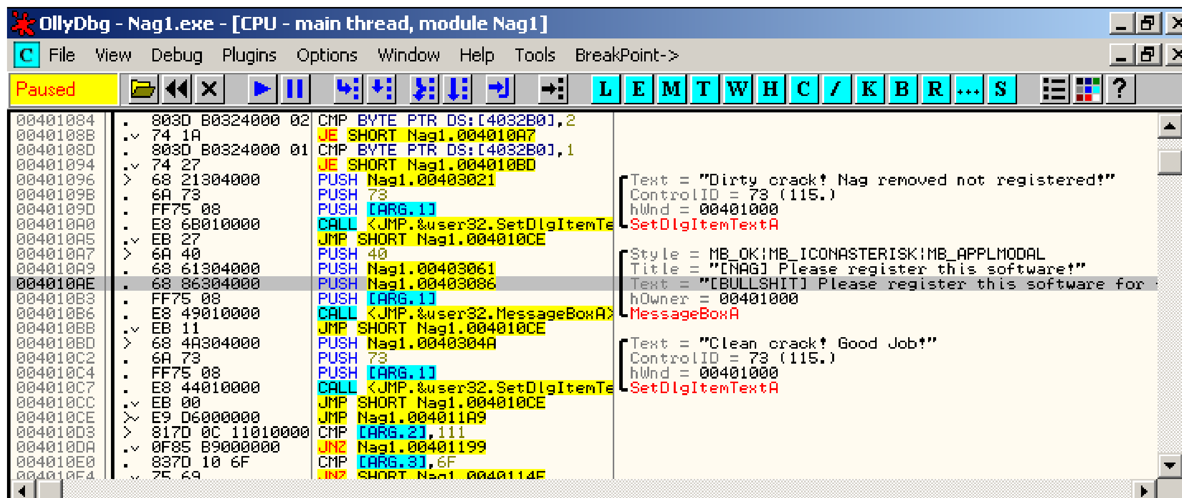
Hacemos clic en Aceptar lo que nos lleva a la pantalla principal.

Vemos una pequeña instrucción de lo que tenemos que hacer y debajo una pestaña de “Hints”. Pulsamos sobre “Hints” lo que nos desvela información muy valiosa.

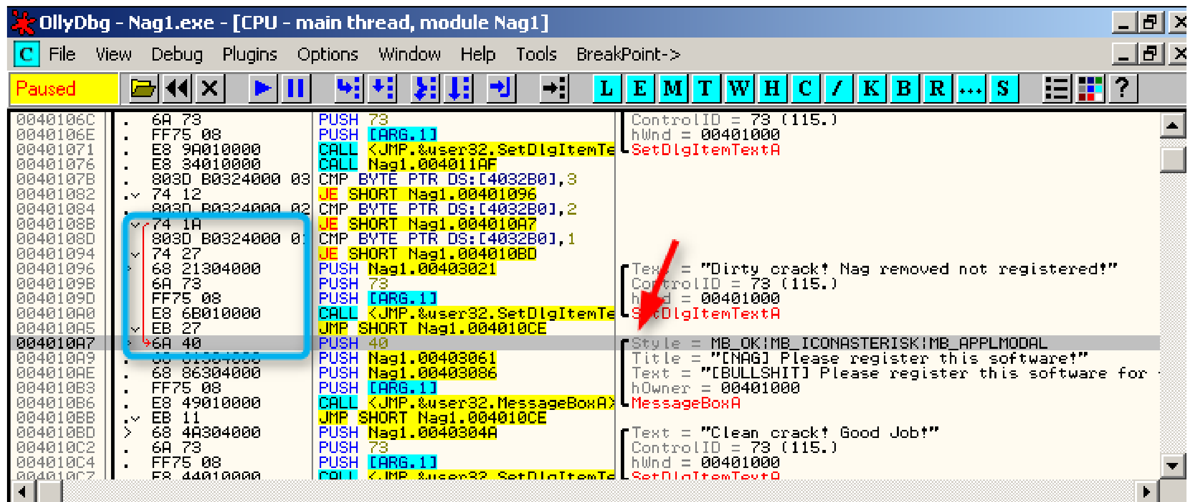
Una vez estudiado el comportamiento del programa, abrimos Olly y cargamos la aplicación. A continuación buscamos cadenas de texto:



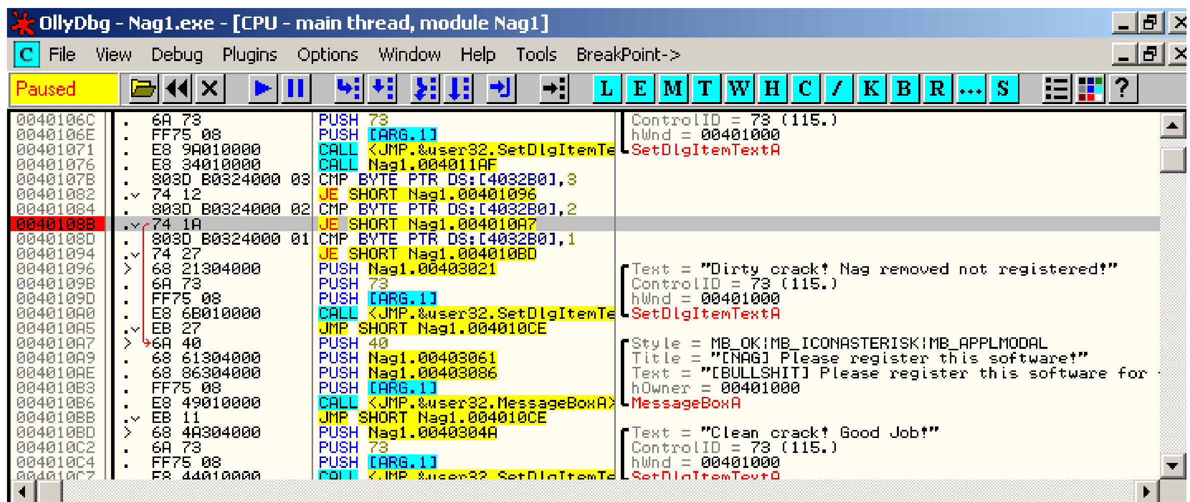
En la dirección 4010AE vemos el texto que corresponde a la ventana del nag. Hacemos doble clic y saltamos al código que nos interesa.



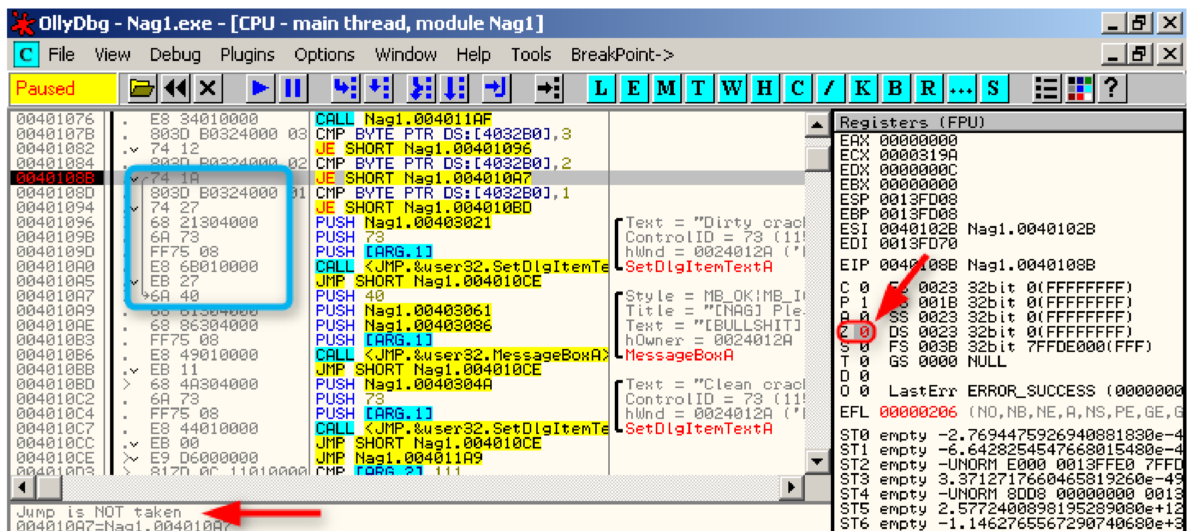
Hacemos clic en la primera línea del cuadro de mensaje en la dirección 4010A7 para averiguar el lugar de llamada:



Vemos que la llamada procede de la instrucción JE en la dirección 40108B, justo después de una instrcción CMP. Pongamos pues un Breakpoint en la instrucción JE.

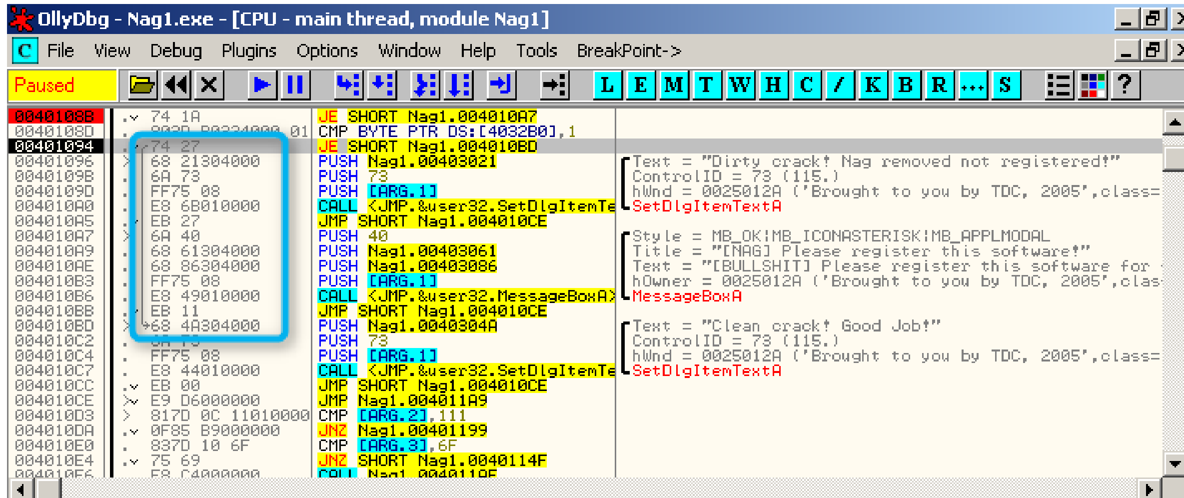


Reiniciamos la aplicación y pulsamos F9. Nos detenemos en el Breakpoint. Cambiamos el valor de la bandera Z para no tomar el salto:

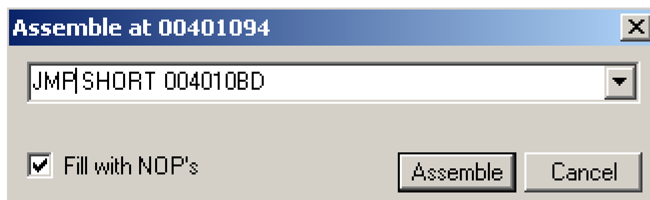


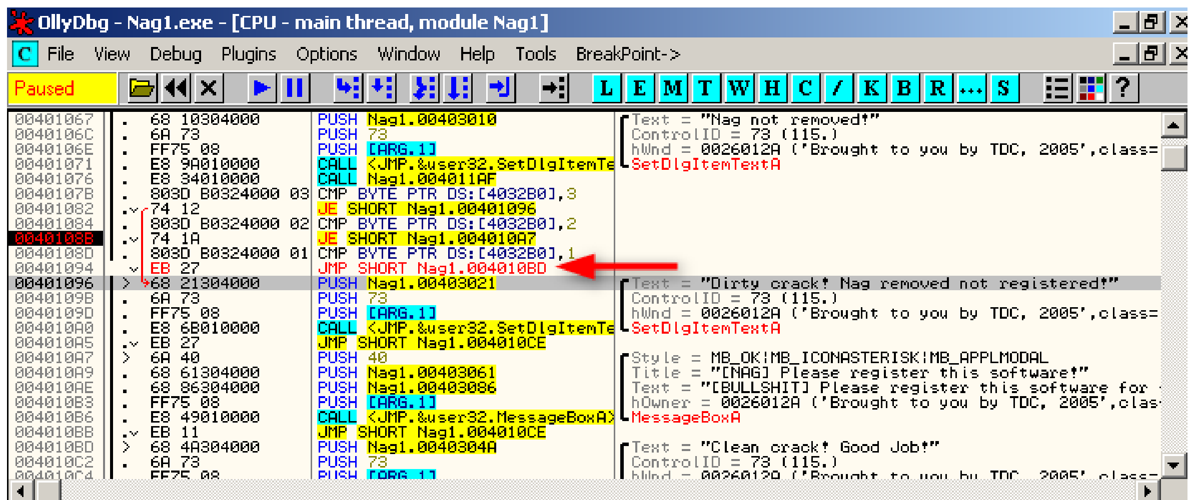
Pulsamos F9.

Parece que no nos hemos parado demasiado en analizar el código. Reiniciamos Olly, pulsamos F9 y nos detenemos en el Breakpoint. Volvemos a cambiar el valor de la bandera Z. Pulsamos F8 dos veces hasta llegar al siguiente salto.



Vemos que saltamos hacia el “good boy”. Cambiemos por tanto la instrucción para que salte siempre al llegar a 401094.





Pulsamos F9.

Podríamos guardar este parche y cambiar el primero en la dirección 40108B para que nunca salte y asi guardar los dos parches para crear un nuevo ejecutable que guradariamos en nuestro disco duro. Pero llegado a este punto es conveniente resaltar que por regla general siempre va a ver varias formas de parchear un programa. Así que veremos a continuación una forma alternativa de parchear la aplicación.

Si nos situamos en nuestro Breakpoint podremos observar que la colección de instrucciones ahí presentes se podría traducir en un lenguaje de alto nivel de la siguiente forma:

if (contents of 4032B0 == 3)

jump “Dirty Crack”

else if(contents of 4032B0 == 2)

jump to “Show Nag Screen

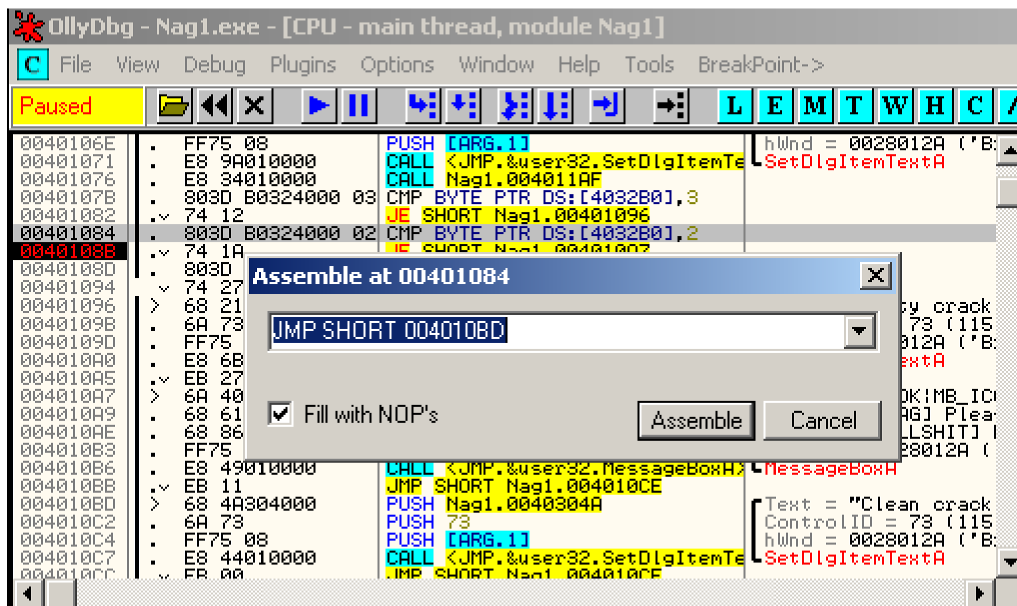
else if (contents of 4032B0 == 1)

jump to Good Boy Msg

else

Display “Dirty Crack”

Sabemos que el contenido de la dirección en memoria 4032B0 siempre será 2 ya que por defecto siempre aparece el nag. ¿Qué pasaría si nos olvidamos de todas estas clausulas if/then y saltamos directamente hacia nuestro “good boy”? De esta forma, al reemplazar el primer salto para que salte siempre al “good boy”, solo necesitamos un parche.



Reiniciamos la aplicación y activamos el parche en 401084.

Patches

Address	Size	State	Old	New	Comment
00401084	7.	Active	CMP BYTE PTR DS:[4032B0],2	JMP SHORT Nag1.004010BD	
00401094	2.	Removed	JE SHORT Nag1.004010BD	JMP SHORT Nag1.004010BD	

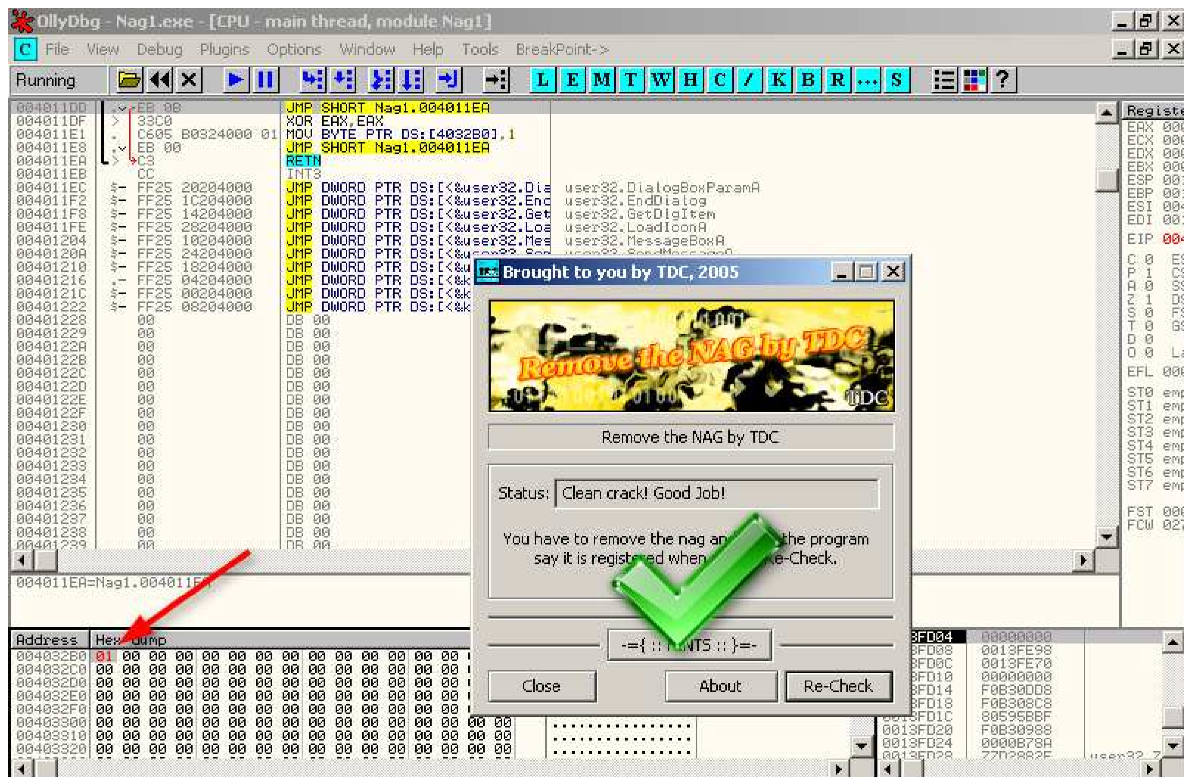
Pulsamos F9.

```
OllyDbg - Nag1.exe - [CPU - main thread, module Nag1]
00401084   EB 37            JMP SHORT Nag1.004010BD
00401086   90               NOP
00401087   90               NOP
00401088   90               NOP
00401089   90               NOP
0040108A   90               NOP
0040108B   74 1A            JE SHORT Nag1.004010A7
0040108D   803D B0324000 01 CMP BYTE PTR DS:[4032B0],1
00401094   74 27            JE SHORT Nag1.004010BD
00401096   68 21304000      PUSH Nag1.00403021               Text = "Dirty crack! Nag removed not registered!"
0040109B   6A 73            PUSH 73                          ControlID = 73 (115.)
0040109D   FF75 08          PUSH [ARG.1]                     hWnd = 7C92B686
004010A0   E8 6B010000      CALL <JMP.&user32.SetDlgItemTe   SetDlgItemTextA
004010A5   EB 27            JMP SHORT Nag1.004010CE
004010A7   6A 40            PUSH 40                          Style = MB_OK|MB_ICONASTERISK|MB_APPLMODAL
004010A9   68 61304000      PUSH Nag1.00403061               Title = "[NAG] Please register this software!"
004010AE   68 86304000      PUSH Nag1.00403086               Text = "[BULLSHIT] Please register this software for
004010B3   FF75 08          PUSH [ARG.1]                     hOwner = 7C92B686
004010B6   E8 49010000      CALL <JMP.&user32.MessageBoxA>   MessageBoxA
004010BB   EB 11            JMP SHORT Nag1.004010CE
004010BD   68 4A304000      PUSH Nag1.0040304A               Text = "Clean crack! Good Job!"
004010C2   6A 73            PUSH 73                          ControlID = 73 (115.)
004010C4   FF75 08          PUSH [ARG.1]                     hWnd = 7C92B686
004010C7   E8 44010000      CALL <JMP.&user32.SetDlgItemTe   SetDlgItemTextA
```

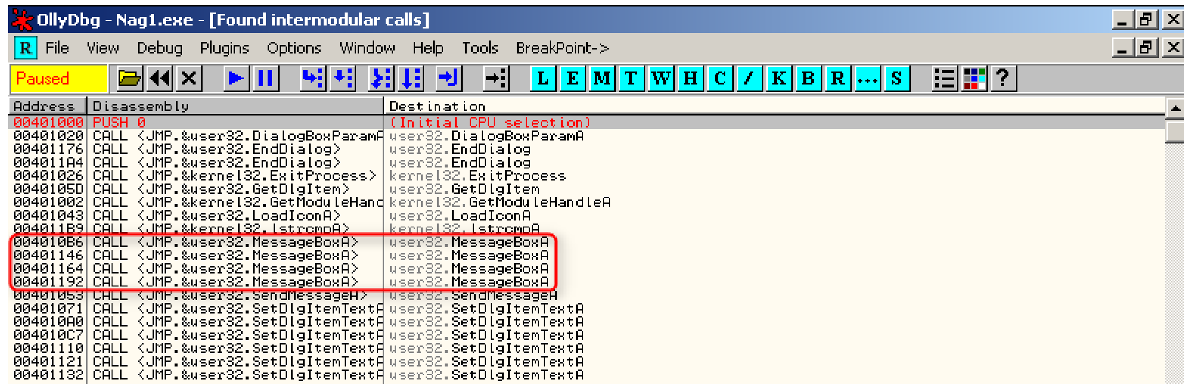
Podemos ver que hemos llegado al mismo resultado.

Otra solución incluso más elegante sería pensar: “Si el contenido de 4032B0 siempre es igual a dos, y para alcanzar el ‘good boy’ necesitamos que sea igual a uno, porque no poner un uno en ese lugar de la memoria de tal forma que siempre llegaríamos al ‘good boy’.”

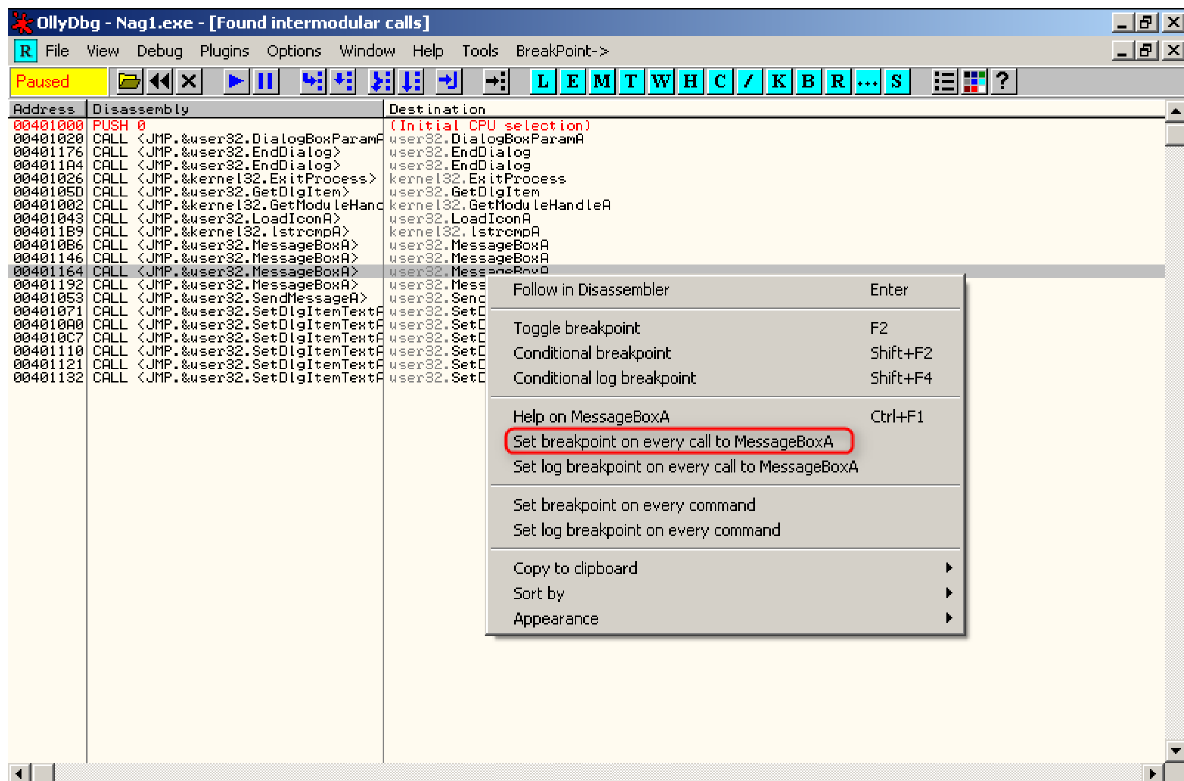
Reiniciamos la aplicación, seguimos la dirección 4032B0 en la ventana Dump e editamos el binario para que sea igual a uno.



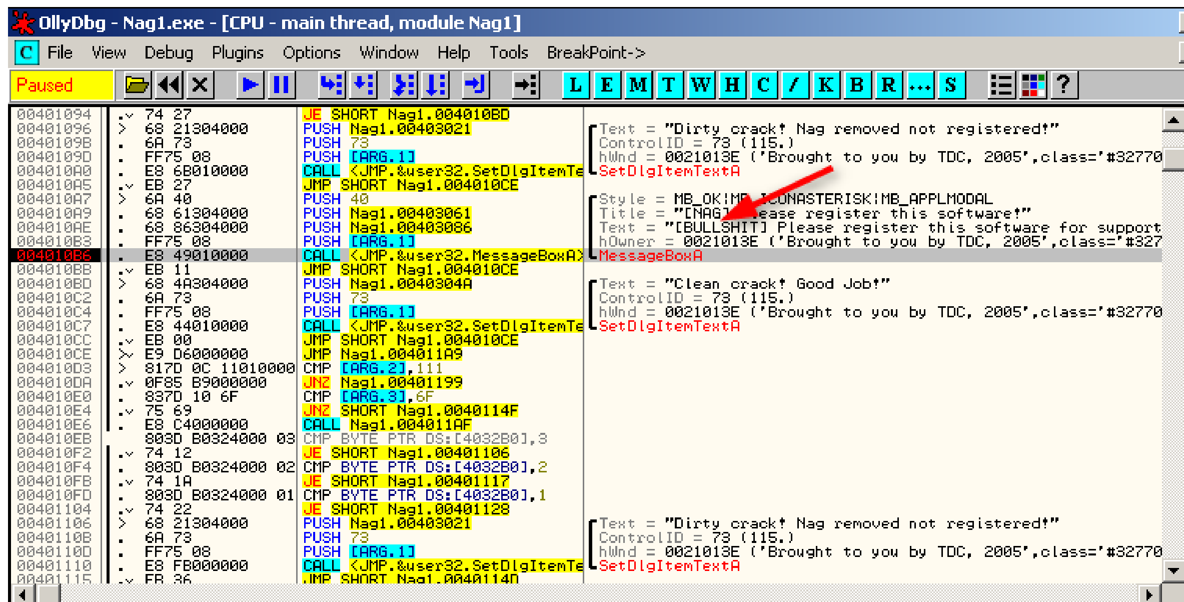
Otra cosa a tener en cuenta es que siempre va haber varias formas de llegar al área que nos interesa estudiar. Por ejemplo, si no existen cadenas de texto u existiendo no nos aportan una ayuda significativa, podemos buscar por “Intermodular calls”.



En nuestro caso existen cuatro llamadas a MessageBoxA. Hacemos clic con el botón derecho en cualquiera de ellos y seleccionamos “Set breakpoint on every call to MessageBoxA”.



Si ahora ejecutamos la aplicación nos detendremos en la siguiente línea de código:

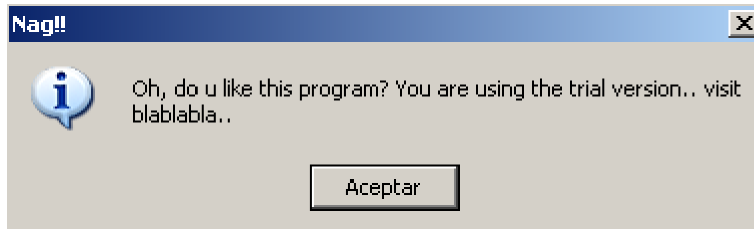


Que es precisamente el nag que aparece nada más ejecutar el programa.

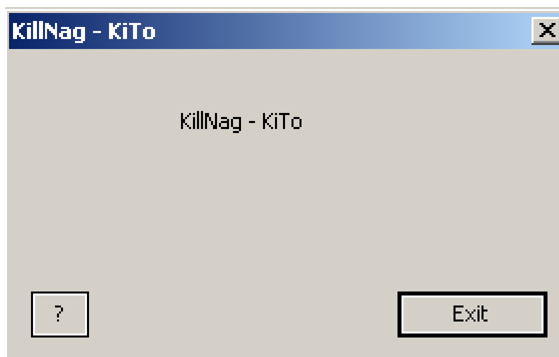
Conclusión: tengamos en mente que siempre vamos a tener varias opciones para conseguir nuestro objetivo.

7.11.2. Nag2

Hacemos doble clic sobre el ejecutable nag2.exe y aparece la siguiente ventana:

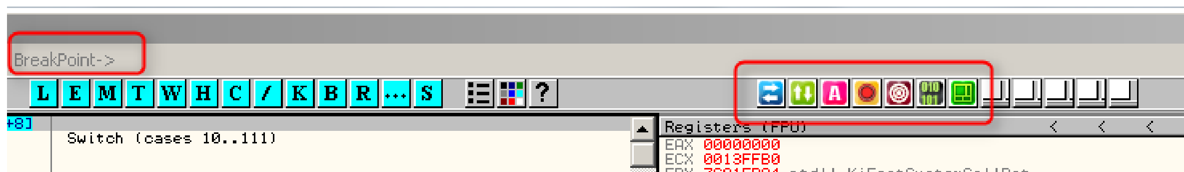


Y después de hacer clic en Aceptar se abre la ventana principal:



Cerramos la aplicación y la cargamos en Olly.

Nota*: Si hemos instalado el plugin IDAFicator nos aparecerá entre otras cosas un grupo de botones en la barra de herramientas que hará nuestra busqueda de textos de cadenas mucho más facil. También tendremos acceso a un nuevo elemento en la barra de menú: "BreakPoint->". Si desplegamos la pestaña podemos poner breakpoints a las API's de forma automática.*



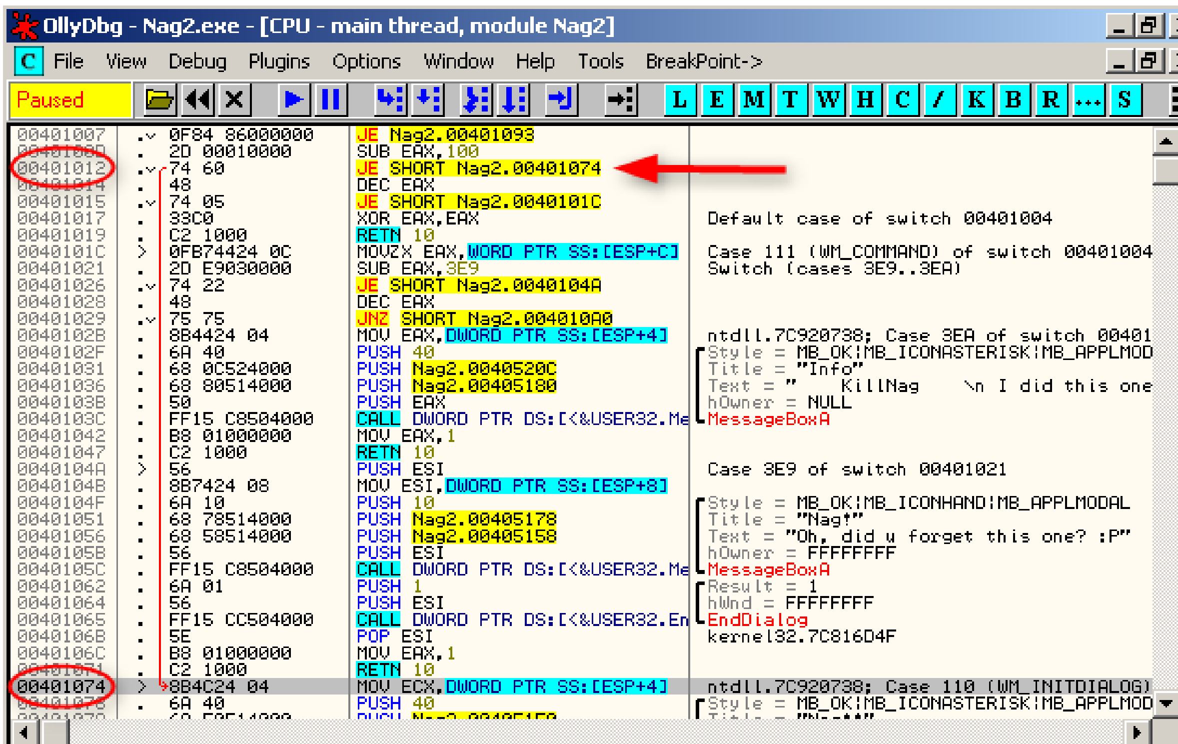
Pulsemos pues el siguiente botón para acceder a las cadenas de texto:

Hacemos doble clic sobre la séptima línea en la que aparece el texto de nuestro nag.

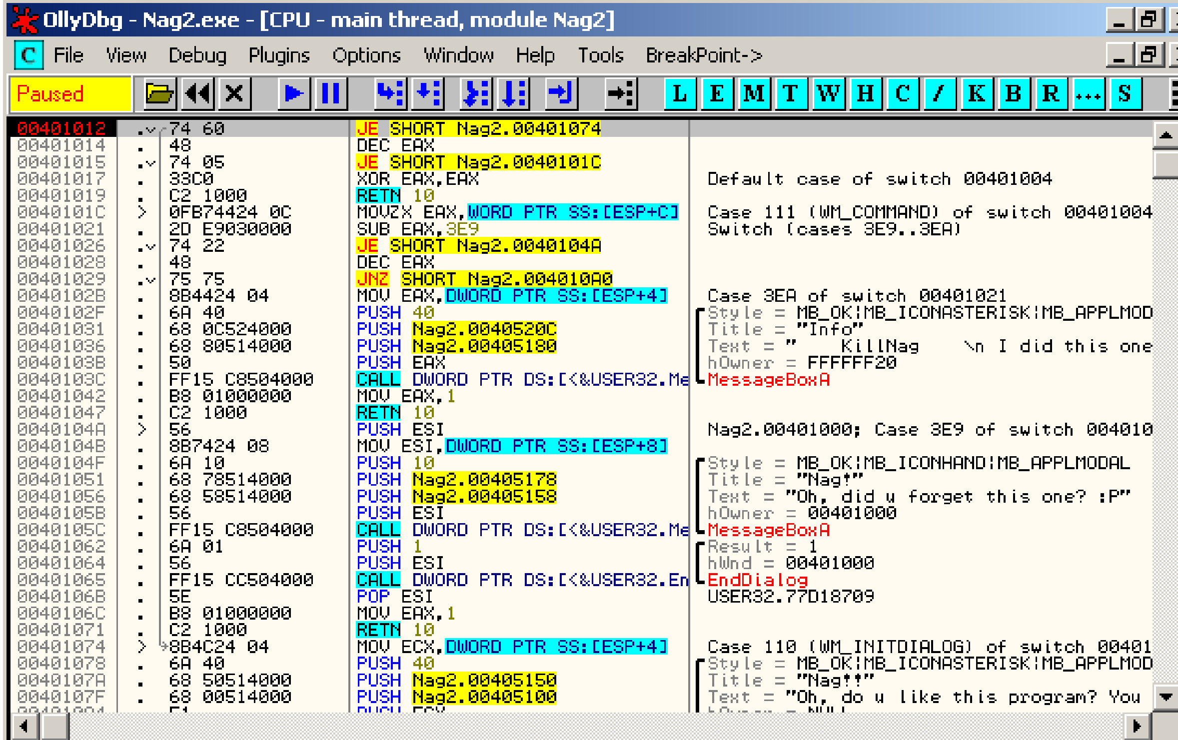
```
OllyDbg - Nag2.exe - [Found strings are:]
R File View Debug Plugins Options Window Help Tools BreakPoint->
Paused                                                   L E M T W H C / K B R ... S
Address  Disassembly                             Text string
00401000 MOV EAX,DWORD PTR SS:[ESP+8]            (Initial CPU selection)
00401031 PUSH Nag2.0040520C                      ASCII "Info"
00401036 PUSH Nag2.00405180                      ASCII "     KillNag     \n I did this one for your newbies who w
00401051 PUSH Nag2.00405178                      ASCII "Nag!"
00401056 PUSH Nag2.00405158                      ASCII "Oh, did u forget this one? :P"
0040107A PUSH Nag2.00405150                      ASCII "Nag!!"
0040107F PUSH Nag2.00405100                      ASCII "Oh, do u like this program? You are using the trial ver
004012CF PUSH Nag2.00405234                      ASCII "mscoree.dll"
004012DE PUSH Nag2.00405224                      ASCII "CorExitProcess"
004014F8 PUSH Nag2.00405634                      ASCII "<program name unknown>"
0040152B PUSH Nag2.00405630                      ASCII "..."
0040155F PUSH Nag2.00405614                      ASCII "Runtime Error!\n\nProgram: "
00401571 PUSH Nag2.00405610                      ASCII "\n\n"
0040158D PUSH Nag2.004055E8                      ASCII "Microsoft Visual C++ Runtime Library"
004020EC PUSH Nag2.004056D0                      ASCII "user32.dll"
00402107 PUSH Nag2.004056C4                      ASCII "MessageBoxA"
00402118 PUSH Nag2.004056B4                      ASCII "GetActiveWindow"
00402120 PUSH Nag2.004056A0                      ASCII "GetLastActivePopup"
0040213B PUSH Nag2.00405684                      ASCII "GetUserObjectInformationA"
0040214C PUSH Nag2.0040566C                      ASCII "GetProcessWindowStation"
00403D8E MOV EDI,Nag2.0040586C                   ASCII "Unknown security failure detected!"
00403D93 MOV DWORD PTR SS:[EBP-128],Nag2.0       ASCII "A security error of unknown cause has been detected whi
00403DA4 MOV EDI,Nag2.00405798                   ASCII "Buffer overrun detected!"
00403DA9 MOV DWORD PTR SS:[EBP-128],Nag2.0       ASCII "A buffer overrun has been detected which has corrupted
00403DD2 PUSH Nag2.00405634                      ASCII "<program name unknown>"
00403E13 PUSH Nag2.00405630                      ASCII "..."
00403E43 MOV EDI,Nag2.00405610                   ASCII "\n\n"
00403E4F PUSH Nag2.004056EC                      ASCII "Program: "
00403E79 PUSH Nag2.004055E8                      ASCII "Microsoft Visual C++ Runtime Library"
```

```
OllyDbg - Nag2.exe - [CPU - main thread, module Nag2]
C File View Debug Plugins Options Window Help Tools BreakPoint->
Paused                                                   L E M T W H C / K B R ... S
0040104F  .  6A 10          PUSH 10                                 Style = MB_OK|MB_ICONHAND|MB_APPLMODAL
00401051  .  68 78514000    PUSH Nag2.00405178                      Title = "Nag!"
00401056  .  68 58514000    PUSH Nag2.00405158                      Text = "Oh, did u forget this one? :P"
0040105B  .  56             PUSH ESI                                hOwner = FFFFFFFF
0040105C  .  FF15 C8504000  CALL DWORD PTR DS:[<&USER32.Me          MessageBoxA
00401062  .  6A 01          PUSH 1                                  Result = 1
00401064  .  56             PUSH ESI                                hWnd = FFFFFFFF
00401065  .  FF15 CC504000  CALL DWORD PTR DS:[<&USER32.En          EndDialog
0040106B  .  5E             POP ESI                                 kernel32.7C816D4F
0040106C  .  B8 01000000    MOV EAX,1
00401071  .  C2 1000        RETN 10
00401074  >  8B4C24 04      MOV ECX,DWORD PTR SS:[ESP+4]            ntdll.7C920738; Case 110 (WM_INITDIALOG)
00401078  .  6A 40          PUSH 40                                 Style = MB_OK|MB_ICONASTERISK|MB_APPLMOD
0040107A  .  68 50514000    PUSH Nag2.00405150                      Title = "Nag!!"
0040107F  .  68 00514000    PUSH Nag2.00405100                      Text = "Oh, do u like this program? You
00401084  .  51             PUSH ECX                                hOwner = 0013FFB0
00401085  .  FF15 C8504000  CALL DWORD PTR DS:[<&USER32.Me          MessageBoxA
0040108B  .  B8 01000000    MOV EAX,1
00401090  .  C2 1000        RETN 10
00401093  >  8B5424 04      MOV EDX,DWORD PTR SS:[ESP+4]            ntdll.7C920738; Case 10 (WM_CLOSE) of sw
00401097  .  6A 00          PUSH 0                                  Result = 0
00401099  .  52             PUSH EDX                                hWnd = 7C91EB94
0040109A  .  FF15 CC504000  CALL DWORD PTR DS:[<&USER32.En          EndDialog
004010A0  >  B8 01000000    MOV EAX,1                               Default case of switch 00401021
004010A5  .  C2 1000        RETN 10
004010A8     CC             INT3
004010A9     CC             INT3
004010AA     CC             INT3
004010AB     CC             INT3
004010AC     CC             INT3
004010AD     CC             INT3
004010AE     CC             INT3
004010AF     CC             INT3
004010B0  /$ 8B4424 04      MOV EAX,DWORD PTR SS:[ESP+4]            ntdll.7C920738
004010B4  |. 6A 00          PUSH 0                                  lParam = NULL
```

Lo primero que vemos es que el nag se encuentra dentro de un método autonomo. (Tiene una instrucción de RETN al principio y al final). Sabemos por lo tanto que es llamado desde algún lugar de la aplicación. Hacemos clic en la primera línea correspondiente a la dirección 401074 y vemos que se accede a ella a través de la instrucción JE en la dirección 401012.

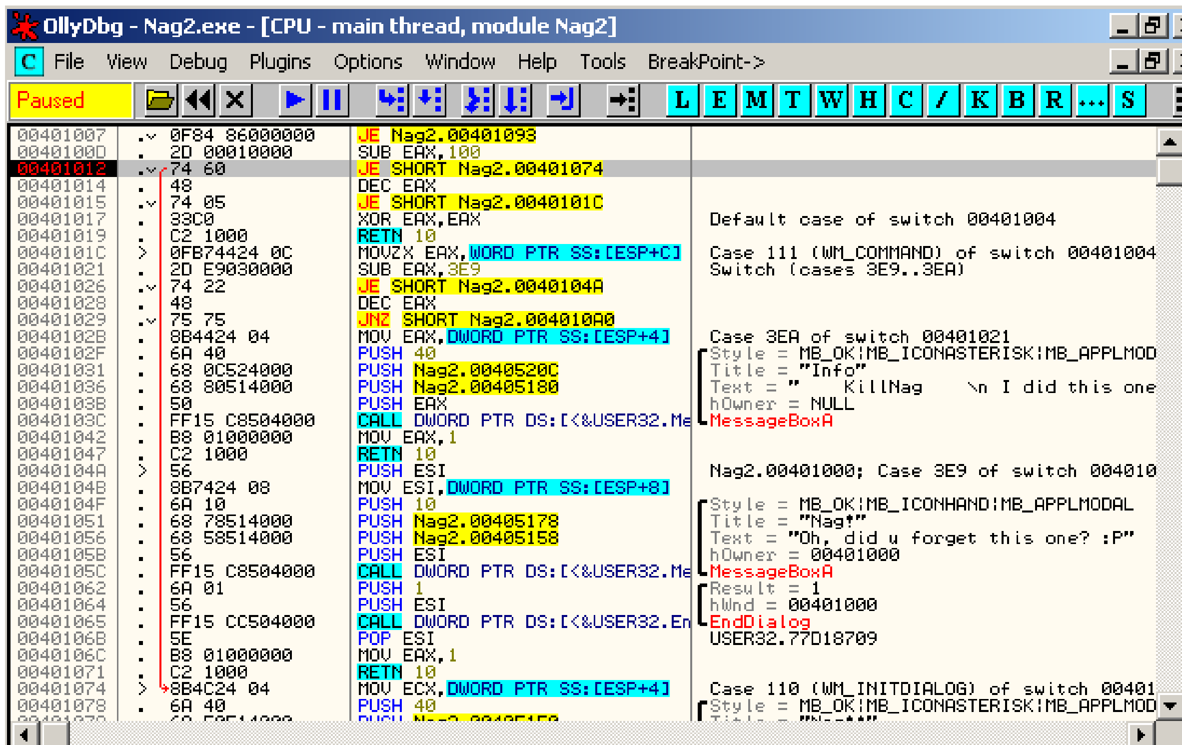


Ponemos un Breakpoint en la dirección 401012. Ejecutamos la aplicación y Olly se detiene en 401012.

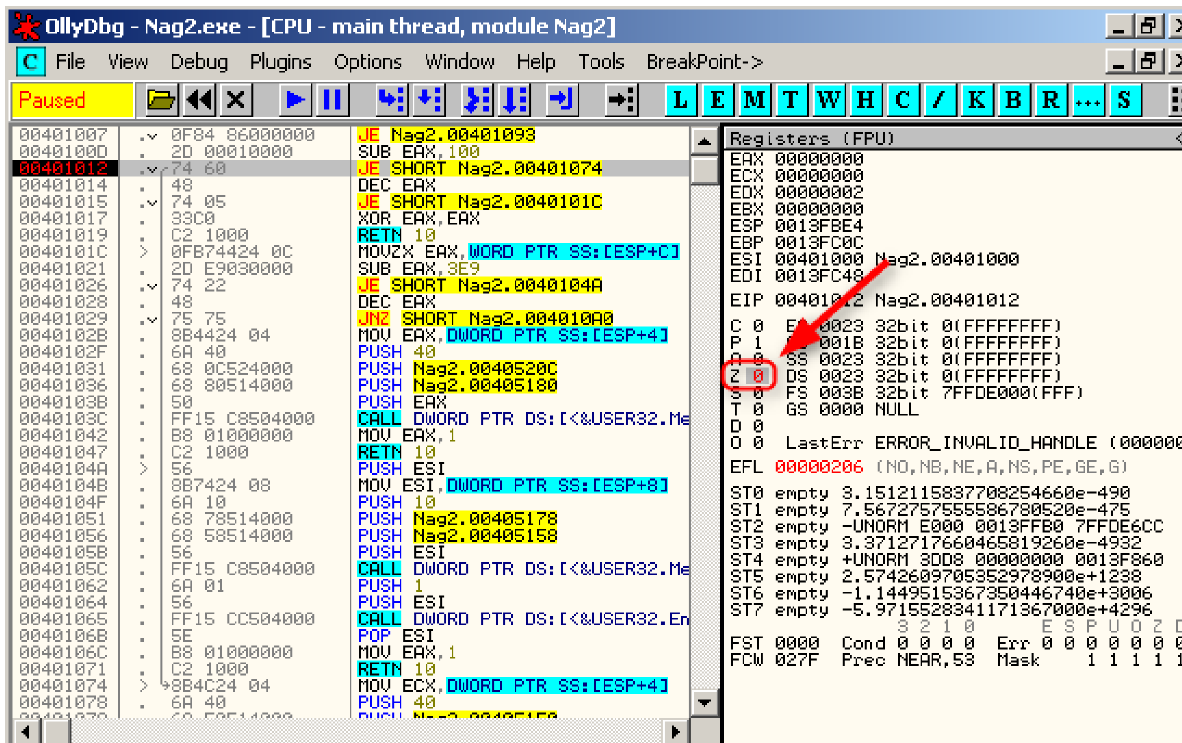


Vemos que nuestro nag no está siendo llamado. La razón de ello es que nuestro Breakpoint parece estar en la mitad del controlador de mensajes de Windows. Lo que significa que el primer mensaje que ha pasado por el controlador de mensajes no coincide con el mensaje que esperaba sobreescribir la aplicación para que salte el nag.

Pulsamos F9 para ejecutar la aplicación y nos detendremos en el mismo Breakpoint, pero esta vez si se va a tomar el salto lo que hará que aparezca nuestro nag en pantalla.



Cambiamos el valor de la bandera Z para decirle a Olly que no queremos saltar.

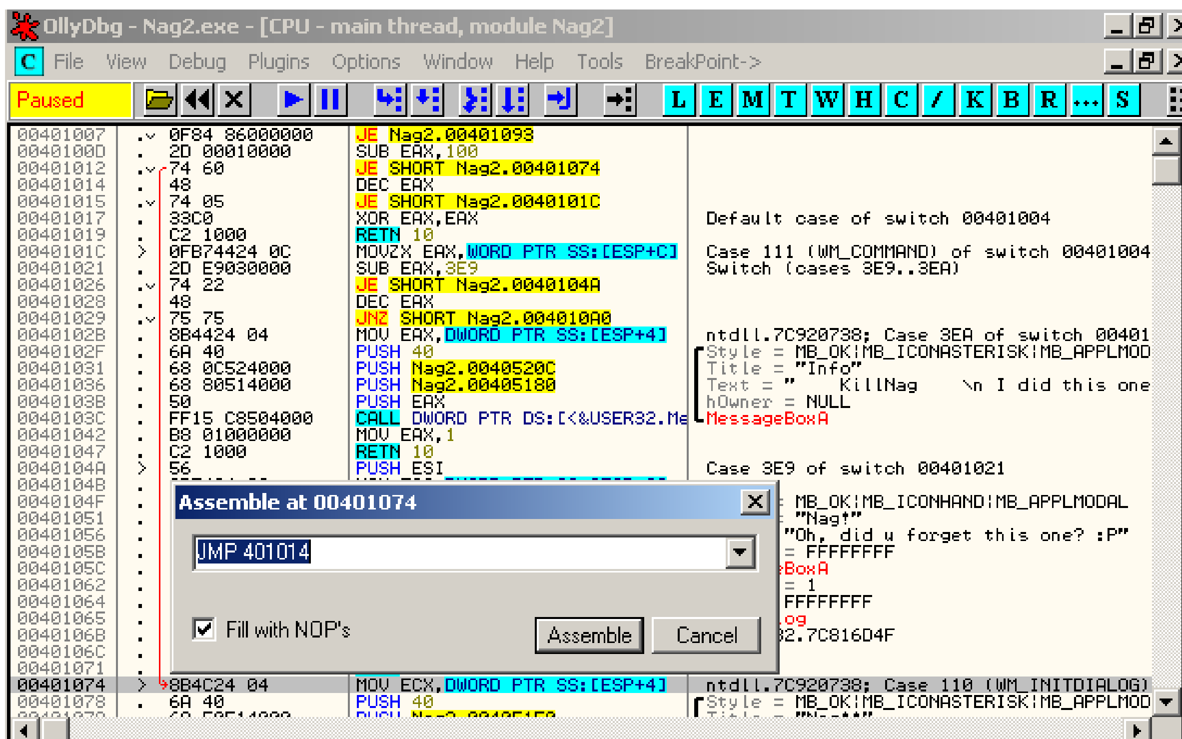


Si pulsamos F9 para seguir desplazandonos por la aplicación, van pasar 34 mensajes por el controlador de mensajes. Podemos o bien dejar el Breakpoint y pulsar F9 treintaycuatro veces, o podemos eliminar el Breakpoint y pulsar F9 solo una vez. En este caso el call no

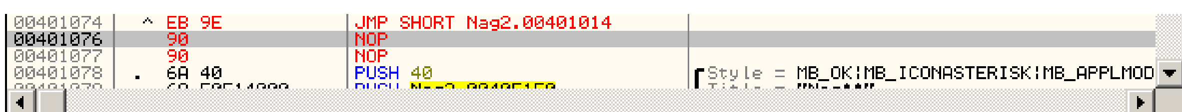
volverá a llamar a nuestro nag. Vamos a continuar siguiento la segunda opción y aparece la ventana principal (¡El nag de inicio ha desaparecido!).

El siguiente paso es parchear la aplicación. Normalmente lo hariamos cambiando la instrucción JE que hace el salto al nag, por la instrucción NOP, para que nunca salte. Pero veremos a continuación otra forma de conseguir el mismo resultado.

Sabemos que cuando el mensaje correcto pasa por el controlador de mensajes, (en este caso es el segundo mensaje) se va llamar el código de nuestro nag. ¿Qué pasaría si dejamos el salto al nag tal como está pero cambiando el nag para que vuelva a saltar hacia atras?

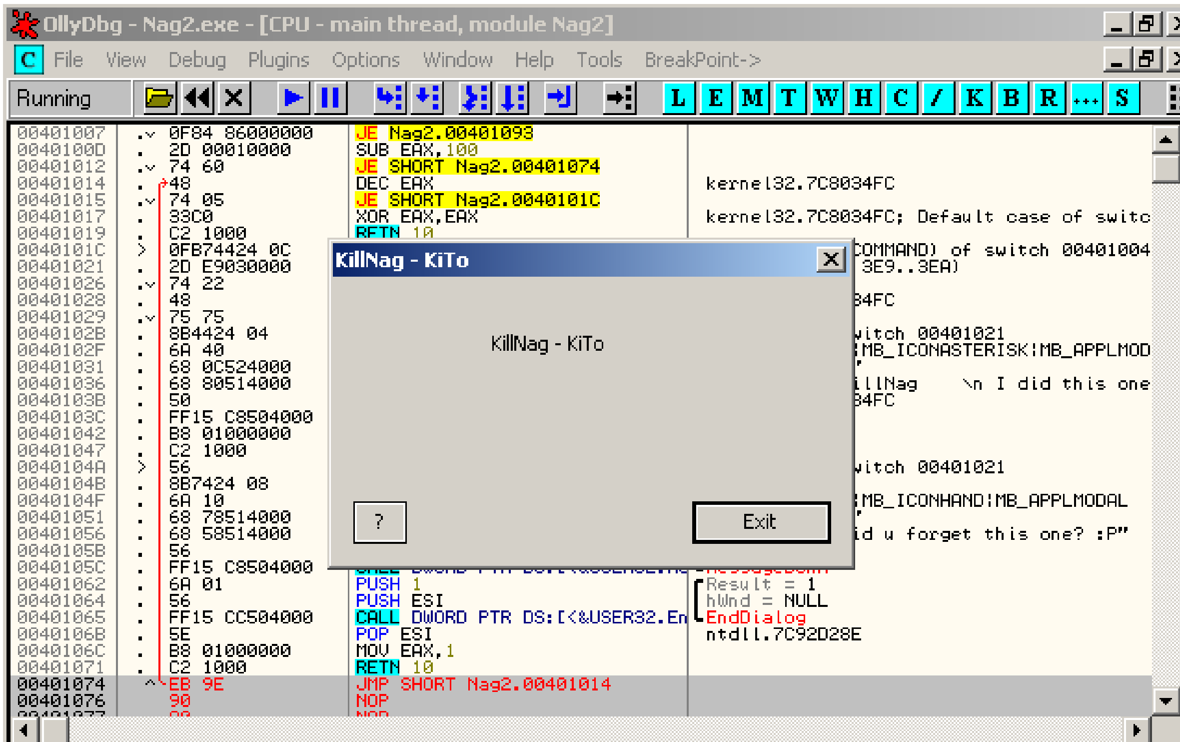


Vemos que el salto al nag corresponde con la dirección 401074. Cambiemos la instrucción en esa dirección para que vuelva saltar hacia atrás, justo después de la instrucción de salto JE, en 401014.

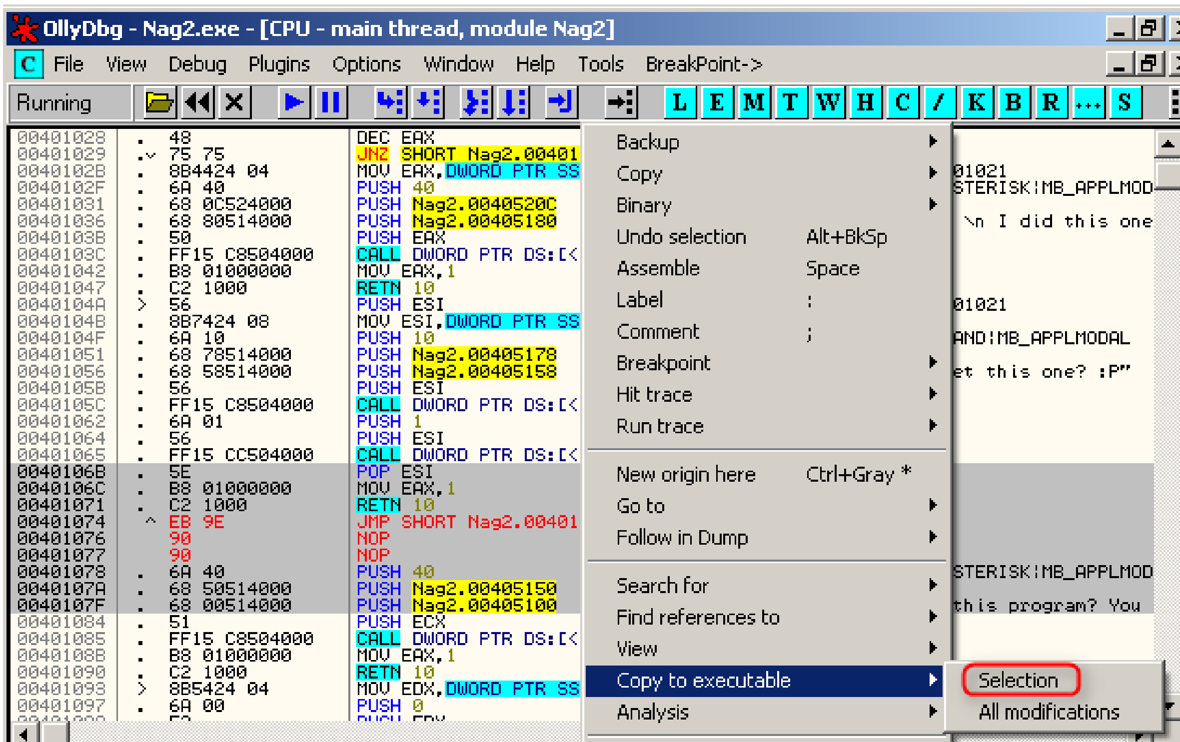


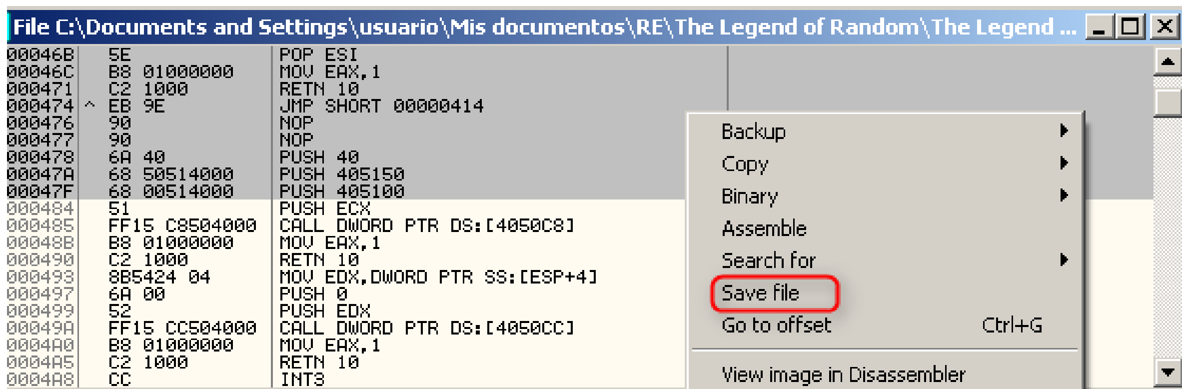
Realmente no exieste diferencia entre este método y convertir la instrucción JE en 401012 a una instrucción NOP. Lo relevante de este ejercicio es ver que siempre vamos a tener varias opciones para parchear nuestra aplicación. Además sustituir un CALL / JE por un NOP no siempre nos llevará al resultado esperado.

Pulsamos F9 y vemos que hemos omitido el primer nag.

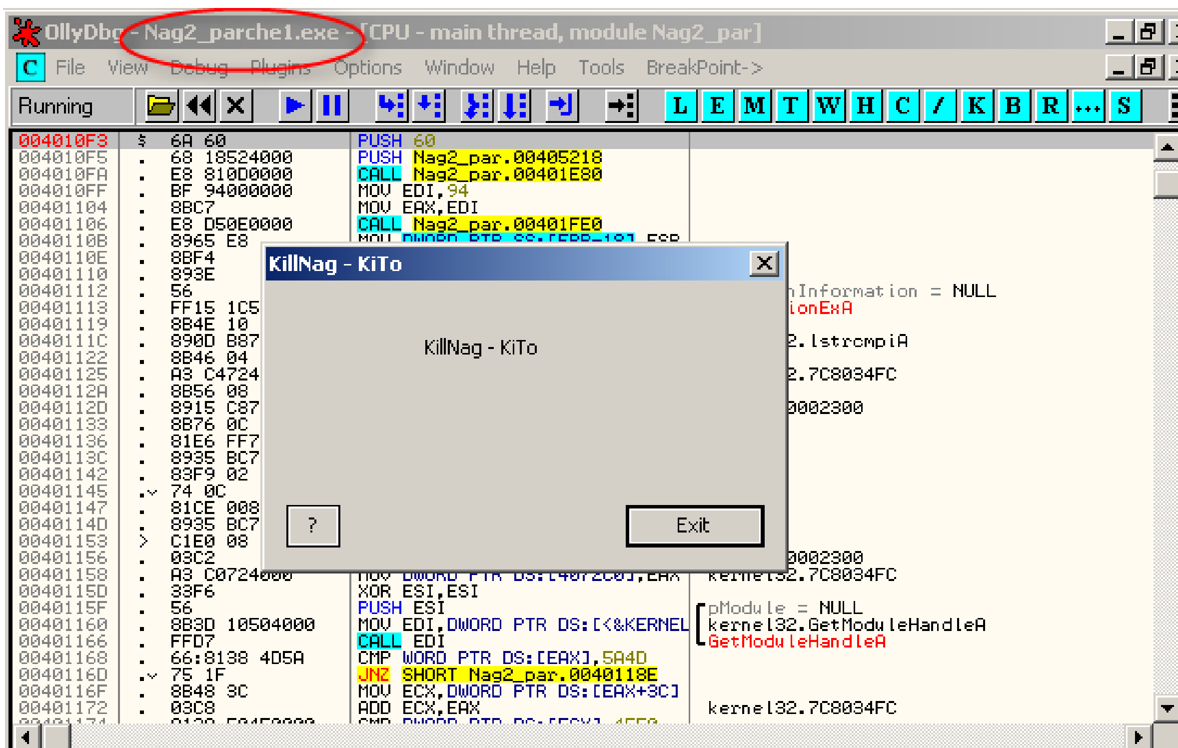


Guardamos la aplicación parcheada como nag2_parche1.exe.

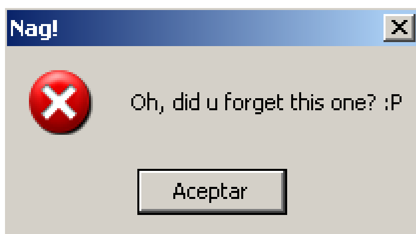




Cargamos esta nueva versión del programa parcheada en Olly y vemos que aparece la ventana principal.



Hacemos clic en “Exit” y se abre una nueva ventana:



Volvemos abrir la ventana de las cadenas de texto y buscamos la que aparece en la ventana anterior.

```
OllyDbg - Nag2_parche1.exe - [Text strings referenced in Nag2_par:.text]
R File View Debug Plugins Options Window Help Tools BreakPoint->
Paused                                     L E M T W H C / K B R ... S
Address  Disassembly                     Text string
00401031 PUSH Nag2_par.0040520C          ASCII "Info"
00401036 PUSH Nag2_par.00405180          ASCII "     KillNag     \n I did this one for your newbies who wants to practice more on r
00401051 PUSH Nag2_par.00405178          ASCII "Nag!"
00401056 PUSH Nag2_par.00405158          ASCII "Oh, did u forget this one? :P"
00401078 ASCII "j@hPQ@",0
0040107F PUSH Nag2_par.00405100          ASCII "Oh, do u like this program? You are using the trial version.. visit blablabla.."
004010F3 PUSH 60                         (Initial CPU selection)
004012CF PUSH Nag2_par.00405234          ASCII "mscoree.dll"
004012DE PUSH Nag2_par.00405224          ASCII "CorExitProcess"
004014F8 PUSH Nag2_par.00405634          ASCII "<program name unknown>"
0040152B PUSH Nag2_par.00405630          ASCII "..."
0040155F PUSH Nag2_par.00405614          ASCII "Runtime Error!\n\nProgram: "
00401571 PUSH Nag2_par.00405610          ASCII "\n\n"
0040158D PUSH Nag2_par.004055E8          ASCII "Microsoft Visual C++ Runtime Library"
004020EC PUSH Nag2_par.004056D0          ASCII "user32.dll"
00402107 PUSH Nag2_par.004056C4          ASCII "MessageBoxA"
00402118 PUSH Nag2_par.004056B4          ASCII "GetActiveWindow"
00402120 PUSH Nag2_par.004056A0          ASCII "GetLastActivePopup"
0040213B PUSH Nag2_par.00405684          ASCII "GetUserObjectInformationA"
0040214C PUSH Nag2_par.0040566C          ASCII "GetProcessWindowStation"
00403D8E MOV EDI,Nag2_par.0040586C       ASCII "Unknown security failure detected!"
00403D93 MOV DWORD PTR SS:[EBP-128],Nag2_p ASCII "A security error of unknown cause has been detected which has\ncorrupted the prog
00403DA4 MOV EDI,Nag2_par.00405798       ASCII "Buffer overrun detected!"
00403DA9 MOV DWORD PTR SS:[EBP-128],Nag2_p ASCII "A buffer overrun has been detected which has corrupted the program's\ninternal st
00403DD2 PUSH Nag2_par.00405634          ASCII "<program name unknown>"
00403E13 PUSH Nag2_par.00405630          ASCII "..."
00403E43 MOV EDI,Nag2_par.00405610       ASCII "\n\n"
00403E4F PUSH Nag2_par.004056EC          ASCII "Program: "
00403E79 PUSH Nag2_par.004055E8          ASCII "Microsoft Visual C++ Runtime Library"
```

Hacemos doble clic sobre la línea.

```
OllyDbg - Nag2_parche1.exe - [CPU - main thread, module Nag2_par]
C File View Debug Plugins Options Window Help Tools BreakPoint->
Paused                                     L E M T W H C / K B R ... S
00401000  .  8B4424 08      MOV EAX,DWORD PTR SS:[ESP+8]
00401004  .  83E8 10        SUB EAX,10                        Switch (cases 10..110)
00401007  .v 0F84 86000000  JE Nag2_par.00401093
0040100D  .  2D 00010000    SUB EAX,100
00401012  .v 74 60          JE SHORT Nag2_par.00401074
00401014  >  48             DEC EAX                           Default case of switch 00401004
00401015  .v 74 05          JE SHORT Nag2_par.0040101C
00401017  .  33C0           XOR EAX,EAX
00401019  .  C2 1000        RETN 10
0040101C  >  0FB74424 0C    MOVZX EAX,WORD PTR SS:[ESP+C]
00401021  .  2D E9030000    SUB EAX,3E9                       Switch (cases 3E9..3EA)
00401026  .v 74 22          JE SHORT Nag2_par.0040104A
00401028  .  48             DEC EAX
00401029  .v 75 75          JNZ SHORT Nag2_par.004010A0
0040102B  .  8B4424 04      MOV EAX,DWORD PTR SS:[ESP+4]      ntdll.7C920738; Case 3EA of switch 00401021
0040102F  .  6A 40          PUSH 40                           Style = MB_OK|MB_ICONASTERISK|MB_APPLMODAL
00401031  .  68 0C524000    PUSH Nag2_par.0040520C            Title = "Info"
00401036  .  68 80514000    PUSH Nag2_par.00405180            Text = "     KillNag     \n I did this one for your newbies who wants
0040103B  .  50             PUSH EAX                          hOwner = NULL
0040103C  .  FF15 C8504000  CALL DWORD PTR DS:[<&USER32.Me    MessageBoxA
00401042  .  B8 01000000    MOV EAX,1
00401047  .  C2 1000        RETN 10
0040104A  >  56             PUSH ESI                          Case 3E9 of switch 00401021
0040104B  .  8B7424 08      MOV ESI,DWORD PTR SS:[ESP+8]
0040104F  .  6A 10          PUSH 10                           Style = MB_OK|MB_ICONHAND|MB_APPLMODAL
00401051  .  68 78514000    PUSH Nag2_par.00405178            Title = "Nag!"
00401056  .  68 58514000    PUSH Nag2_par.00405158            Text = "Oh, did u forget this one? :P"
0040105B  .  56             PUSH ESI                          hOwner = FFFFFFFF
0040105C  .  FF15 C8504000  CALL DWORD PTR DS:[<&USER32.Me    MessageBoxA
00401062  .  6A 01          PUSH 1                            Result = 1
00401064  .  56             PUSH ESI                          hWnd = FFFFFFFF
00401065  .  FF15 CC504000  CALL DWORD PTR DS:[<&USER32.En    EndDialog
0040106B  .  5E             POP ESI                           kernel32.7C816D4F
0040106C  .  B8 01000000    MOV EAX,1
00401071  .  C2 1000        RETN 10
00401074  >^ EB 9E          JMP SHORT Nag2_par.00401014       Case 110 of switch 00401004
00401076     90             NOP
00401077     90             NOP
00401078  .  6A 40 68 50 51 4 ASCII "j@hPQ@",0
0040107F  .  68 00514000    PUSH Nag2_par.00405100            Text = "Oh, do u like this program? You are using the trial versior
00401084  .  51             PUSH ECX                          hOwner = 0013FFB0
00401085  .  FF15 C8504000  CALL DWORD PTR DS:[<&USER32.Me    MessageBoxA
0040108B  .  B8 01000000    MOV EAX,1
00401090  .  C2 1000        RETN 10
00401093  >  8B5424 04      MOV EDX,DWORD PTR SS:[ESP+4]      ntdll.7C920738; Case 10 of switch 00401004
00401097  .  6A 00          PUSH 0                            Result = 0
00401099  .  52             PUSH EDX                          hWnd = 7C91EB94
0040109A  .  FF15 CC504000  CALL DWORD PTR DS:[<&USER32.En    EndDialog
```

Y nos situamos sobre la primera línea del método que nos lleva al origen de la llamada para este segundo nag. Vemos que la llamada se encuentra justo después de la llamada al primer nag.

Nuestra primera idea podría llevarnos a poner una instrucción de salto para volver hacia atrás como se hizo con el primer nag. Pero si nos fijamos bien, podemos observar que cuando se llama al segundo nag inmediatamente después se llama a la API EndDialog. Asi que saltando hacia atrás no va a funcionar ya que el dialogo nunca se cerraría.

Lo siguiente que podríamos pensar es cambiar la instrucción JE en 401026 para que salte a EndDialog, sin pasar por el segundo nag.

Cambiemos pues la instrucción JE en 401026 para que salte a la primera línea de EndDialog en 401062.

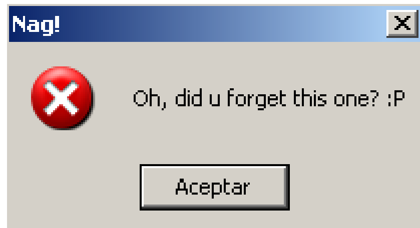
```
00401026  .v 74 22          JE SHORT Nag2_par.0040104A
00401028  .  48             DEC EAX
00401029  .v 75 75          JNZ SHORT Nag2_par.004010A0
0040102B  .  8B4424 04      MOV EAX,DWORD PTR SS:[ESP+4]
0040102F  .  6A 40          PUSH 40                          ntdll.7C920738; Case 3EA of switch 00401021
00401031  .  68 0C524000    PUSH Nag2_par.0040520C           Style = MB_OK|MB_ICONASTERISK|MB_APPLMODAL
00401036  .  68 80514000    PUSH Nag2_par.00405180           Title = "Info"
0040103B  .  50             PUSH EAX                         Text = "     KillNag     \n I did this one for your newbies who wants
0040103C  .  FF15 C8504000  CALL DWORD PTR DS:[<&USER32.Me   hOwner = NULL
00401042  .  B8 01000000    MOV EAX,1                        MessageBoxA
00401047  .  C2 1000        RETN 10
0040104A  >  56             PUSH ESI                         Case 3E9 of switch 00401021
0040104B  .  8B7424 08      MOV ESI,DWORD PTR SS:[ESP+8]
0040104F  .  6A 10          PUSH 10                          Style = MB_OK|MB_ICONHAND|MB_APPLMODAL
00401051  .  68 78514000    PUSH Nag2_par.00405178           Title = "Nag!"
00401056  .  68 58514000    PUSH Nag2_par.00405158           Text = "Oh, did u forget this one? :P"
0040105B  .  56             PUSH ESI                         hOwner = FFFFFFFF
0040105C  .  FF15 C8504000  CALL DWORD PTR DS:[<&USER32.Me   MessageBoxA
00401062  .  6A 01          PUSH 1                           Result = 1
                                                             hWnd = FFFFFFFF
Assemble at 00401026                                         EndDialog
JE SHORT 00401062                                            kernel32.7C816D4F
[x] Fill with NOP's   [Assemble] [Cancel]                    Case 110 of switch 00401004
                                                             Text = "Oh, do u like this program? You are using the trial version
                                                             hOwner = 0013FFB0
```

Pulsamos F9.

Esto no es lo que esperabamos. Parace que hicimos algo mal. Esto es lo que haremos: ejecutaremos la aplicación sin el parche para ver como se comporta, después ejecutaremos la aplicación con el parche y miramos en que se diferencian.

Reiniciamos la aplicación, hacemos clic en Exit y nos detenemos en nuestro parche.

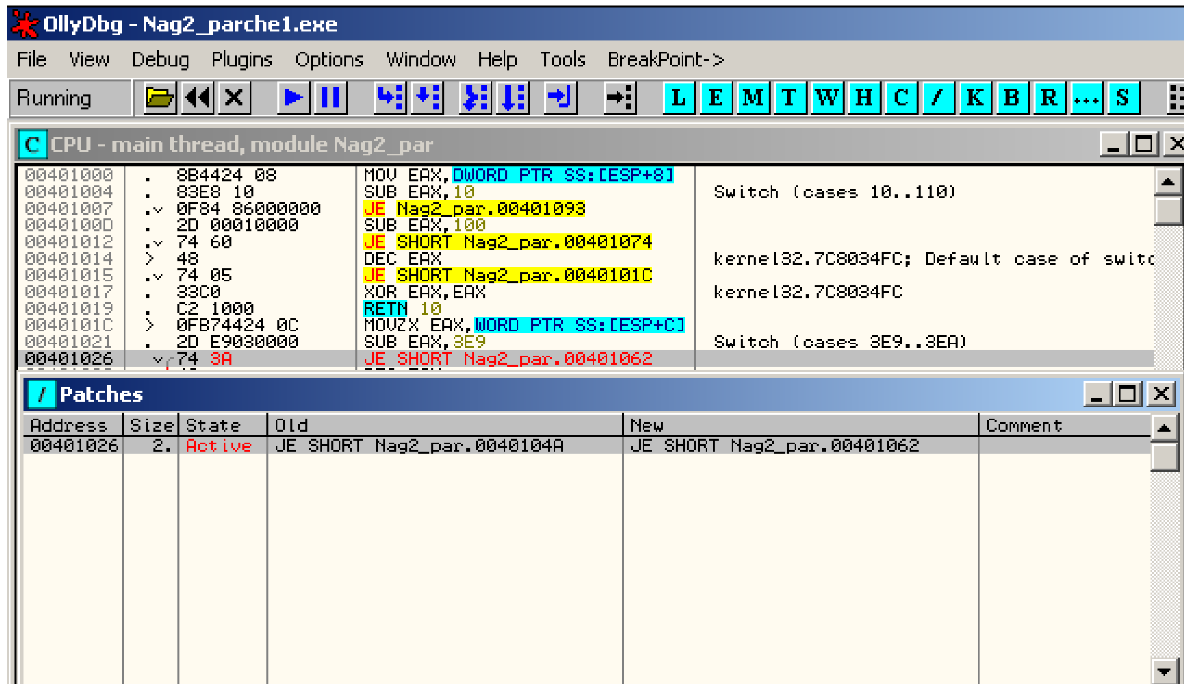
Pulsamso F8 hasta llegar a MessageBoxA. Aparece nuestro nag.



Continuamos pulsando F8 hasta llegar a EndDialog. En la ventana de la pila podemos ver las cuatro siguientes instrucciones:

- Un controlador a nuestra ventana.
- El resultado de EndDialog.
- Un indicador hacia la primera linea de nuestro código (401000)
- Una dirección de retorno a USER32.

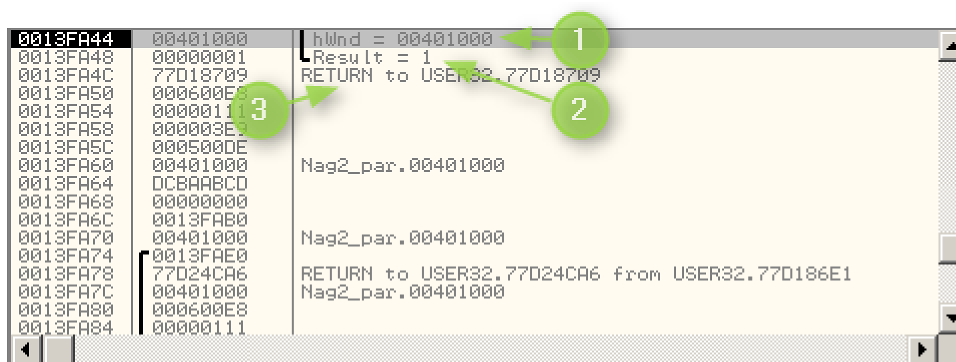
Reiniciamos nuestra aplicación. Nos detenemos en nuestro parche. Lo activamos y vemos que saltamos por encima de la llamada a nuestro Nag hasta llegar al CALL EndDialog:



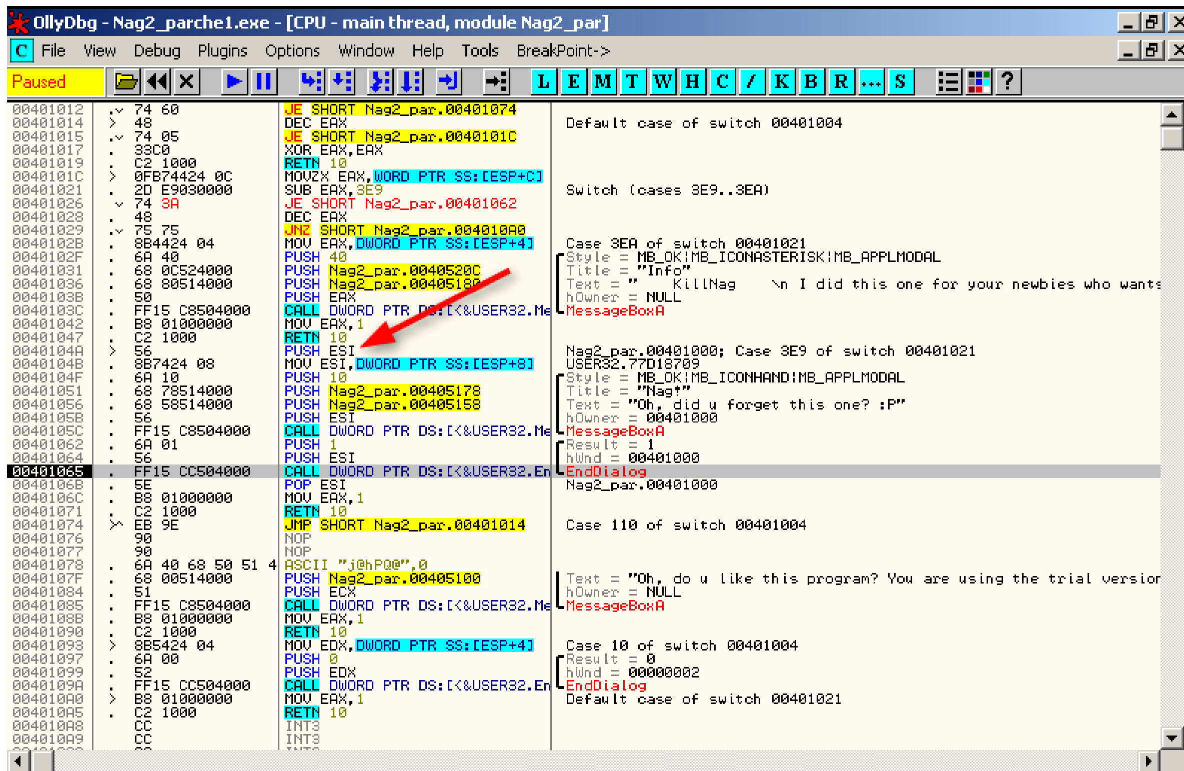
En la ventana de la pila podemos observar las tres siguientes instrucciones:

- El controlador de ventanas
- El resultado del código.
- Y el regreso a USER32.

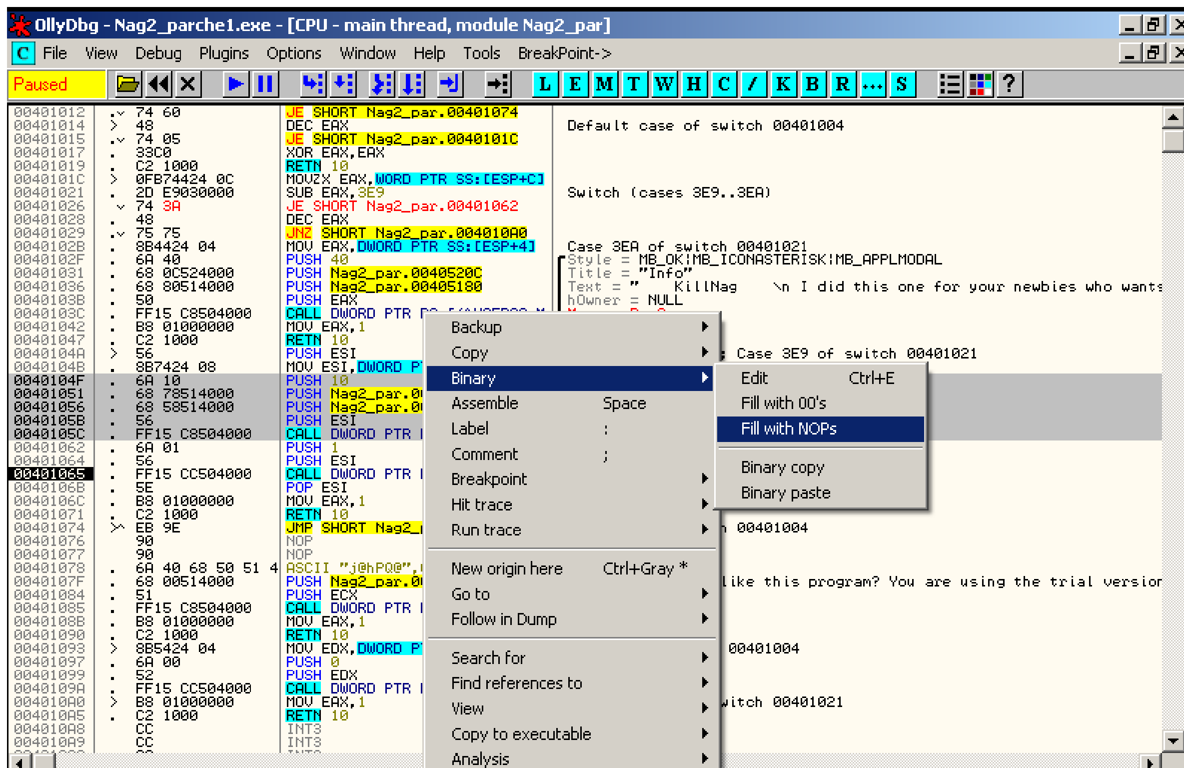
<u>¡FALTA EL INDICADOR HACIA LA PRIMERA LINEA DE NUESTRO CODIGO (401000)!</u>

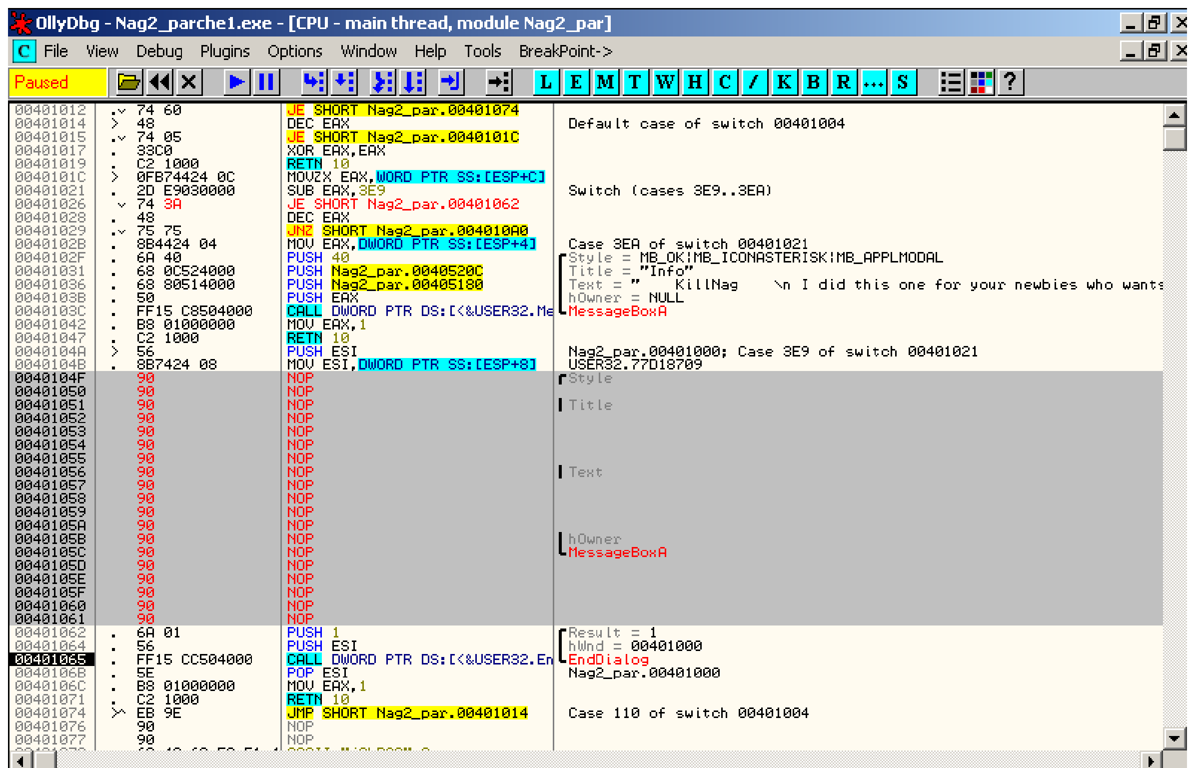


Si nos fijamos en el segundo nag vemos que antes de crear el cuadro de mensaje, ESI es empujado a la pila, que es precisamente el puntero de nuestro código. Este PUSH también se podría haber hecho después de la llamada al cuadro de mensaje.



El problema es que tenemos un código de inizialicación que necesitamos, una llamada al nag que no queremos y una llamada a EndDialog que sí queremos. Borremos pues el código que no queremos. Seleccionamos el código desde 40104F a 40105C, hacemos clic con el botón derecho y seleccionamos “Binary” -> “Fill with NOPs”:



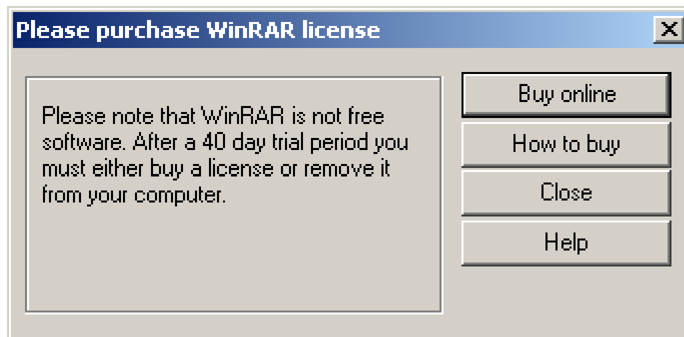


Guardamos el parche como nag2_parche2.exe. Cerramos Olly y ejecutamos el nuevo parche. Vemos que después de hacer clic en “Exit” ya no sale ningúna ventana molesta.

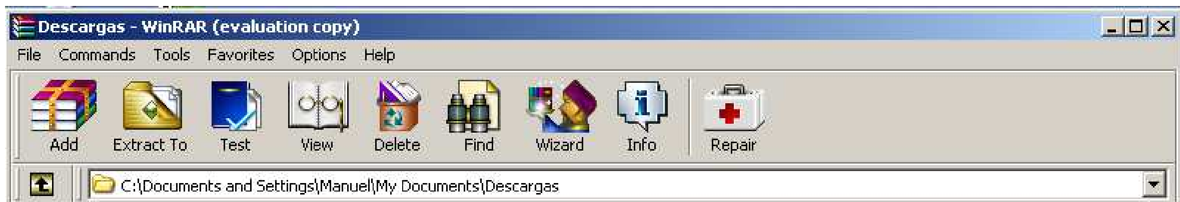
7.12 Caso práctico 12: Usando el call stack

En el siguiente ejercicio utilizaremos los conocimientos adquiridos en el capítulo anterior para eliminir el nag de un programa real. El nag aparece al cabo de 40 días por lo que tenemos dos opciónes para seguir el ejercicio: a) Esperar 40 días y 40 noches o b) Añadir a la fecha del ordenador 41 dias.

Supongamos que hemos optado por una u otra opción, el siguiente paso es ejecutar el programa haciendo doble clic sobre ejecutable. Al cabo de unos segundos aparece el nag.



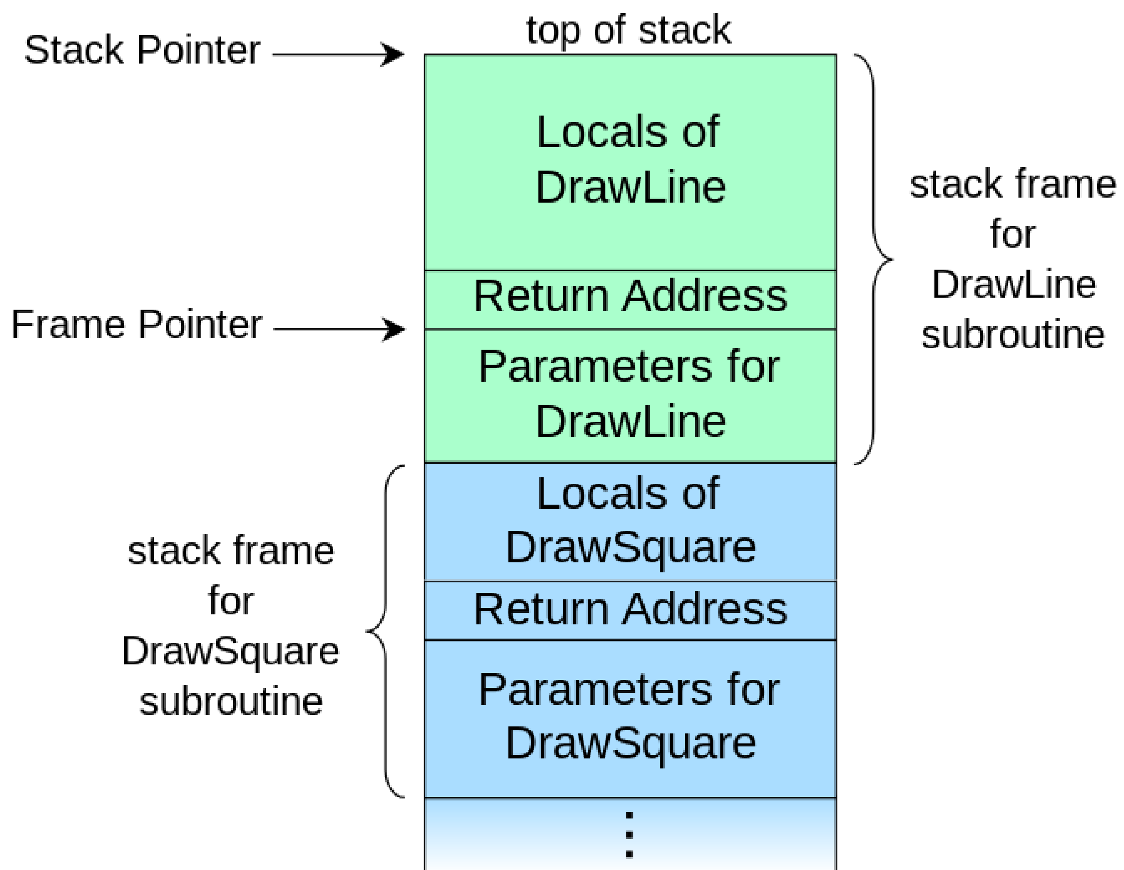
Fijémonos también en el título de la parte superior de la ventana donde pone bien claro que se trata de una "copia de evaluación".



Existen dos caminos para llegar al área del código que nos interesa. Antes de nada, cargamos la aplicación en Olly y esperamos a que aparezca el nag. Podríamos empezar por buscar cadenas de texto o "intermodular calls", pero esta opción no es válida para la mayoría de programas disponibles en la red. Así que aprenderemos a usar una técnica nueva: El Call Stack.

En ciencias de la computación, *una* ***pila de llamadas*** *(en inglés call stack) es una estructura dinámica de datos* LIFO, *(una* pila*), que almacena la información sobre las subrutinas activas de un* programa de computadora. *Esta clase de pila también es conocida como una* ***pila de ejecución***, ***pila de control***, ***pila de función***, *o* ***pila de tiempo de ejecución***, *y a menudo se describe en forma corta como "la pila".*

Una pila de llamadas es de uso frecuente para varios propósitos relacionados, pero la principal razón de su uso, es seguir el curso del punto al cual cada subrutina activa debe retornar el control cuando termine de ejecutar. (Las subrutinas activas son las que se han llamado pero todavía no han completado su ejecución ni retornando al lugar siguiente desde donde han sido llamadas). Si, por ejemplo, una subrutina DibujaCuadrado llama a una subrutina DibujaLinea *desde cuatro lugares diferentes, el código de* DibujaLinea *debe tener una manera de saber a dónde retornar. Esto es típicamente hecho por un código que, para cada llamada dentro de DibujaCuadrado, pone la* dirección *de la* instrucción *después de la* sentencia *de llamada particular (la "dirección de retorno") en la pila de llamadas.*

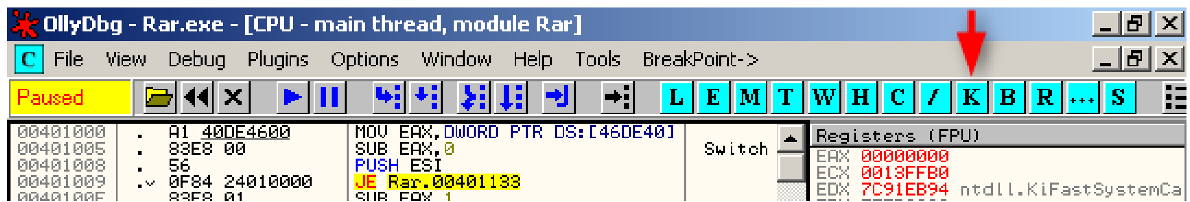


Fuente: https://es.wikipedia.org/wiki/Pila_de_llamadas

En la figura se ve una pila, creciendo de abajo hacia arriba. La subrutina DrawSquare es llamada y se crea un stack frame para ella (en azul). Luego, DrawSquare llama a la subrutina DrawLine, la cual tiene su propio stack frame (en verde).

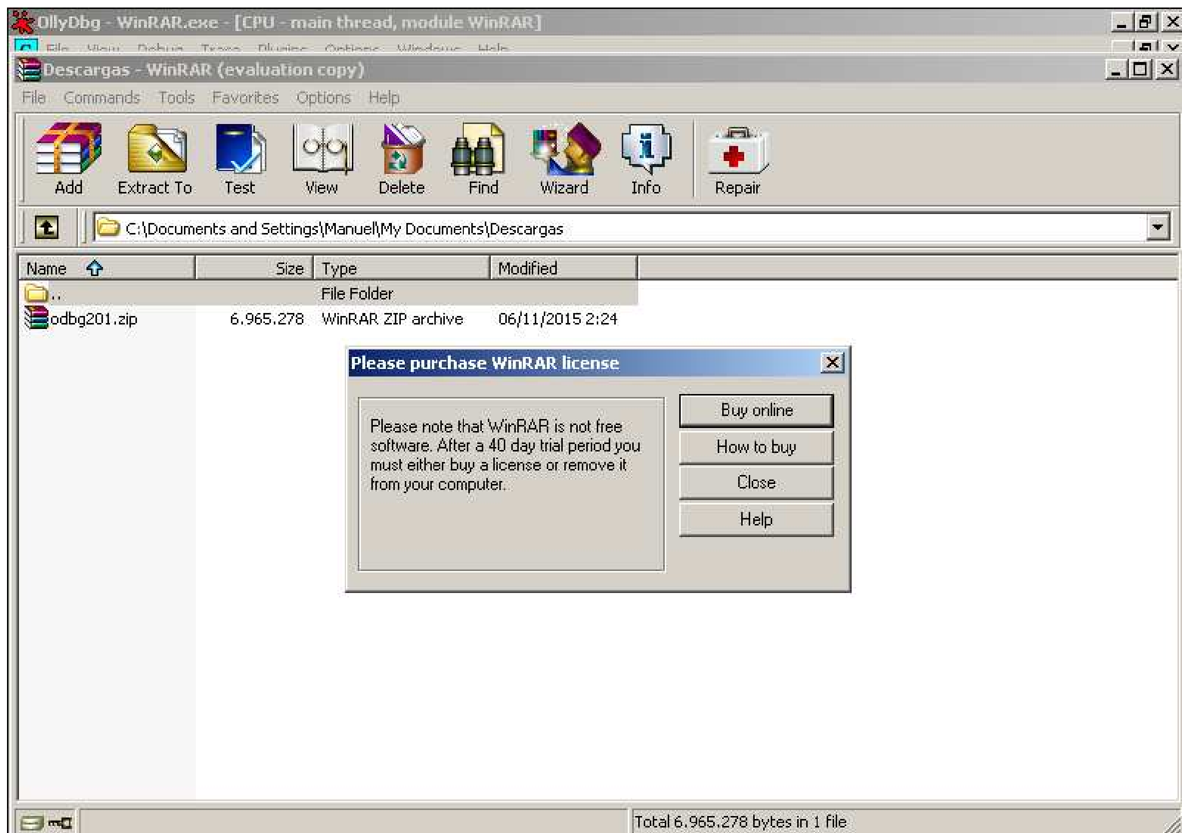
*El stack frame de cada subrutina tiene, en este caso, tres partes: * una dirección de retorno que indica la siguiente dirección a ejecutar después de que termine la subrutina, * los parámetros con que fue llamada la subrutina (que se cargan antes de llamarla), * y un espacio reservado para las variables y las constantes locales de la subrutina.*

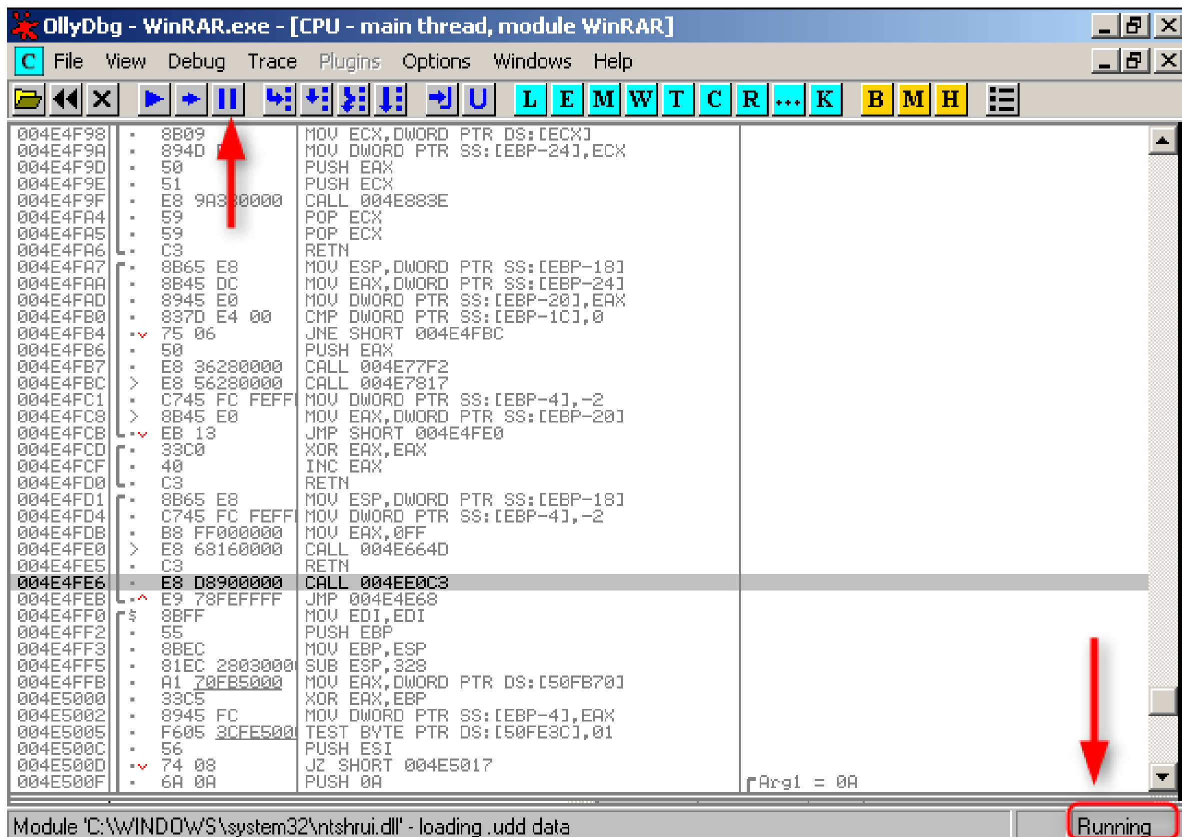
Para visualizar el call stack en Olly basta con pulsar sobre el botón “K”.



Nota: *La opción "call stack" no funciona en la versión 1.0 de Olly para Windows 7de 64-bit. Para solucionar el problema veremos los ejemplos con la versión 2.0 de Olly.*

Cargamos la aplicación en Olly y pulsamos F9. Esperamos a que aparezca el nag. Una vez que aparezca (y antes de cerrarlo) hacemos clic en Olly para activar la ventana y poder detener la aplicación:





Pulsamos el botón del call stack (‘K’) y aparece la siguiente ventana.

Call stack of main thread

Stack	Data	Procedure	Called from	Frame
0012C15C	77D493F5	ntdll.KiFastSystemCall	USER32.WaitMessage+0A	0012C190
0012C160	77D6EA24	USER32.WaitMessage	USER32.77D6EA1F	
0012C194	77D5688A	USER32.77D6E895	USER32.77D56885	
0012C1BC	77D568CC	USER32.77D567D4	USER32.DialogBoxIndirectPar	0012C1B8
0012C1DC	77D5892D	USER32.DialogBoxIndirectParamAorW	USER32.DialogBoxParamA+47	0012C1D8
0012C208	0043F81D	USER32.DialogBoxParamA	WinRAR.0043F818	0012C204
0012E5EC	77D48709	WinRAR.0043F070	USER32.77D48706	0012E5E8
0012E618	77D487EB	USER32.77D486E1	USER32.77D487E6	0012E614
0012E680	77D489A5	USER32.77D48734	USER32.77D489A0	0012E67C
0012E6E0	77D4BCCC	USER32.77D488C9	USER32.DispatchMessageA+0A	0012E6DC
0012E6F0	00442F23	USER32.DispatchMessageA	WinRAR.00442F1E	0012E6EC
0012E724	00442B34	WinRAR.00442C44	WinRAR.00442B2F	
0012FF8C	0049F24B	WinRAR.00441E7C	WinRAR.0049F248	
0012FFC4	7C816D4F	???	kernel32.7C816D4C	0012FFB8

En este ejemplo podemos ver una función ntdll, algunas funciones USER32, una llamada a DialogBoxParamA, más llamadas a USER32 y al final un call de nuestra aplicación WinRAR. Veamos como interpretar todo esto: WinRAR, desde la dirección 442C44, llama a DispatchMessageA, en este caso con un mensaje para mostrar un cuadro de dialogo. A continuación USER32 llama a la función DialogBoxParamA para mostrar el cuadro de dialogo con el texto "evaluation copy" en forma de título. USER32 muestra este cuadro de dialogo y espera algún input utilizando para ello WaitMessage.

Lo importante para nosotros es el call al cuadro de dialogo y el call de la aplicación:

Call stack of main thread

Stack	Data	Procedure	Called from	Frame
0012C15C	77D493F5	ntdll.KiFastSystemCall	USER32.WaitMessage+0A	0012C190
0012C160	77D6EA24	USER32.WaitMessage	USER32.77D6EA1F	
0012C194	77D5688A	USER32.77D6E895	USER32.77D56885	
0012C1BC	77D568CC	USER32.77D567D4	USER32.DialogBoxIndirectPar	0012C1B8
0012C1DC	77D5892D	USER32.DialogBoxIndirectParamAorW	USER32.DialogBoxParamA+47	0012C1D8
0012C208	0043F81D	USER32.DialogBoxParamA	WinRAR.0043F818	0012C204
0012E5EC	77D48709	WinRAR.0043F070	USER32.77D48706	0012E5E8
0012E618	77D487EB	USER32.77D486E1	USER32.77D487E6	0012E614
0012E680	77D489A5	USER32.77D48734	USER32.77D489A0	0012E67C
0012E6E0	77D4BCCC	USER32.77D488C9	USER32.DispatchMessageA+0A	0012E6DC
0012E6F0	00442F23	USER32.DispatchMessageA	WinRAR.00442F1E	0012E6EC
0012E724	00442B34	WinRAR.00442C44	WinRAR.00442B2F	
0012FF8C	0049F24B	WinRAR.00441E7C	WinRAR.0049F248	
0012FFC4	7C816D4F	???	kernel32.7C816D4C	0012FFB8

Hacemos doble clic sobre el call a DialogBoxParamA:

Lo primero que haremos es poner un Breakpoint al comienzo de esta rutina en 43F7F9. Más arriba podemos ver algunos saltos condicionales. Si subimos más vemos comentarios de tipo “Case XX (WM_XXXXXX) of switch 43F0A4”.

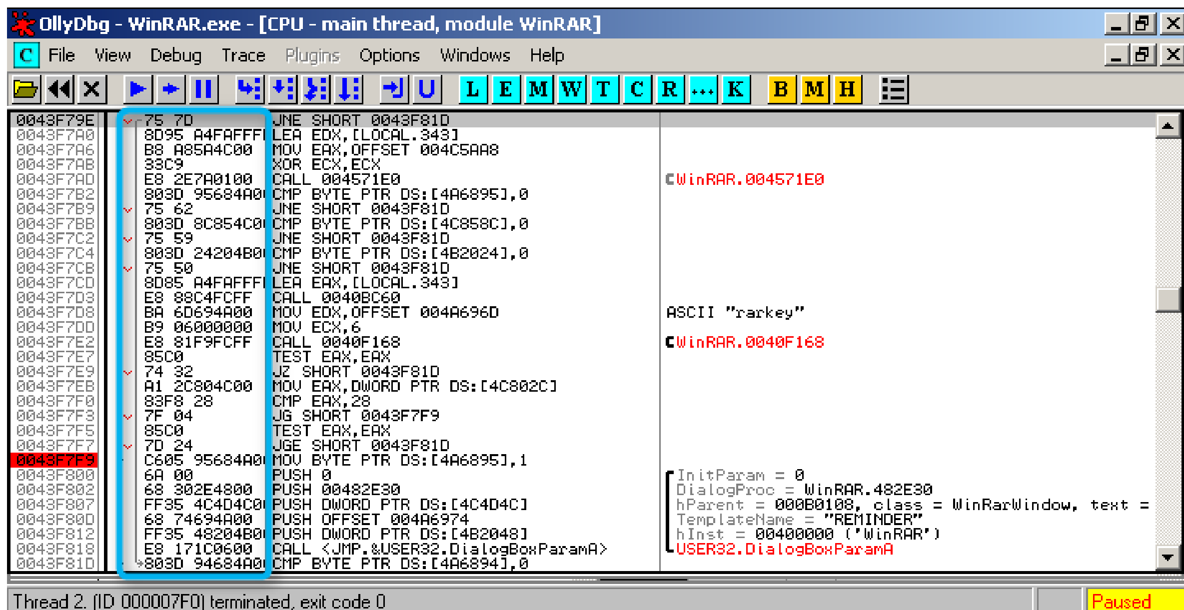
Si seguimos subiendo veremos que se trata de un switch realmente largo, donde WM_XXXXX representan mensajes de Windows. En ejercicios futuros estudiaremos más en detalle los procedimientos de los mensajes de Windows. Por ahora solo nos interesa el código que realiza la llamada al cuadro de dialogo, incluido el mensaje de Windows WM_TIMER. A continuación podemos ver el código completo:

```
0043F797  >  833D F0764C0( CMP DWORD PTR DS:[4C76F0],0      Case 113 (WM_TIMER) of switch WinRAR.43F0A4
0043F79E  .v 75 7D        JNE SHORT 0043F81D
0043F7A0  .  8D95 A4FAFFF( LEA EDX,[LOCAL.343]
0043F7A6  .  B8 A85A4C00  MOV EAX,OFFSET 004C5AA8
0043F7AB  .  33C9         XOR ECX,ECX
0043F7AD  .  E8 2E7A0100  CALL 004571E0                   WinRAR.004571E0
0043F7B2  .  803D 95684A0( CMP BYTE PTR DS:[4A6895],0
0043F7B9  .v 75 62        JNE SHORT 0043F81D
0043F7BB  .  803D 8C854C0( CMP BYTE PTR DS:[4C858C],0
0043F7C2  .v 75 59        JNE SHORT 0043F81D
0043F7C4  .  803D 24204B0( CMP BYTE PTR DS:[4B2024],0
0043F7CB  .v 75 50        JNE SHORT 0043F81D
0043F7CD  .  8D85 A4FAFFF( LEA EAX,[LOCAL.343]
0043F7D3  .  E8 88C4FCFF  CALL 0040BC60
0043F7D8  .  BA 6D694A00  MOV EDX,OFFSET 004A696D         ASCII "rarkey"
0043F7DD  .  B9 06000000  MOV ECX,6
0043F7E2  .  E8 81F9FCFF  CALL 0040F168                   WinRAR.0040F168
0043F7E7  .  85C0         TEST EAX,EAX
0043F7E9  .v 74 32        JZ SHORT 0043F81D
0043F7EB  .  A1 2C804C00  MOV EAX,DWORD PTR DS:[4C802C]
0043F7F0  .  83F8 28      CMP EAX,28
0043F7F3  .v 7F 04        JG SHORT 0043F7F9
0043F7F5  .  85C0         TEST EAX,EAX
0043F7F7  .v 7D 24        JGE SHORT 0043F81D
0043F7F9  >  C605 95684A0( MOV BYTE PTR DS:[4A6895],1
0043F800  .  6A 00        PUSH 0                          InitParam = 0
0043F802  .  68 302E4800  PUSH 00482E30                   DialogProc = WinRAR.482E30
0043F807  .  FF35 4C4D4C0( PUSH DWORD PTR DS:[4C4D4C]     hParent = 000B0108, class = WinRarWindow, text =
0043F80D  .  68 74694A00  PUSH OFFSET 004A6974            TemplateName = "REMINDER"
0043F812  .  FF35 48204B0( PUSH DWORD PTR DS:[4B2048]     hInst = 00400000 ('WinRAR')
0043F818  .  E8 171C0600  CALL <JMP.&USER32.DialogBoxParamA>  USER32.DialogBoxParamA
```

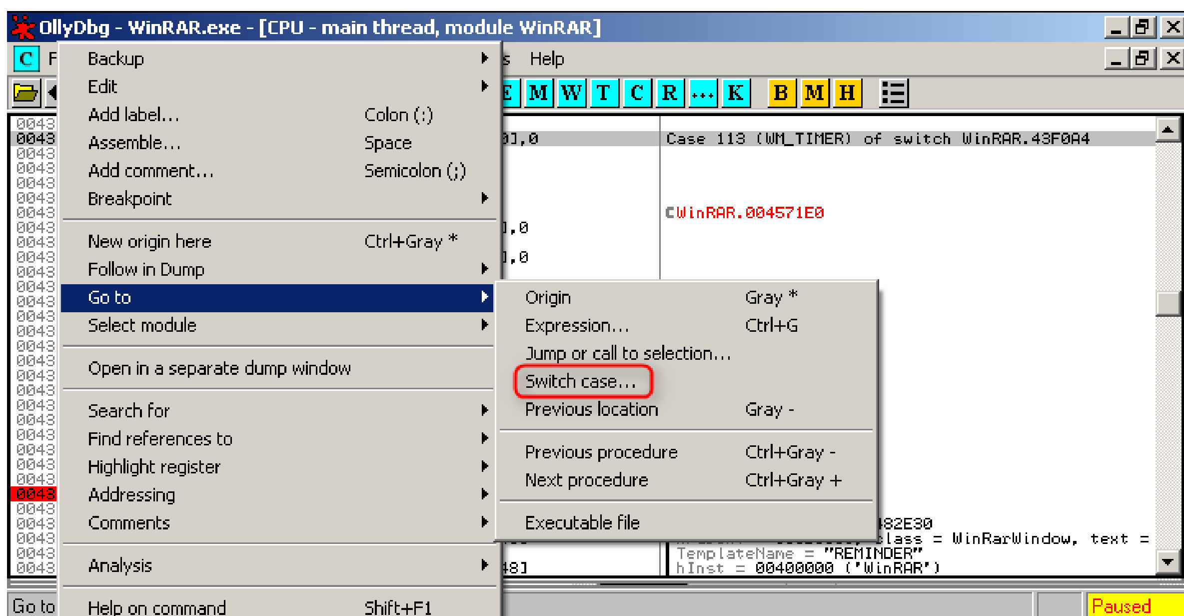
Después de la llamada al cuadro de dialogo, podemos observar varios saltos condicionales y comparaciones. Dependiendo del botón que elegimos, saltaremos a una sección de código u otra. Si por ejemplo decidimos cerrar el cuadro de dialogo saltaremos al código encargado de cerrar el dialogo:

```
OllyDbg - WinRAR.exe - [CPU - main thread, module WinRAR]
File  View  Debug  Trace  Plugins  Options  Windows  Help

0043F7E9  .v 74 32        JZ SHORT 0043F81D
0043F7EB  .  A1 2C804C00  MOV EAX,DWORD PTR DS:[4C802C]
0043F7F0  .  83F8 28      CMP EAX,28
0043F7F3  .v 7F 04        JG SHORT 0043F7F9
0043F7F5  .  85C0         TEST EAX,EAX
0043F7F7  .v 7D 24        JGE SHORT 0043F81D
0043F7F9  >  C605 95684A0( MOV BYTE PTR DS:[4A6895],1
0043F800  .  6A 00        PUSH 0                          InitParam = 0
0043F802  .  68 302E4800  PUSH 00482E30                   DialogProc = WinRAR.482E30
0043F807  .  FF35 4C4D4C0( PUSH DWORD PTR DS:[4C4D4C]     hParent = 000B0108, class = WinRarWindow, text =
0043F80D  .  68 74694A00  PUSH OFFSET 004A6974            TemplateName = "REMINDER"
0043F812  .  FF35 48204B0( PUSH DWORD PTR DS:[4B2048]     hInst = 00400000 ('WinRAR')
0043F818  .  E8 171C0600  CALL <JMP.&USER32.DialogBoxParamA>  USER32.DialogBoxParamA
0043F81D  >  803D 94684A0( CMP BYTE PTR DS:[4A6894],0
0043F824  .v 74 1C        JE SHORT 0043F842
0043F826  .  803D F4764C0( CMP BYTE PTR DS:[4C76F4],0
0043F82D  .v 75 13        JNE SHORT 0043F842
0043F82F  .  C605 94684A0( MOV BYTE PTR DS:[4A6894],0
0043F836  .  6A 00        PUSH 0                          lParam = 0
0043F838  .  6A 00        PUSH 0                          wParam = 0
0043F83A  .  6A 10        PUSH 10                         Msg = WM_CLOSE
0043F83C  .  56           PUSH ESI                        hWnd
0043F83D  .  E8 D81D0600  CALL <JMP.&USER32.PostMessageA>  USER32.PostMessageA
0043F842  >  833D F0764C0( CMP DWORD PTR DS:[4C76F0],0
0043F849  .v 75 2D        JNE SHORT 0043F878
0043F84B  .  833D E8764C0( CMP DWORD PTR DS:[4C76E8],0
0043F852  .v 75 24        JNE SHORT 0043F878
0043F854  .  833D 08774C0( CMP DWORD PTR DS:[4C7708],-1
0043F85B  .v 74 1B        JE SHORT 0043F878
0043F85D  .  6A 0A        PUSH 0A                         Timeout = 10. ms
0043F85F  .  FF35 08774C0( PUSH DWORD PTR DS:[4C7708]     hObject = 0000000C

Thread 2. (ID 000007F0) terminated, exit code 0                          Paused
```

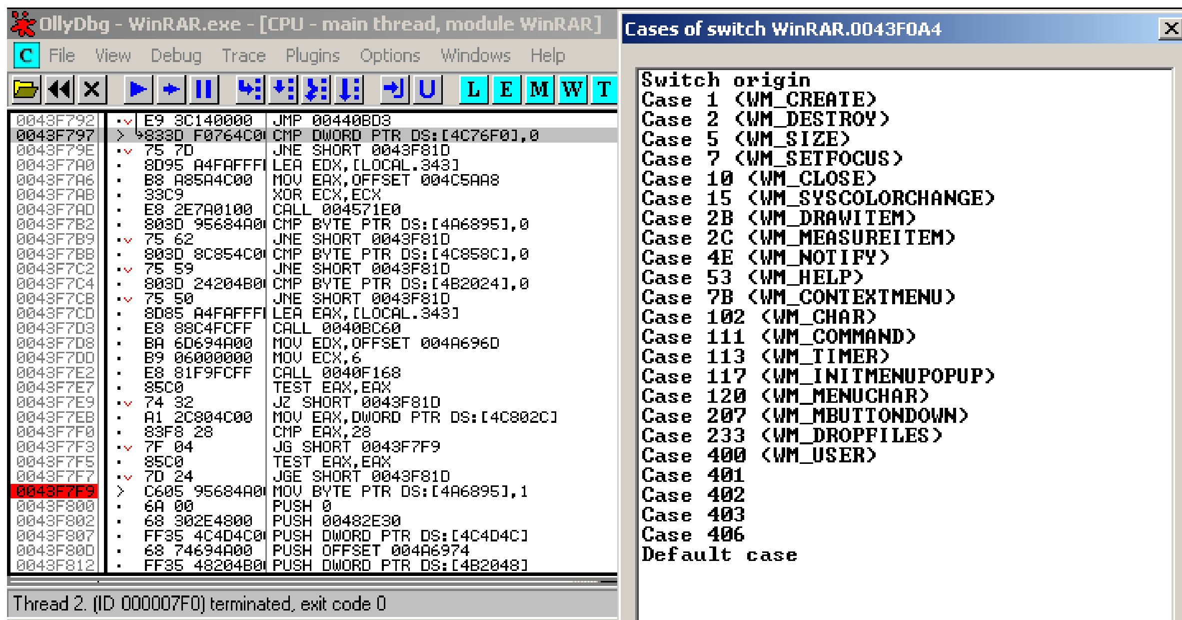
Volviendo al comienzo de nuestra sección, a la altura del switch podemos ver una primera combinación de salto y comparación:



Vemos que la instrucción en la dirección 43F79E salta por encima del call que abre nuestro nag. Situemonos en el inicio de la combinación saltar/comparar. Seleccionamos la línea con el "Case 113 (WM_TIMER)", hacemos clic con el botón derecho y seleccionamos "Go to" -> "Switch case…":

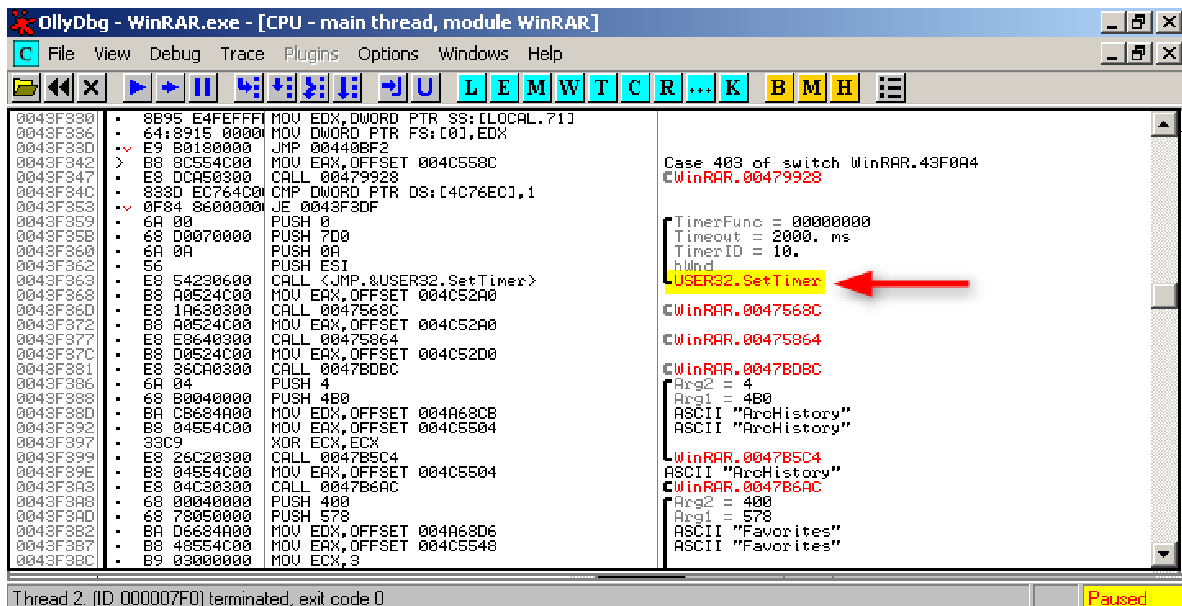


Se abre una ventana con todos los casos que pueden ser manejados por este switch:

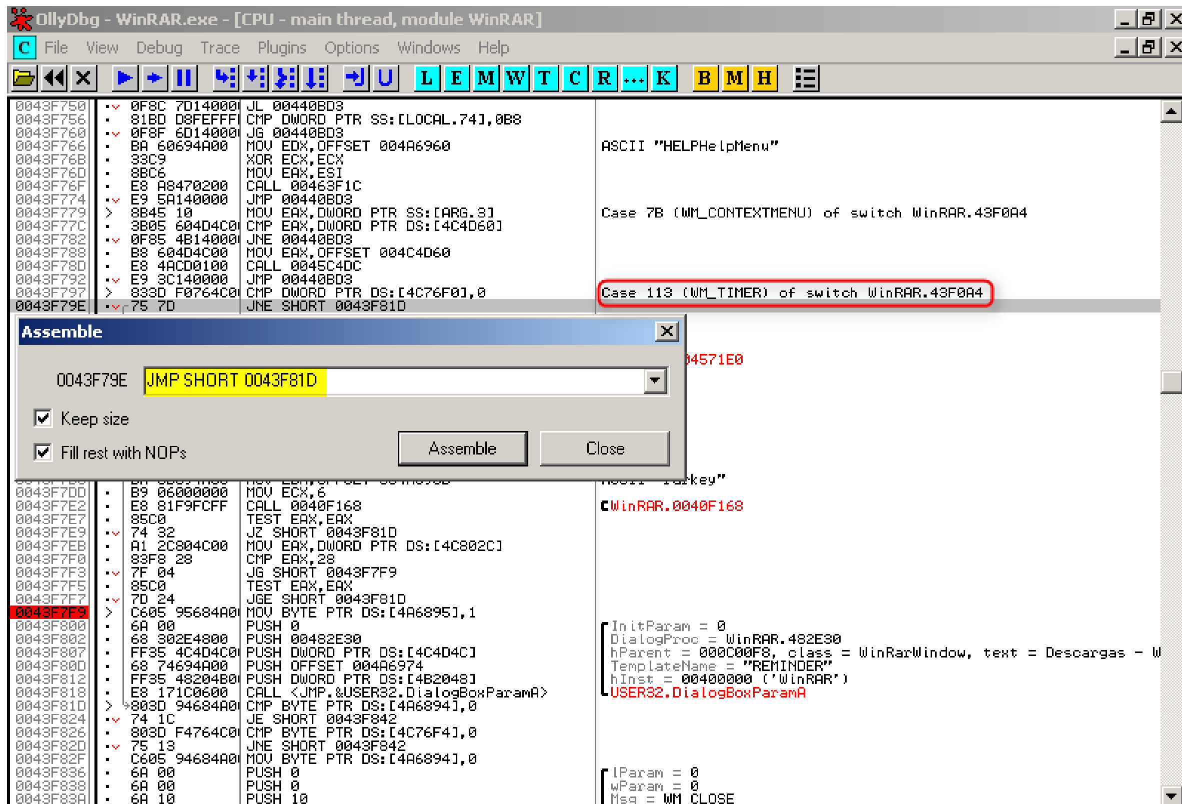


Si hacemos clic en cualquiera de ellos veremos como el lenguaje ensamblador trata estos switch como si fuera una estructura de control if/then.

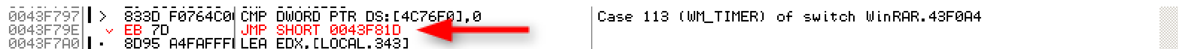
Volviendo a nuestro caso en particular, WM_TIMER hace las funciones de un contador de tiempo. Esto implica que en algún sitio del código tuvo que haberse iniciado. Subiendo en el código hasta la dirección 43F363 encontramos lo que estábamos buscando:



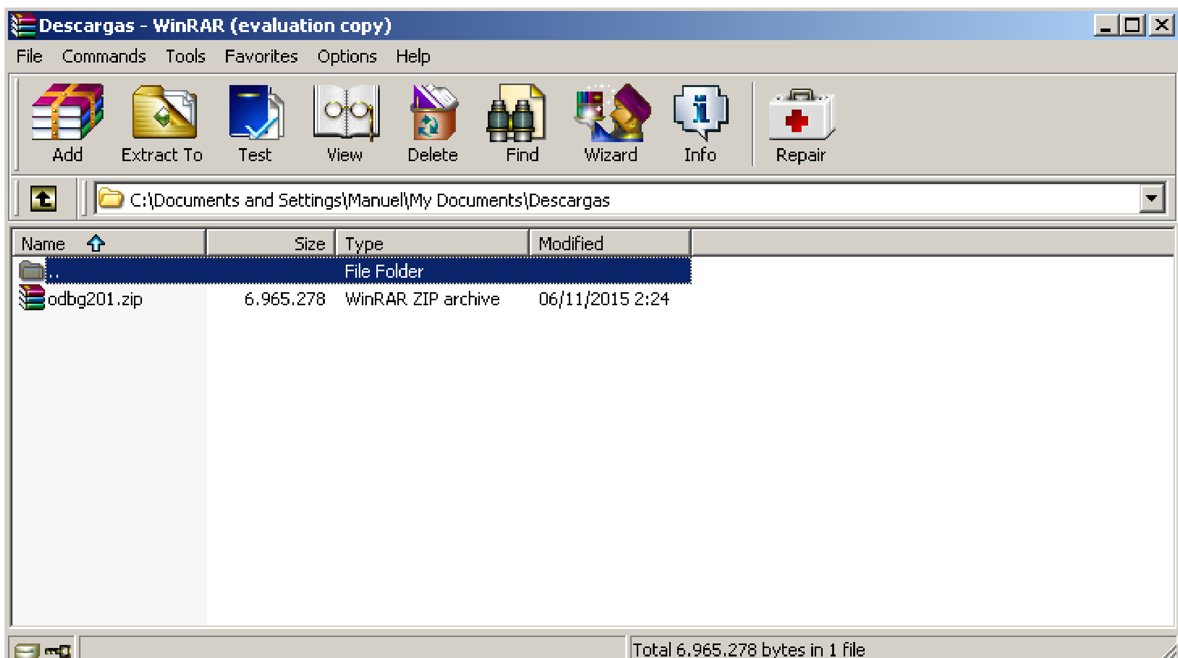
La mejor forma de parchear el programa es no dejar que el contador de tiempo aparezca después de pasado los 40 días. Para ello nos situamos en el principio del “Case 113 (WM_TIMER)” donde se comprueba si el caso es correcto para cambiar la instrucción de JNZ a JMP (es decir, que salte siempre).



Ahora, siempre que el controlador de mensajes recibe un mensaje de que el tiempo se ha agotado, simplemente lo ignorará.



Pulsamos F9 y veremos después de unos segundos que el nag ha desaparecido

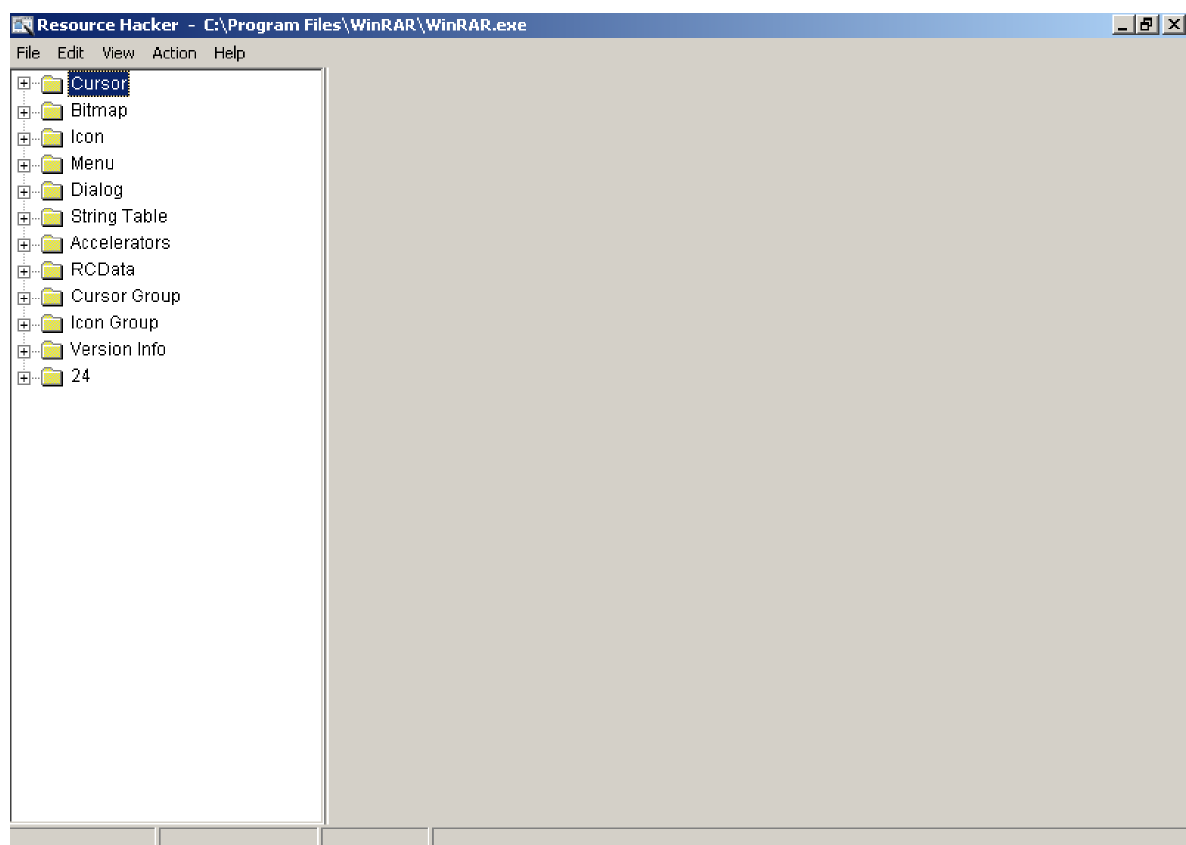


En el título de la ventana sigue poniendo “evaluation copy”, sin embargo esto no va afectar al funcionamiento del programa. Además veremos en el capítulo siguiente como solucionar el problema de los mensajes de las ventanas. Por ahora lo importante es que tenemos una aplicación que no expira en el tiempo.

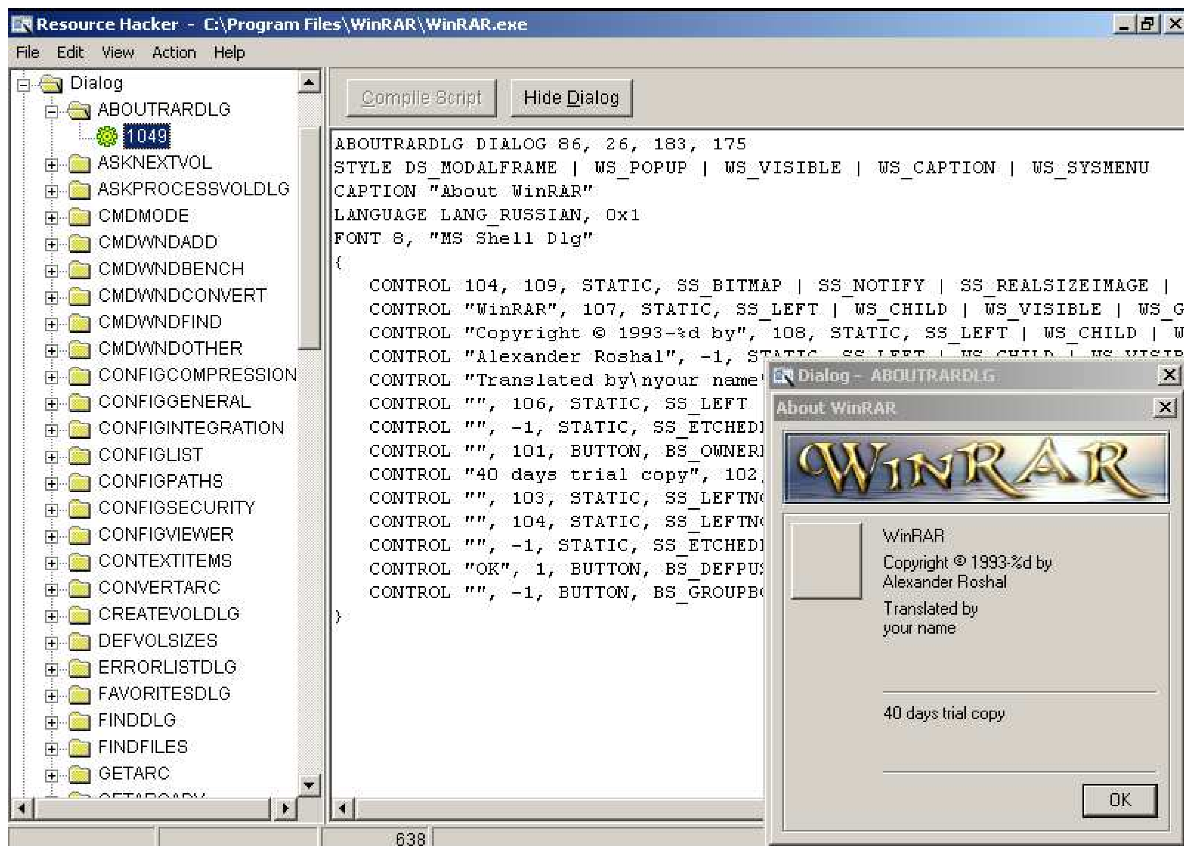
Otra forma de solucionar este ejercicio

Podemos encontrar nuestro cuadro de dialogo utilizando la herramienta “Resource Hacker” en lugar de “Call Stack”. Resource Hacker nos permite manipular los recursos que se encuentran dentro del archivo PE. Cada recurso que utiliza la aplicación (botones, cuador de dialogos, mapas de bits, iconos, cadenas de texto) se almacena en una sección específica del código. Más adelante veremos con más detalle el tema de los recursos al estudiar la estructura del archivo PE.

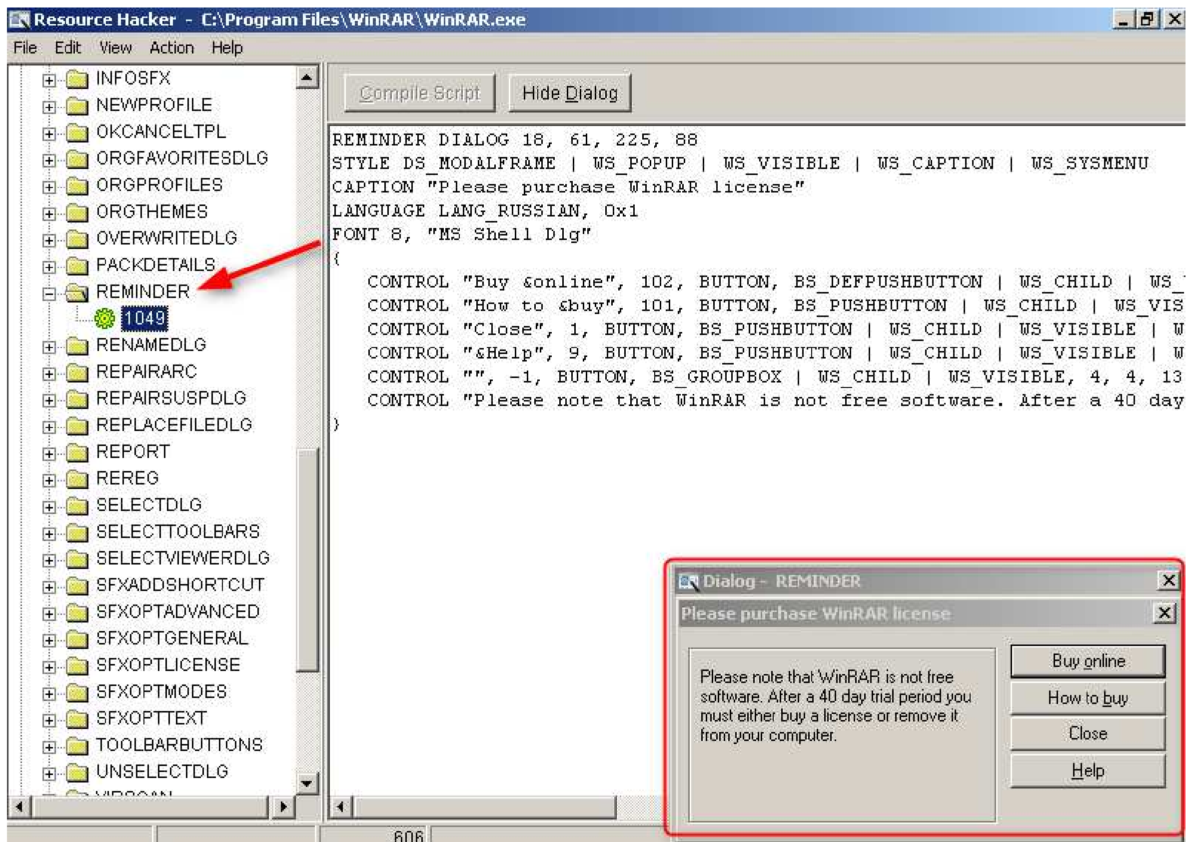
Abrimos “Resource Hacker” y cargamos nuestra aplicación:



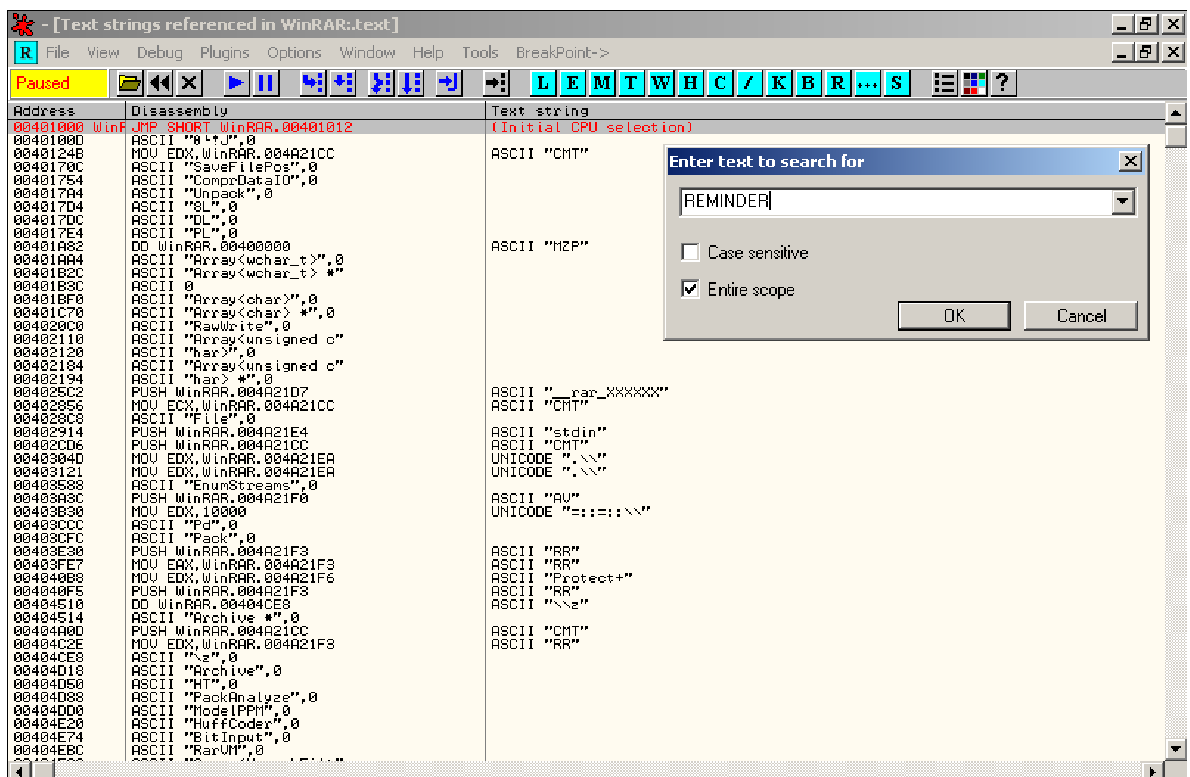
El arbol de la izquierda nos muestra los diferentes recursos de nuestra aplicación. Vamos a desplegar la carpeta Dialog y seleccionamos la primera “ABOUTRARDLG”:



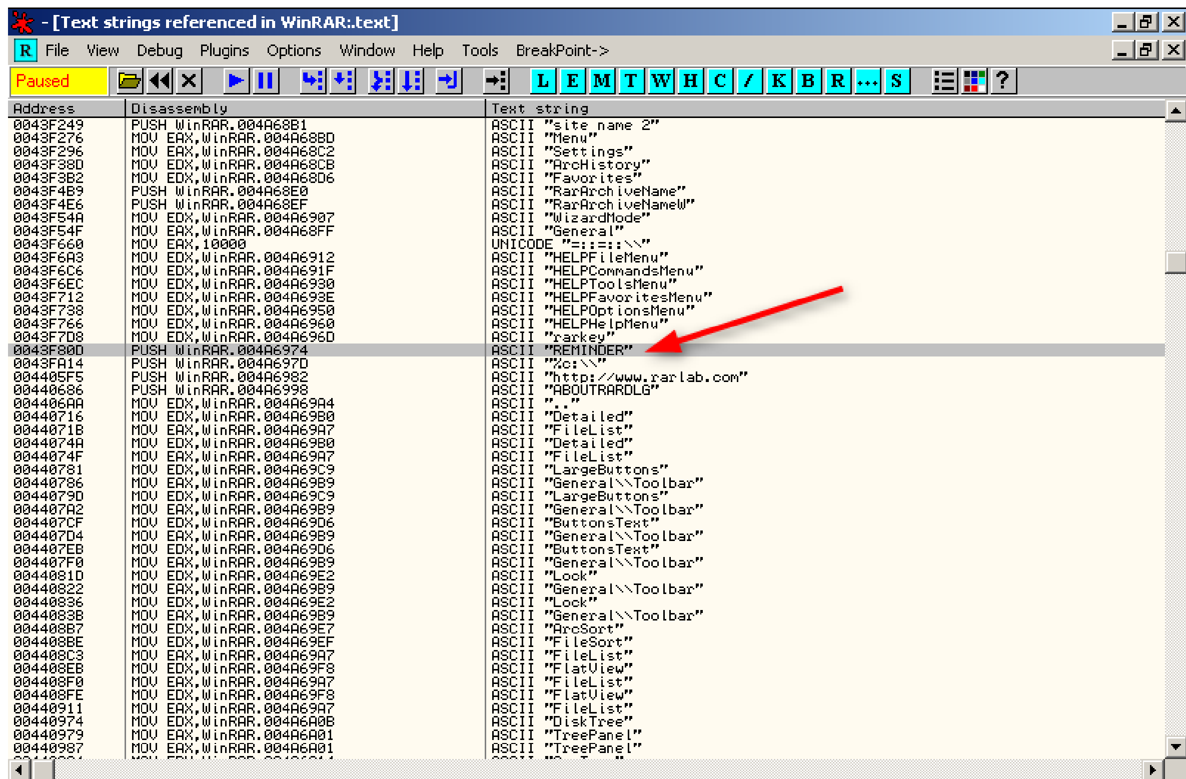
Resource Hacker nos muestra todos los datos correspondientes al dialogo seleccionado. Vemos también que abre una ventana para que tengamos una idea de como va aparecer ese dialogo en el programa. En este caso en particular se abrió la ventana de “About WinRAR”. Después de inspeccionar unas cuantas carpetas dentro del arbol “Dialog”, llegamos a la carpeta que nos interesa:



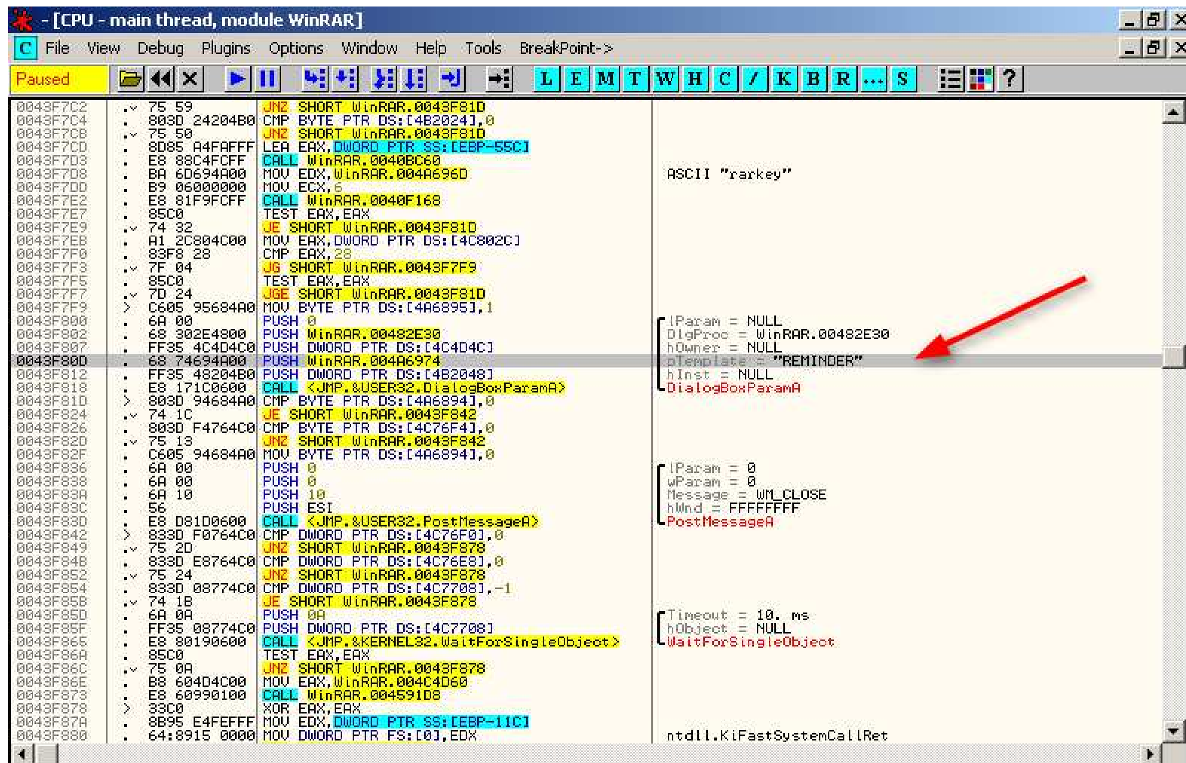
Vemos que el nombre del dialogo es “REMINDER”. Windows utiliza algunas veces un nombre y otras veces un ID para referirse a un cuadro de dialogo. En este caso utiliza el nombre “REMINDER”. Ahora ya podemos abrir Olly y buscar la cadena de texto “REMINDER”:



Hacemos doble clic sobre la línea seleccionada.



Y vemos que aparece la misma sección que obtuvimos utilizando el método del "Call Stack":



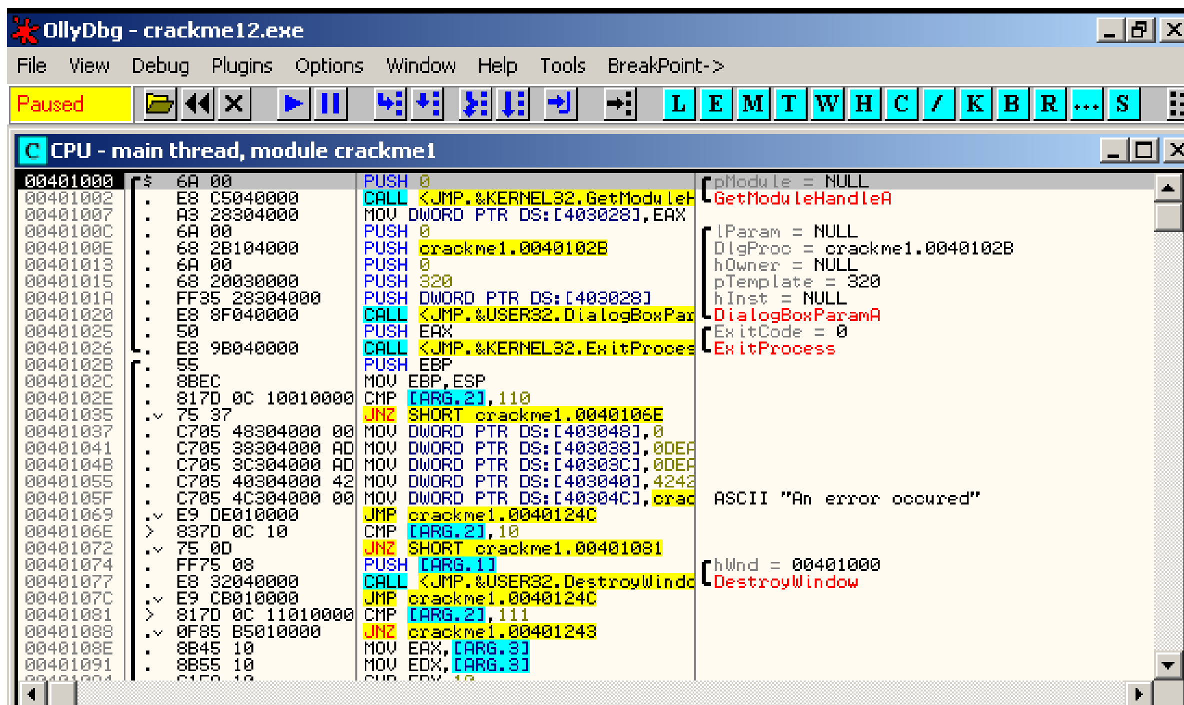
Si el recurso fuese identificado a través de un ID en lugar de un nombre, bastaría con hacer clic con el botón derecho sobre la ventana de desensamblaje y seleccionar "Search for" ->

“Command”. A continuación introduciriamos “PUSH xx”, donde ‘xx’ sería el ID (en hexadecimal) del recurso, lo que nos llevaría al CALL del cuadro de dialogo.

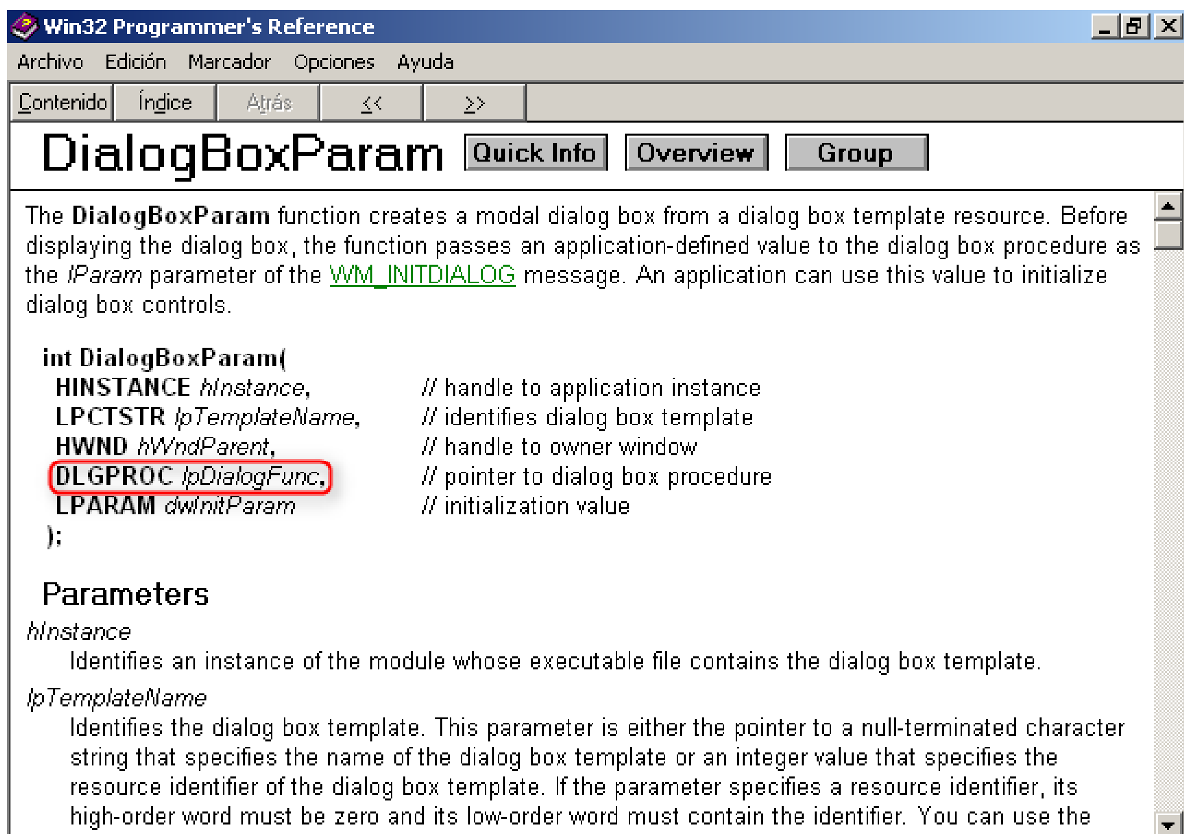
7.13 Caso práctico 13: Los mensajes de las ventanas.

En casi todos los programas, con excepción de las aplicaciones escritas en Visual Basic *sigh*, .NET, o JAVA, las tareas se llevan a cabo utilizando para los mensajes el procedimiento de devolución de llamada. Es decir, a diferencia de la programación en DOS, en Windows se crean ventanas que proveen las diferentes opciones para configurar los ajustes de un programa, los mapas de bits, los elementos del menú, etc. Después se añade un loop que continuará hasta que finalice el programa. Estos loops son responsables de recibir los mensajes de las ventanas para enviarlos a los procedimientos de devolución de llamadas. Estos mensajes pueden ser cualquier cosa, desde mover el ratón hasta hacer clic en un botón. Cuando creamos una aplicación para Windows incluimos estos loops en el procedimiento WinMain junto a una dirección para hacer la llamada. El loop envía el mensaje que recibe a nuestra función de devolución de llamada con la dirección que hemos suminisrado, y en esta devolución de llamada decidiremos si queremos hacer algo con ese mensaje en particular o, por el contrario dejar que lo maneje Windows.

Abrimos Olly y cargamos el programa Crackme12.exe.

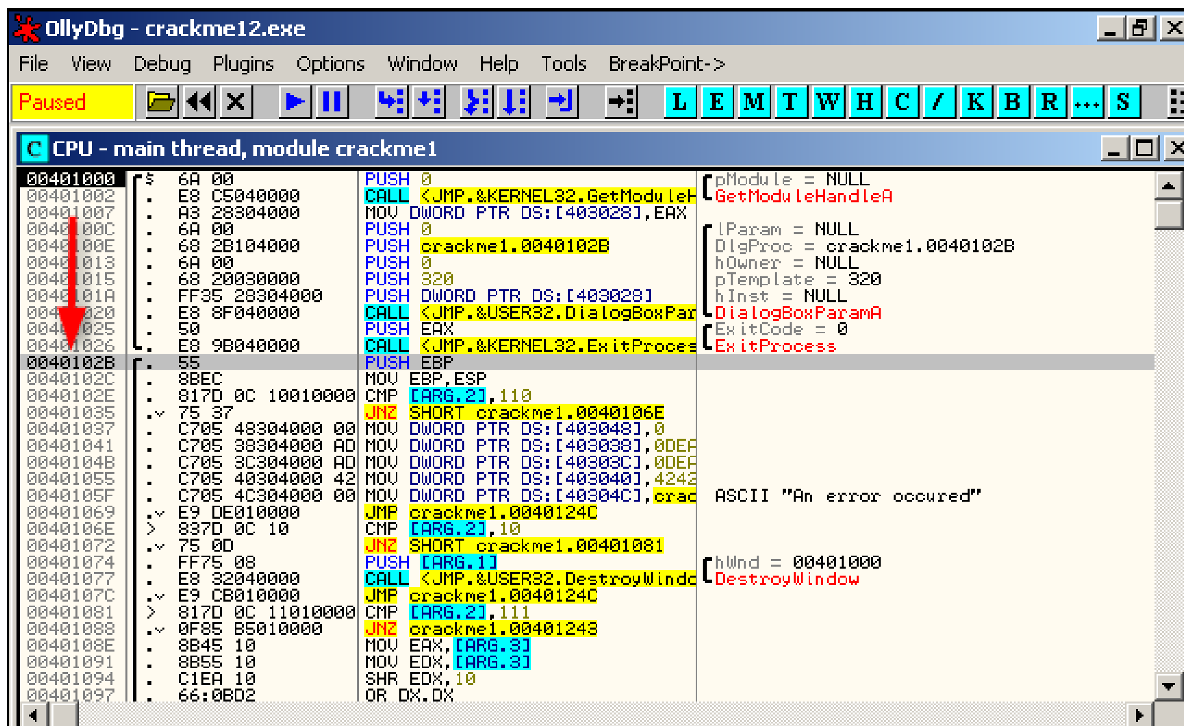


Los programas escritos en C o C++, utilizan el "dialog box" para abrir la ventana principal de la aplicación. Hacemos clic sobre la línea 401020 para ver la definición exacta:



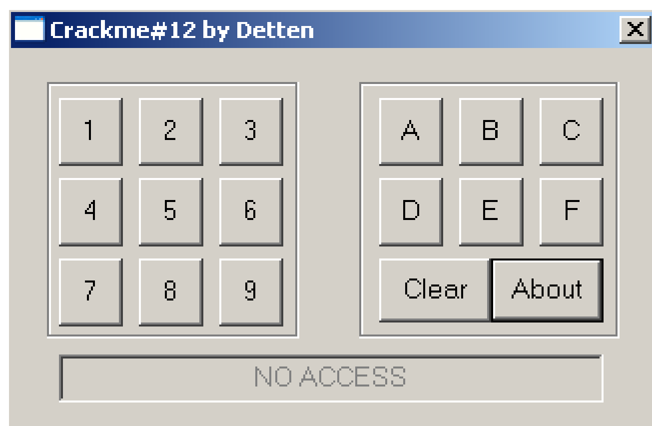
En nuesro caso, lo más importante a tener en cuenta en este CALL es la dirección de DLGPROC. Se trata pues de la dirección de la devolución de llamada de nuestra aplicación y

que se ocupará de todos los mensajes de Windows. Si volvemos a la ventana de desensamblaje podemos ver que comienza en la dirección 40102B.



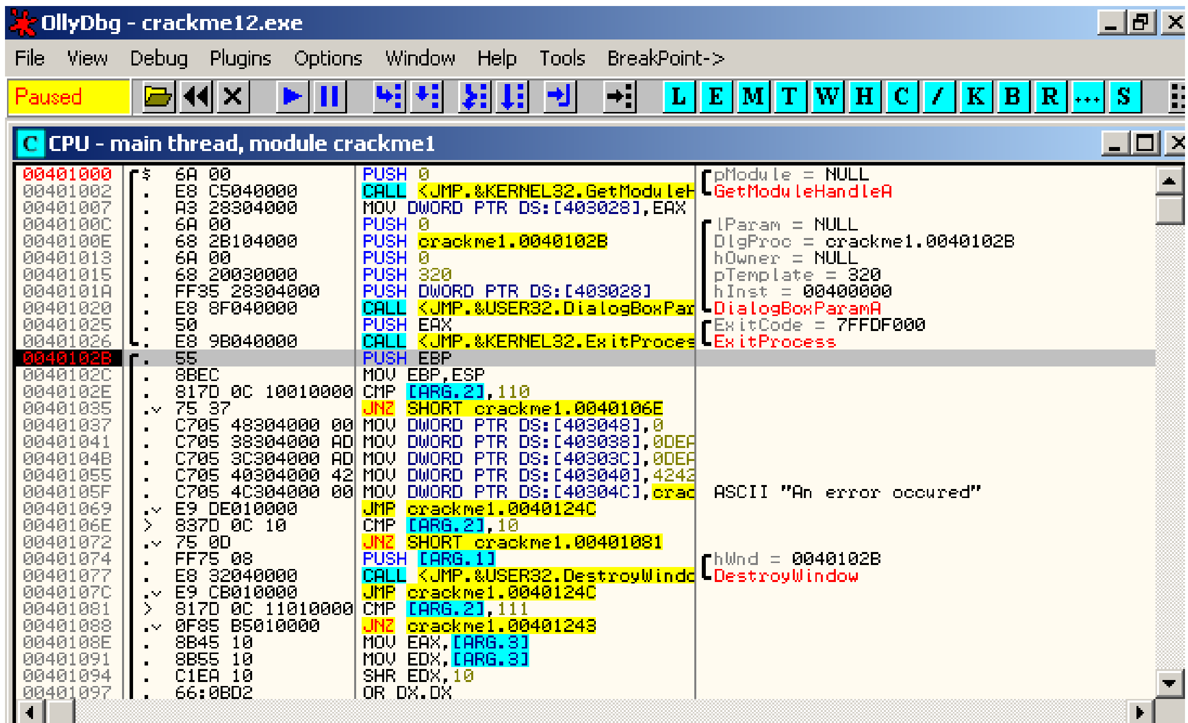
El DlgProc es como un switch gigantesco. Este procedimiento esta aquí fundamentalmente por una razón: dar respuestas a los mensajes de las ventanas que queremos responder. Si nos fijamos bien podemos observar un conjunto de instrucciones de saltos y comparación que comprobarán cada sección de código con el ID del mensaje que la ventana ha enviado. Si el código coincide con alguna de estas instrucciones de comparación, este va a ser ejecutado. En caso contrario se devolverá a la ventana para su posterior procesamiento.

Veamos este proceso un poco más de cerca. Pulsamos F9.



Si hacemos clic en cualquier botón, vemos que no sucede nada. Parece que tenemos que introducir un código especifico sin el cual la ventana permanecerá inalterable.

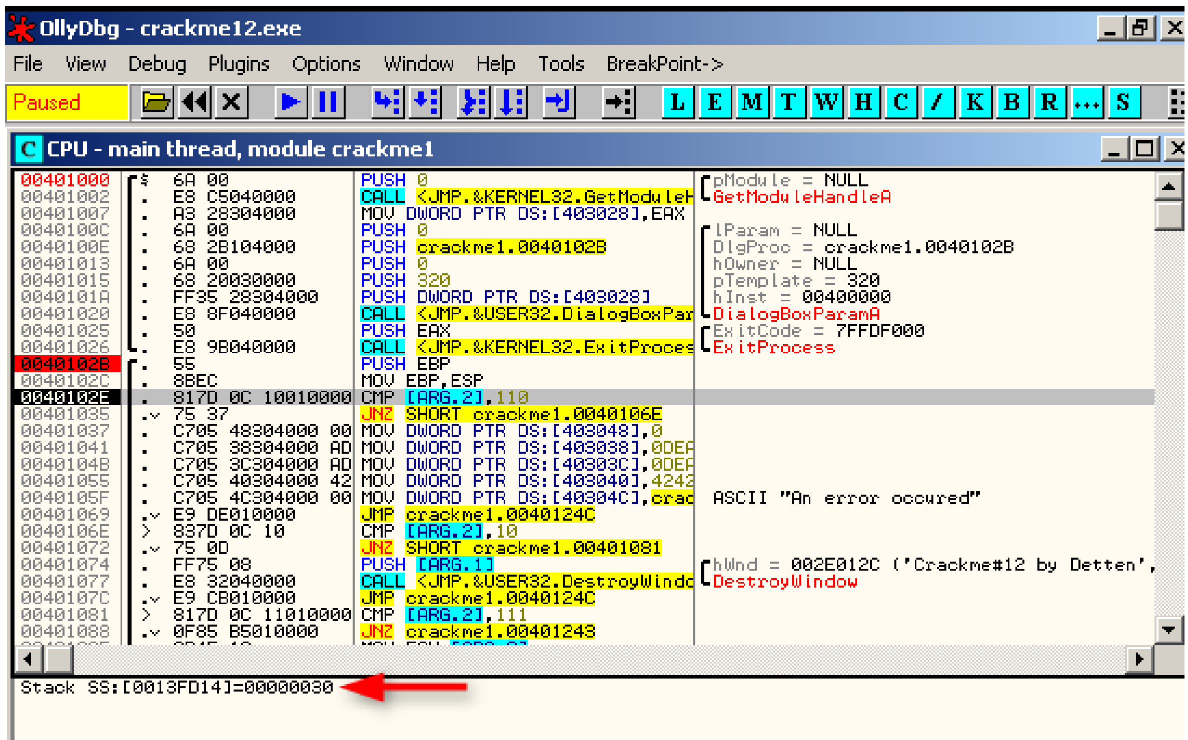
Ponemos un Breakpoint al principio de DlgProc en la dirección 40102B y reiniciamos la aplicación para poder ver 'entrar' el mensaje.



Vemos que tampronto pulsamos F9 nos detenemos en el Breakpoint. Podemos ver unas cuantas instrucciones antes de que aparezca la primera comparación en 40102E.

40102E CMP [ARG.2], 110

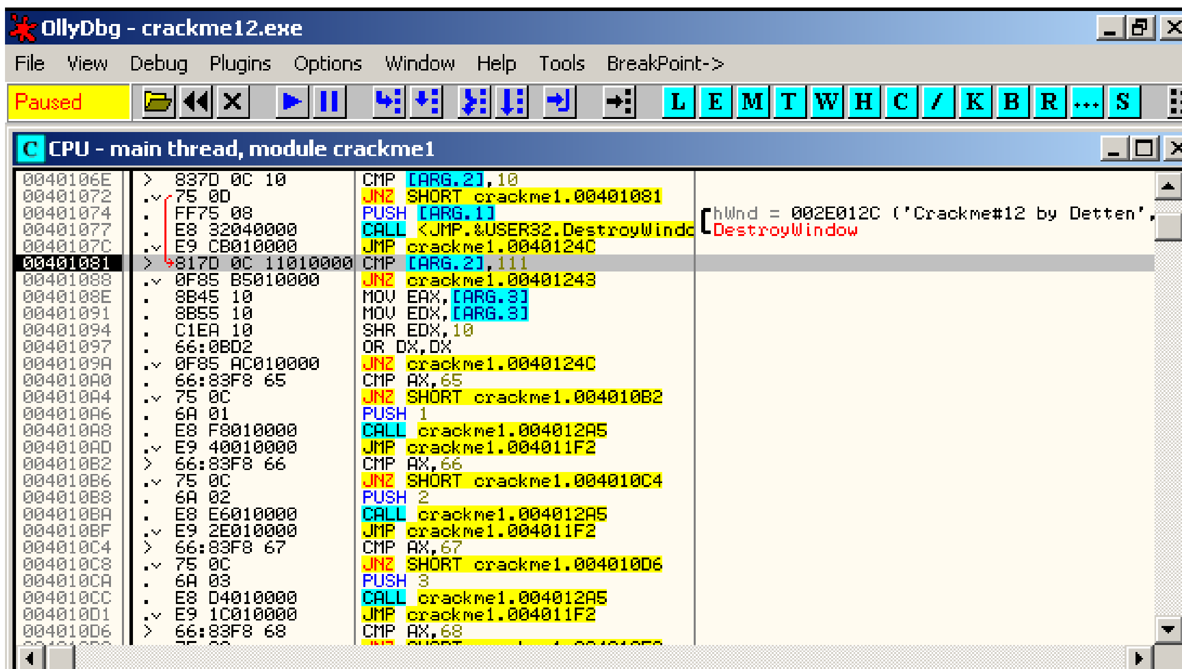
Si nos fijamos en el listado de los mensajes de Windows vemos que el ID 110 corresponde a InitDialog. Este mensaje le da a nuestra aplicación la oportunidad de iniciar alguna cosa. En nuestro caso se ejecutará el código que comienza en 401037.



Si nos fijamos en el área entre el desensamblador y el dump vemos que [ARG.2] no es 110 sino 30. Y según nuestro listado, 30 corresponde al mensaje para configurar la fuente (WM_SETFONT). Este es pues el primer mensaje que Windows envía al código.

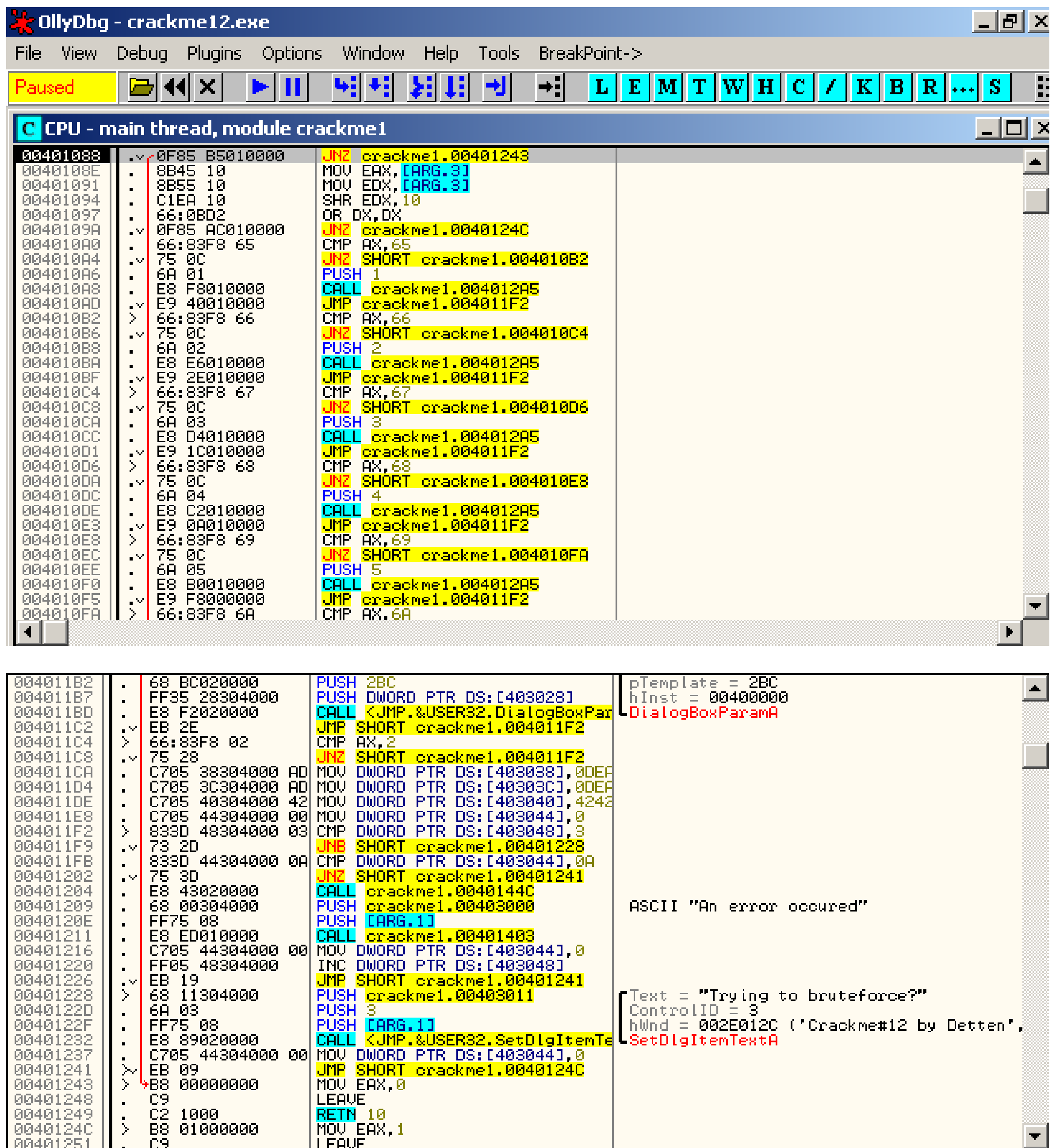
La siguiente comparación en 40106E es con 10, que corresponde a WM_CLOSE. Es decir, cuando pulsamos el botón close, se ejecutará el código en la dirección 40106E. La siguiente comparación en la dirección 401081 es con 111, lo que corresponde a WM_COMMAND.

Junto a WM_COMMAND un segundo integro es enviado en ARG.3 para ayudar a esclarecer el mensaje command.



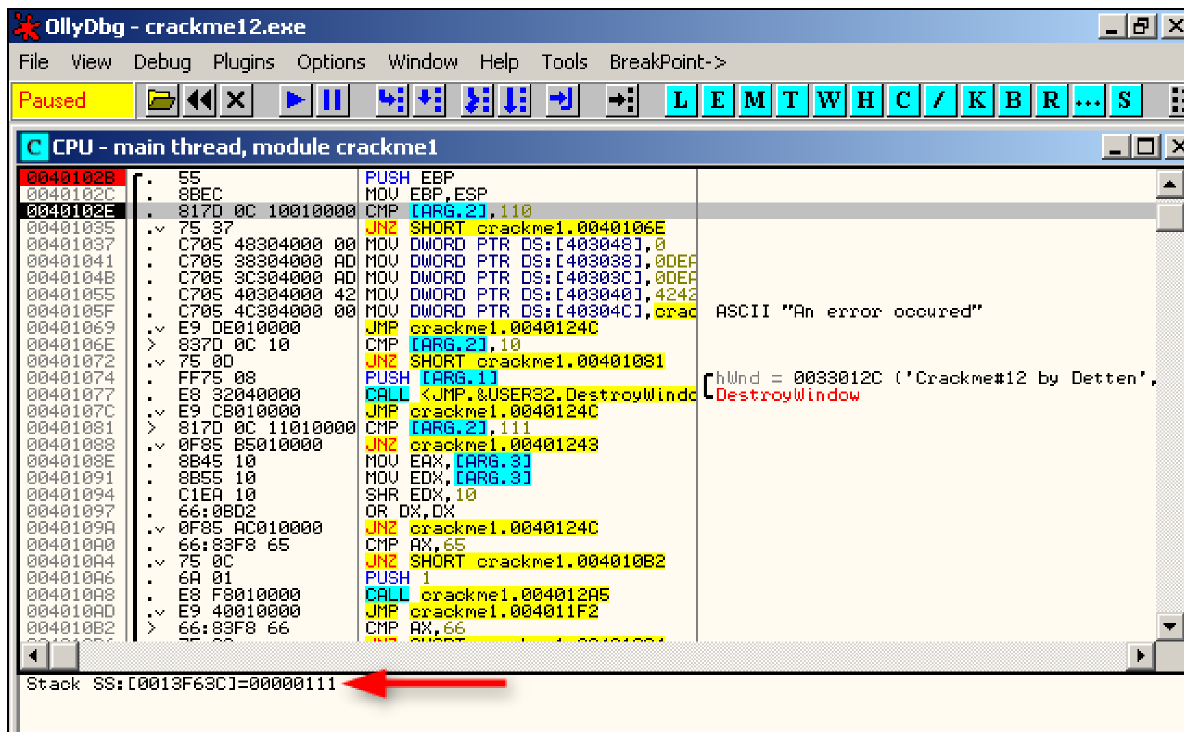
Por ejemplo, si hacemos clic en un botón, vendrá un mensaje WM_COMMAND y ARG.3 puede que tenga el ID de ese botón. Si estamos trabajando con un programa de diseño, ARG.3 podría contener las coordenadas X e Y correspondiente a la posición del ratón.

Si nos fijamos bien, WM_COMMAND es el único mensaje que maneja este procedimiento. Si pasamos línea por línea veremos que no se está ejecutando código para nuestro actual mensaje, WM_SETFONT, y simplemente regresaremos al finalizar el procedimiento.

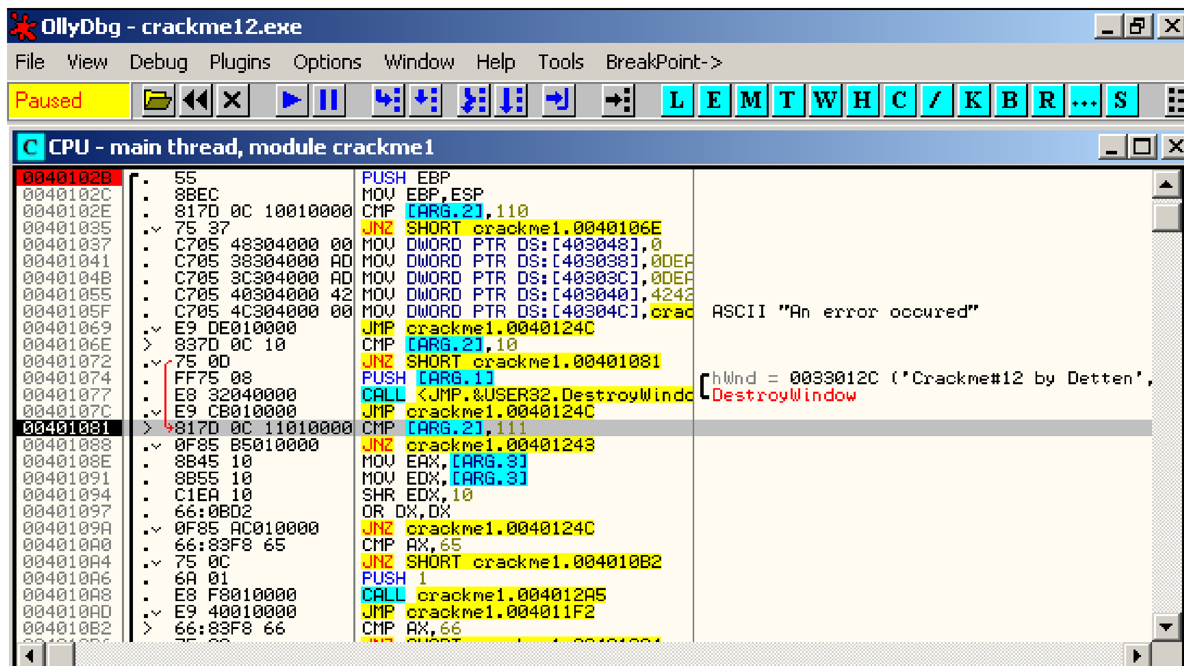


Es decir, queremos que Windows maneje este mensaje, no nosotros.

Pulsamos F9 y nos detendremos en el siguiente mensaje:



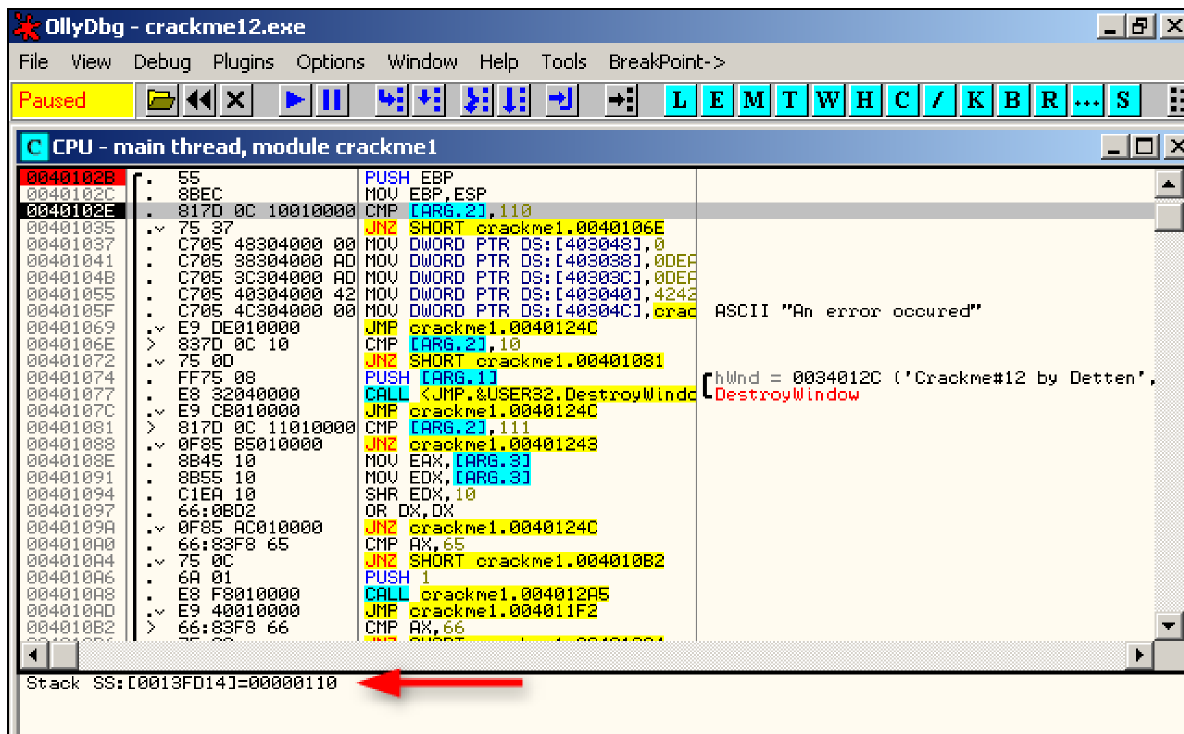
Vemos que esta vez se trata de un mensaje WM_COMMAND. Pulsamos F8 hasta llegar a la comparación que comprueba este mensaje en la dirección 401081.



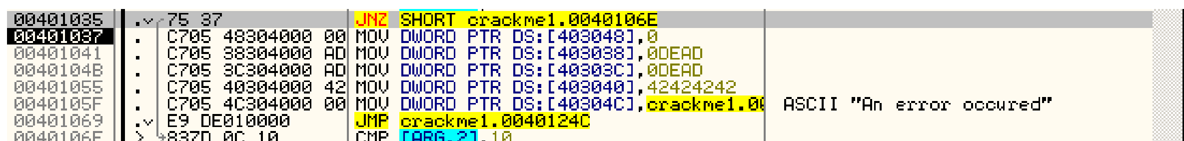
Estudiemos más de cerca el controlador WM_COMMAND:

Vemos que mueve ARG.3 a EAX y EDX. Realiza un SHR (Shift Right) en el registro EDX por la cantidad de 10 (16d). Sigue la instrucción OR, que si no es cero, entonces saltaremos. Lo que basicamente comprueba es si el quinto bit de este argumento es cero o no. Esto es asi porque los bits superiores de EDX nos muestran el ID del origen que ha sido afectado. En este caso es un cero, asi que saltaremos por encima del código restante y regresaremos de nuestra devolución de llamada.

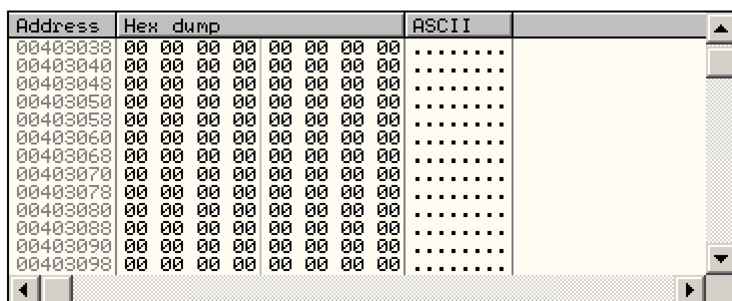
Pulsamos F9 y volvemos a parar en nuestro Breakpoint. Pulsamos F8 hasta llegar a la dirección 40102E y vemos que esta vez vamos a manejar mensajes de tipo WM_INITDIALOG.



Pulsamos F8 y vemos que no vamos coger el salto en la dirección 401035 sino que se va a ejecutar el código que viene a continuación.



En este Crackme, el siguiente código parace ser importante. Vemos unos cuantos integros que son almacenados en memoria, empezando por 403038. Veamos lo que contiene en la ventana dump.



Vemos que antes de ejecutar esas líneas la dirección en memoria se ha inicializado en ceros. Pulsamos F8 para pasar por la primera instrucción MOV y parece que no está sucediendo nada, sin embargo se ha copiado un cero en la dirección 403048. Volviendo a pulsar F8 podemos ver los efectos:

```
Address  Hex dump                 ASCII
00403038 AD DE 00 00 00 00 00 00  ↓ï......
00403040 00 00 00 00 00 00 00 00  ........
00403048 00 00 00 00 00 00 00 00  ........
00403050 00 00 00 00 00 00 00 00  ........
00403058 00 00 00 00 00 00 00 00  ........
00403060 00 00 00 00 00 00 00 00  ........
00403068 00 00 00 00 00 00 00 00  ........
00403070 00 00 00 00 00 00 00 00  ........
00403078 00 00 00 00 00 00 00 00  ........
00403080 00 00 00 00 00 00 00 00  ........
00403088 00 00 00 00 00 00 00 00  ........
00403090 00 00 00 00 00 00 00 00  ........
00403098 00 00 00 00 00 00 00 00  ........
```

Vemos que 0xDEAD ha sido copiado en memoria (en el orden little endian).

Lo mismo ocurre si volvemos a pulsar F8 pero esta vez en la dirección 40303C:

```
Address  Hex dump                 ASCII
00403038 AD DE 00 00 AD DE 00 00  ↓ï..↓ï..
00403040 00 00 00 00 00 00 00 00  ........
00403048 00 00 00 00 00 00 00 00  ........
00403050 00 00 00 00 00 00 00 00  ........
00403058 00 00 00 00 00 00 00 00  ........
00403060 00 00 00 00 00 00 00 00  ........
00403068 00 00 00 00 00 00 00 00  ........
00403070 00 00 00 00 00 00 00 00  ........
00403078 00 00 00 00 00 00 00 00  ........
00403080 00 00 00 00 00 00 00 00  ........
00403088 00 00 00 00 00 00 00 00  ........
00403090 00 00 00 00 00 00 00 00  ........
00403098 00 00 00 00 00 00 00 00  ........
```

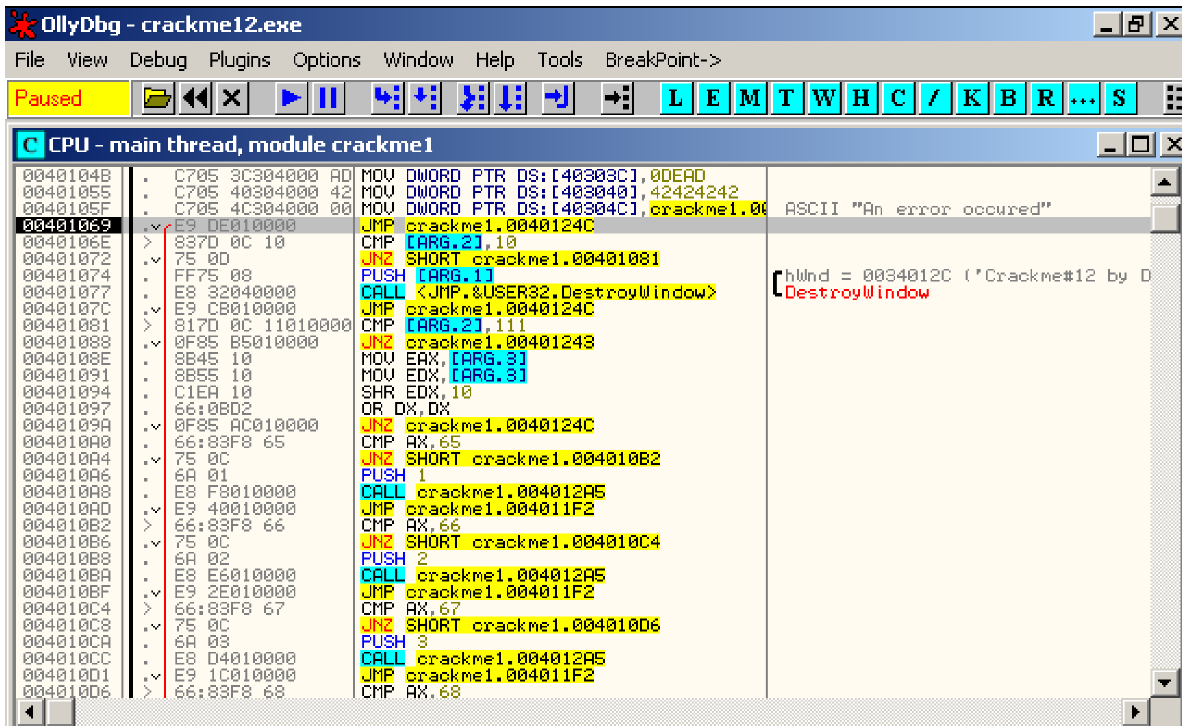
A continuación se copia el valor 42 cuatro veces en la dirección 403040. En la columna ASCII podemos ver el equivalente (BBBB).

```
Address  Hex dump                 ASCII
00403038 AD DE 00 00 AD DE 00 00  ↓ï..↓ï..
00403040 42 42 42 42 00 00 00 00  BBBB....
00403048 00 00 00 00 00 00 00 00  ........
00403050 00 00 00 00 00 00 00 00  ........
00403058 00 00 00 00 00 00 00 00  ........
00403060 00 00 00 00 00 00 00 00  ........
00403068 00 00 00 00 00 00 00 00  ........
00403070 00 00 00 00 00 00 00 00  ........
00403078 00 00 00 00 00 00 00 00  ........
00403080 00 00 00 00 00 00 00 00  ........
00403088 00 00 00 00 00 00 00 00  ........
00403090 00 00 00 00 00 00 00 00  ........
00403098 00 00 00 00 00 00 00 00  ........
```

Finalmente se copia el integro 403000 en la dirección 40304C, lo que según Olly puede ser algún código o dato que empieze en 403000.

```
Address  Hex dump                 ASCII
00403038 AD DE 00 00 AD DE 00 00  ↓ï..↓ï..
00403040 42 42 42 42 00 00 00 00  BBBB....
00403048 00 00 00 00 00 30 40 00  .....0@.
00403050 00 00 00 00 00 00 00 00  ........
00403058 00 00 00 00 00 00 00 00  ........
00403060 00 00 00 00 00 00 00 00  ........
00403068 00 00 00 00 00 00 00 00  ........
00403070 00 00 00 00 00 00 00 00  ........
00403078 00 00 00 00 00 00 00 00  ........
00403080 00 00 00 00 00 00 00 00  ........
00403088 00 00 00 00 00 00 00 00  ........
00403090 00 00 00 00 00 00 00 00  ........
00403098 00 00 00 00 00 00 00 00  ........
```

Llegados a la instrucción JMP, saltaremos y regresaremos para recibir el siguiente mensaje.



Repetiremos el proceso unas cuantas veces más (10) y veremos aparecer la ventana principal:

Según vayamos pulsando F9, aparecerá algún elemento nuevo.

El siguiente mensaje es 135, o WM_CTLCOLORBUTTON, lo que dibujará un botón en la ventana:

El siguiente es el botón número 2:

Pulsaremos F9 alrededor de 35 veces hasta completar la ventana.

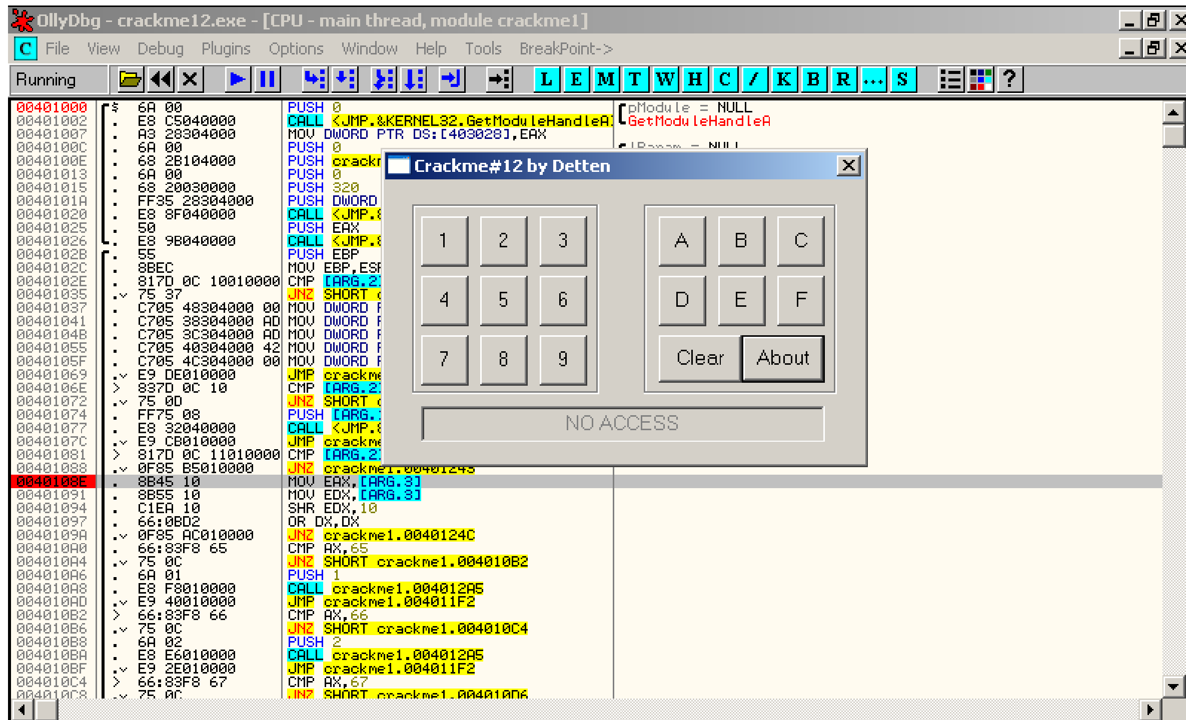
7.14 Caso práctico 14: Auto-modificación del código

Ahora que hemos visto como funciona la devolución de llamada del controlador de mensajes, intentemos crackear el Crackme12.exe.

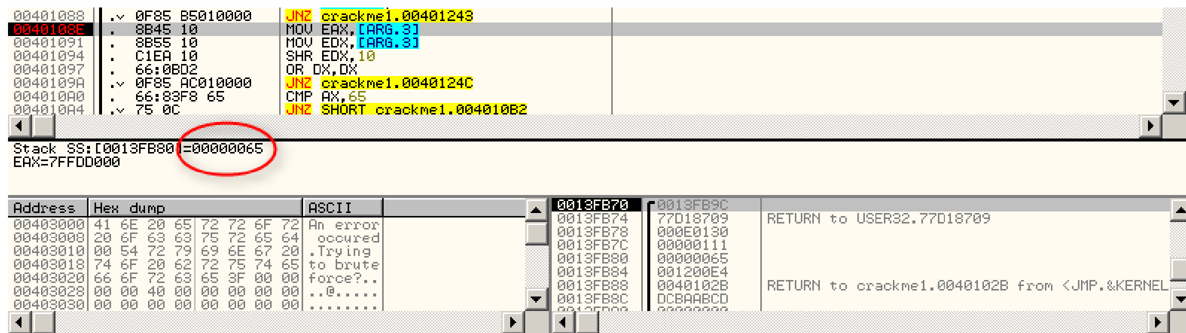
Hemos visto que solo hay tres mensajes en esta aplicación; 110 (INITDIALOG), 10(DESTROY_WINDOW) y 111(COMMAND). Todos los demás mensajes son ignorados. Ya hemos estudiado el código de "InitDialog", el código de "destroy window" solo se llama para cerrar la aplicación, así que no nos pararemos más en el. Centremonos por lo tanto en la sección WM_COMMAND.

Vamos a detener Olly en esa sección. Eliminaremos todos los Breakpoints antiguos y pongamos uno en la dirección 40108E, o después de la combinación compara/saltar para el ID 111.

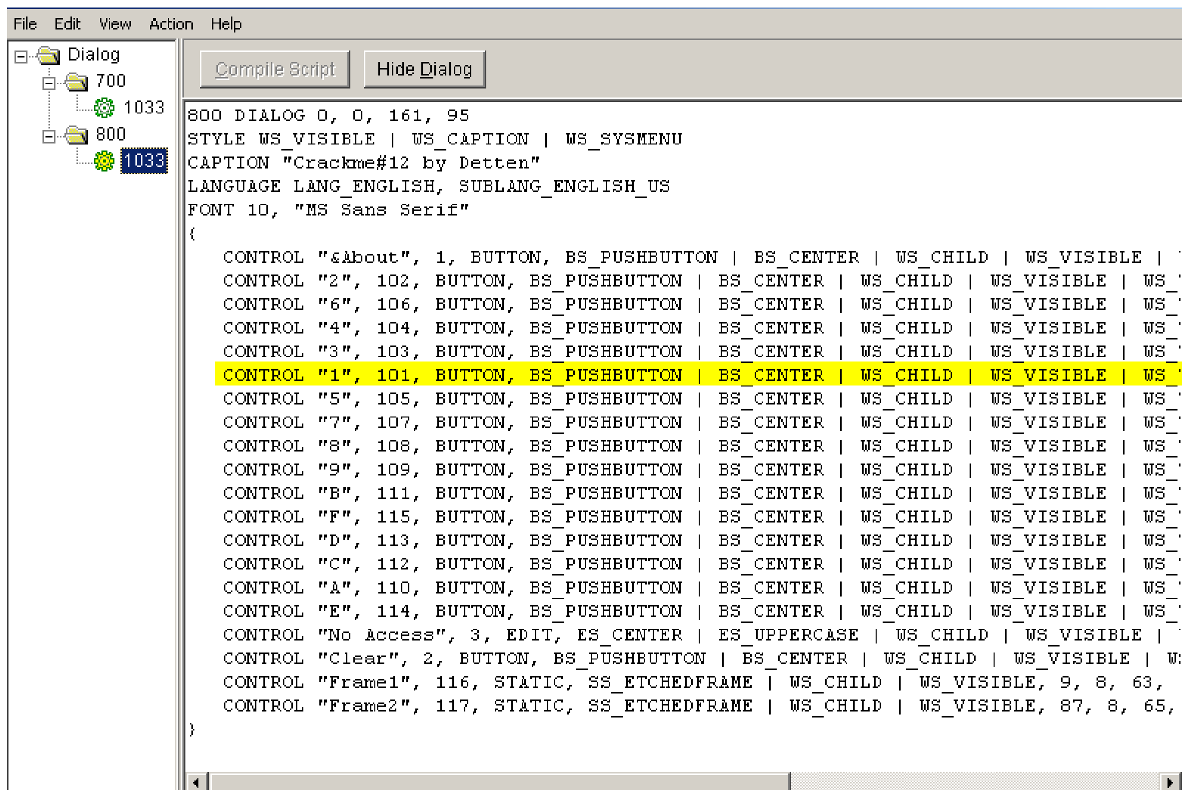
Pulsamos F9 y Olly se detiene en nuestro Breakpoint. Volvemos a pulsar F9 y aparece la ventana principal de la aplicación:



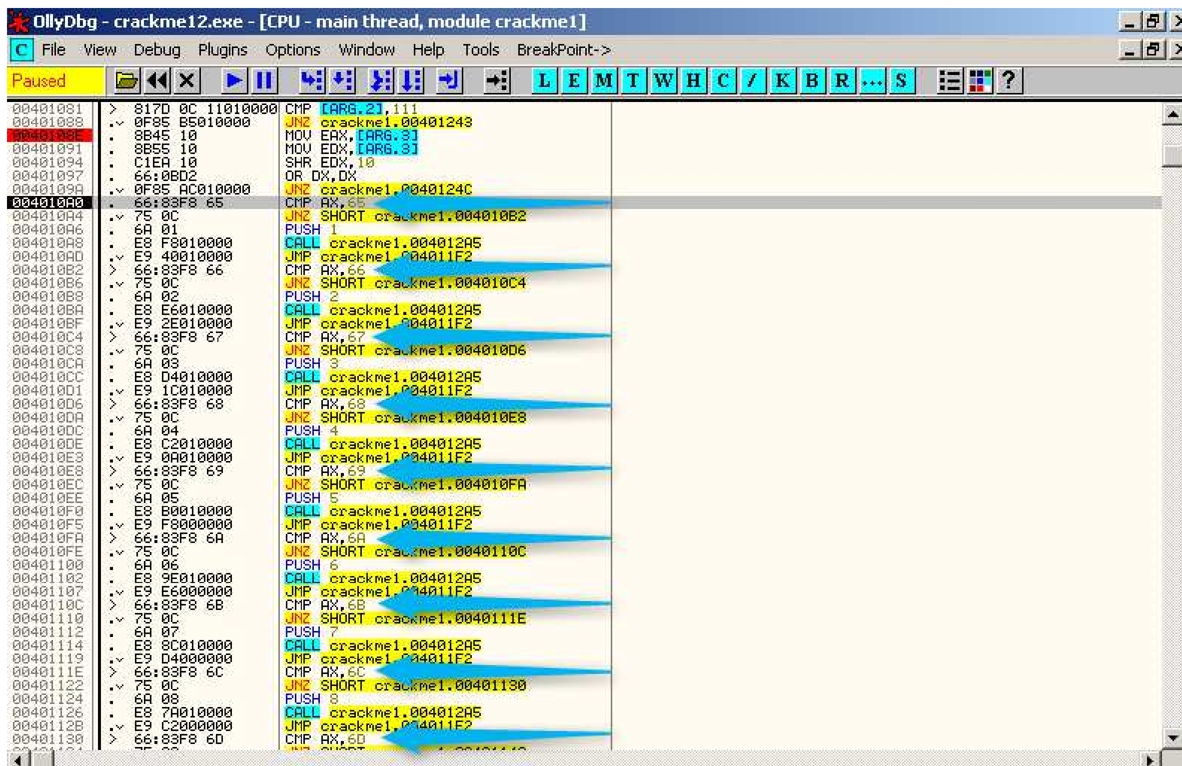
Si ahora hacemos clic en el botón ‘1’, Olly se detendrá en el Breakpoint. Además podemos ver que el contenido de la variable ARG.3 es ‘65’:



Podemos comprobar el ID del botón número ‘1’, abriendo la aplicación con “Resource Hacker” y verificar que coincide con el número 65 (101d).



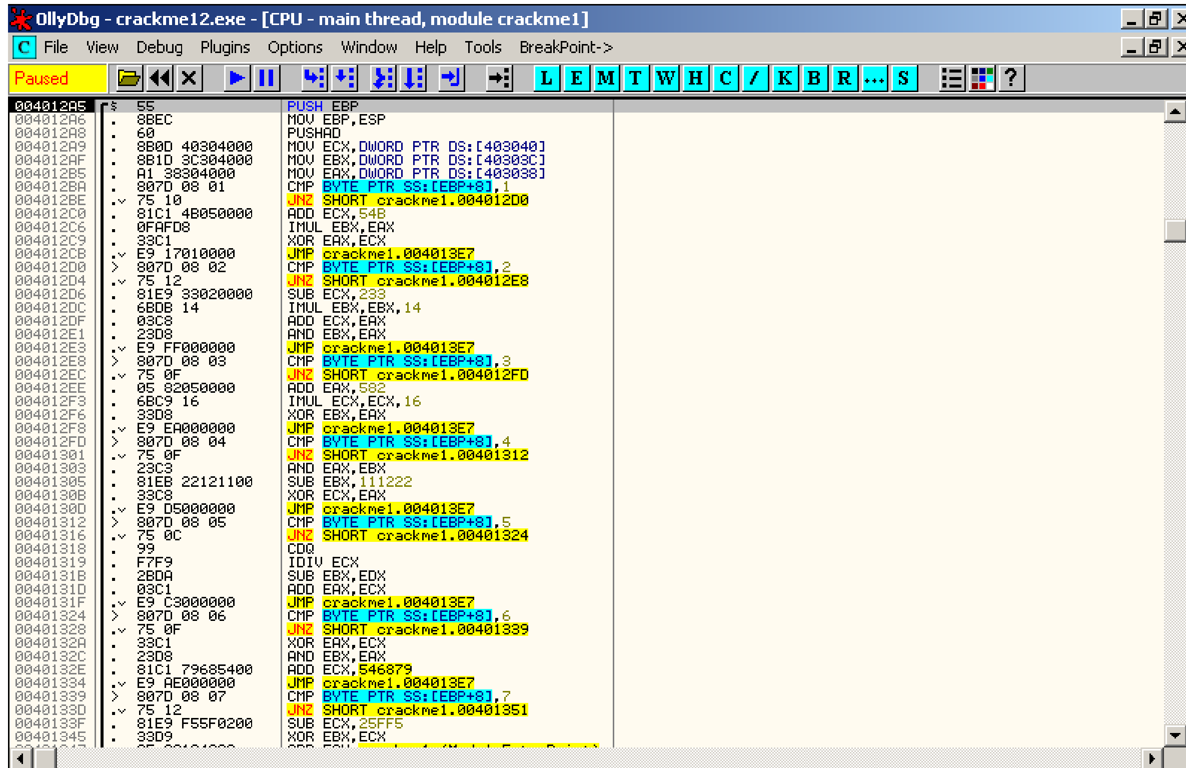
Pulsamos F8 y vamos pasando línea tras línea hasta llegar a la primera instrucción de comparación en la dirección 4010A0. Aquí es donde se compara el ID que se ha enviado con el mensaje, con el ID inmodificable de la aplicación:



Resumiendo, esta sección lo que hace es compara el ID con todos los posibles ID's y cuando encuentra una coincidencia, llamará a una sección en el código que controlará ese botón

en particular. Vemos también que justo antes del CALL se empuja un valor a la pila; 1 para 0x65, 2 para 0x66, etc. Ya que todos los CALL's están llamando al mismo lugar, el código en esa sección va a diferenciar el botón que ha sido pulsado del valor que está en la pila; 1 para el botón 1, 2 para el botón 2, etc.

Vamos pasando línea por línea hasta llegar al CALL, donde pulsaremos F7 para situarnos en la siguiente sección de código:



Vemos que estamos accediendo al mismo lugar en la memoria, al que accedimos en la sección de WM_INITDIALOG (403038). Miremos lo que hay en esa dirección en la ventana dump.

Address	Hex dump	ASCII
00403038	AD DE 00 00 AD DE 00 00	¡Þ..¡Þ..
00403040	42 42 42 42 00 00 00 00	BBBB....
00403048	00 00 00 00 00 30 40 00	.....0@.
00403050	00 00 00 00 00 00 00 00	....
00403058	00 00 00 00 00 00 00 00	....
00403060	00 00 00 00 00 00 00 00	....
00403068	00 00 00 00 00 00 00 00	....
00403070	00 00 00 00 00 00 00 00	....
00403078	00 00 00 00 00 00 00 00	....
00403080	00 00 00 00 00 00 00 00	....
00403088	00 00 00 00 00 00 00 00	....
00403090	00 00 00 00 00 00 00 00	....
00403098	00 00 00 00 00 00 00 00	....
004030A0	00 00 00 00 00 00 00 00	....
004030A8	00 00 00 00 00 00 00 00	....

Aqui esta nuestro DEAD, dos veces, los 0x42's y la dirección 403000.

Pulsamos F8 y vemos como los 42's se mueven dentro de ECX, y los dos 0xDEAD's dentro de EBX y EAX:

```
Registers (FPU)
EAX 0000DEAD
ECX 42424242
EDX 00000000
EBX 0000DEAD
ESP 0013FB44
EBP 0013FB64
ESI 0040102B crackme1.0040102B
EDI 0013FBD8
EIP 004012C0 crackme1.004012C0
```

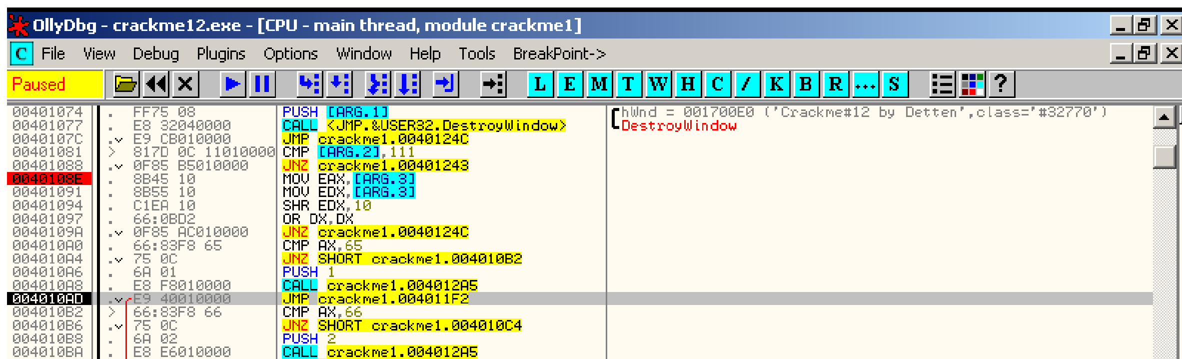
A continuación se realizan una serie de comparaciones, para verificar que botón hemos pulsado basado en el valor que se empuja en la pila. Aquí, SS:[EBP+8] está accediendo directamente al valor empujado. Ya que hicimos clic en el primer botón vamos a ejecutar el primer conjunto de instrucciones:

```
OllyDbg - crackme12.exe - [CPU - main thread, module crackme1]
004012A5  $ 55             PUSH EBP
004012A6  . 8BEC           MOV EBP,ESP
004012A8  . 60             PUSHAD
004012A9  . 8B0D 40304000  MOV ECX,DWORD PTR DS:[403040]
004012AF  . 8B1D 3C304000  MOV EBX,DWORD PTR DS:[40303C]
004012B5  . A1 38304000    MOV EAX,DWORD PTR DS:[40303
004012BA  . 807D 08 01     CMP BYTE PTR SS:[EBP+8],1
004012BE  .v 75 10         JNZ SHORT crackme1.004012D0
004012C0  . 81C1 4B050000  ADD ECX,54B
004012C6  . 0FAFD8         IMUL EBX,EAX
004012C9  . 33C1           XOR EAX,ECX
004012CB  .v E9 17010000   JMP crackme1.004013E7
004012D0  > 807D 08 02     CMP BYTE PTR SS:[EBP+8],2
004012D4  .v 75 12         JNZ SHORT crackme1.004012E8
004012D6  . 81E9 33020000  SUB ECX,233
004012DC  . 6BDB 14        IMUL EBX,EBX,14
004012DF  . 03C8           ADD ECX,EAX
004012E1  . 23D8           AND EBX,EAX
004012E3  .v E9 FF000000   JMP crackme1.004013E7
004012E8  > 807D 08 03     CMP BYTE PTR SS:[EBP+8],3
004012EC  .v 75 0F         JNZ SHORT crackme1.004012FD
004012EE  . 05 82050000    ADD EAX,582
004012F3  . 6BC9 16        IMUL ECX,ECX,16
004012F6  . 33D8           XOR EBX,EAX
```

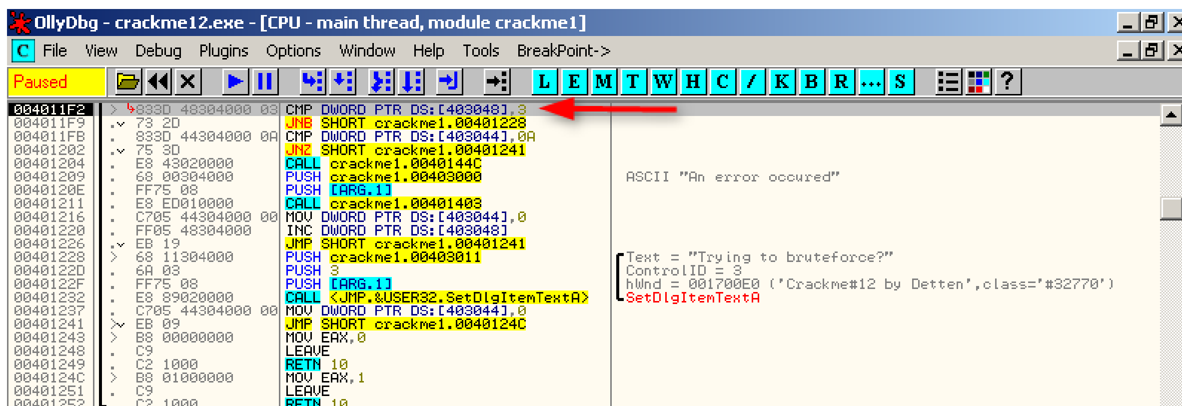
Lo primero que haremos es añadir 0x54B a ECX (42424242) lo que nos da como resultado 4242478D. Seguimos multiplicando EAX por EBX (0xDEAD veces por 0xDEAD) lo que nos da como resultado C1B080E9. Y finalmente aplicamos XOR entre el registro ECX y el registro EAX para saltar a la dirección 4013E7. Pasamos el salto y llegamos hasta:

```
OllyDbg - crackme12.exe - [CPU - main thread, module crackme1]
004013C3  . 0FAFD9         IMUL EBX,ECX
004013C6  .v EB 1F         JMP SHORT crackme1.004013E7
004013C8  > 807D 08 0E     CMP BYTE PTR SS:[EBP+8],0E
004013CC  .v 75 0D         JNZ SHORT crackme1.004013DB
004013CE  . 35 55550500    XOR EAX,55555
004013D3  . 81EB 51735800  SUB EBX,587351
004013D9  .v EB 0C         JMP SHORT crackme1.004013E7
004013DB  > 807D 08 0F     CMP BYTE PTR SS:[EBP+8],0F
004013DF  .v 75 06         JNZ SHORT crackme1.004013E7
004013E1  . 03C3           ADD EAX,EBX
004013E3  . 03D9           ADD EBX,ECX
004013E5  . 03C8           ADD ECX,EAX
004013E7  > FF05 44304000  INC DWORD PTR DS:[403044]
004013ED  . A3 38304000    MOV DWORD PTR DS:[403038],EAX
004013F2  . 891D 3C304000  MOV DWORD PTR DS:[40303C],EBX
004013F8  . 890D 40304000  MOV DWORD PTR DS:[403040],ECX
004013FE  . 61             POPAD
004013FF  . C9             LEAVE
00401400  . C2 0400        RETN 4
00401403  $ 55             PUSH EBP
00401404  . 8BEC           MOV EBP,ESP
00401406  . 50             PUSH EAX
00401407  .v EB 3F         JMP SHORT crackme1.00401448
```

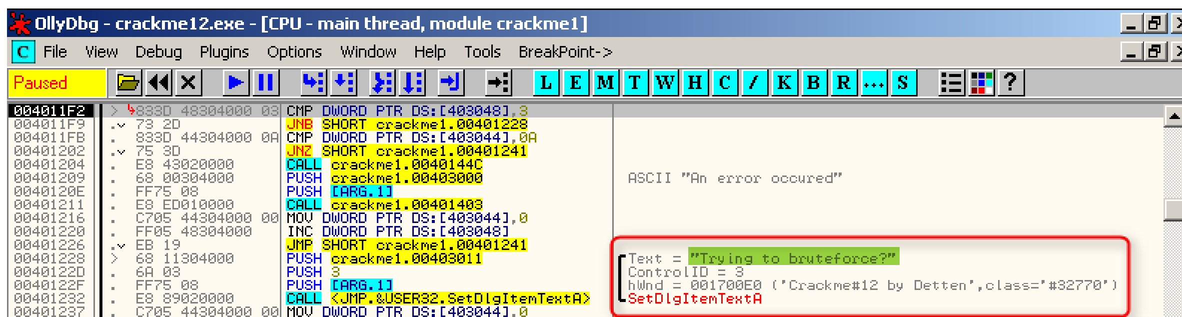
Vemos que coincide con el final del método. Si miramos atrás veremos que todos los botones hacen lo mismo; añaden un valor, aplican XOR con otro valor y saltan hasta el final. Solo se diferencian en los valores. Y llegados al final, incrementan el contenido de la memoria en la dirección 403044 (que se inició en cero), por lo que podemos suponer que se trata de una especie de contador. Después almacenan los nuevos valores de ECX, EBX y EAX de vuelta en la misma memoria de donde han sido leidos. De regreso, volvemos a estar en la función principal:



Para saltar a la dirección 4011F2:



A continuación comparamos la dirección 403048 (que es cero) con tres, y despues comparamos nuestro contador en la dirección 403044 con 0x0A. Este indica que 403044 contiene un contador que cuenta hasta 0x0A. Después saltamos si no es igual a 0x0A, señalándonos que vamos iterar por el loop 10 veces antes de salir. Si nos fijamos, podemos ver en la instrucción JMP de la dirección 4011F9 que apunta hacia un mensaje the fuerza bruta. Como la dirección 403048 tiene algún tipo de contador, si está por encima de tres, obtendremos el siguiente mensaje:



Continuamos ejecutando el programa y ahora hacemos clic en el botón '2'. Nos detenemos en nuestro Breakpoint:

```
OllyDbg - crackme12.exe - [CPU - main thread, module crackme1]
File View Debug Plugins Options Window Help Tools BreakPoint->
Paused

00401077  .   E8 32040000    CALL <JMP.&USER32.DestroyWindow>     └DestroyWindow
0040107C  .v  E9 CB010000    JMP crackme1.0040124C
00401081  >   817D 0C 11010000  CMP [ARG.2],111
00401088  .v  0F85 B5010000  JNZ crackme1.00401243
0040108E  .   8B45 10        MOV EAX,[ARG.3]
00401091  .   8B55 10        MOV EDX,[ARG.3]
00401094  .   C1EA 10        SHR EDX,10
00401097  .   66:0BD2        OR DX,DX
0040109A  .v  0F85 AC010000  JNZ crackme1.0040124C
004010A0  .   66:83F8 65     CMP AX,65
004010A4  .v  75 0C          JNZ SHORT crackme1.004010B2
004010A6  .   6A 01          PUSH 1
004010A8  .   E8 F8010000    CALL crackme1.004012A5
004010AD  .v  E9 40010000    JMP crackme1.004011F2
004010B2  >   66:83F8 66     CMP AX,66
004010B6  .v  75 0C          JNZ SHORT crackme1.004010C4
004010B8  .   6A 02          PUSH 2
004010BA  .   E8 E6010000    CALL crackme1.004012A5
004010BF  .v  E9 2E010000    JMP crackme1.004011F2
004010C4  >   66:83F8 67     CMP AX,67
004010C8  .v  75 0C          JNZ SHORT crackme1.004010D6
004010CA  .   6A 03          PUSH 3
004010CC  .   E8 D4010000    CALL crackme1.004012A5
004010D1  .v  E9 1C010000    JMP crackme1.004011F2
004010D6  >   66:83F8 68     CMP AX,68
004010DA  .v  75 0C          JNZ SHORT crackme1.004010E8
004010DC  .   6A 04          PUSH 4
004010DE  .   E8 C2010000    CALL crackme1.004012A5
004010E3  .v  E9 0A010000    JMP crackme1.004011F2
004010E8  >   66:83F8 69     CMP AX,69
004010EC  .v  75 0C          JNZ SHORT crackme1.004010FA
004010EE  .   6A 05          PUSH 5
004010F0  .   E8 B0010000    CALL crackme1.004012A5
```

ARG.3, y de regreso EAX y EDX, van a ser iguales al ID del botón número 2, o 0x66:

```
Stack SS:[0013FB80]=00000066
EAX=00000066
```

Lo que significa que ejecutaremos el código asociado al botón 2:

```
004010AD  .v  E9 40010000    JMP crackme1.004011F2
004010B2  >   66:83F8 66     CMP AX,66
004010B6  .v  75 0C          JNZ SHORT crackme1.004010C4
004010B8  .   6A 02          PUSH 2
004010BA  .   E8 E6010000    CALL crackme1.004012A5
004010BF  .v  E9 2E010000    JMP crackme1.004011F2
```

Si saltamos dentro del CALL en 4010BA, haremos lo mismo que anteriormente, solo que ahora:

1. La memoria no va contener 0xDEAD ni 42424242, sino que tendrá valores ajustados.
2. Como hizimos clic en el botón '2', vamos a ejecutar el código en 4012D6 que realizará una SUB ECX, 233 y IMUL EBX, EBX, 14 etc.

```
OllyDbg - crackme12.exe - [CPU - main thread, module crackme1]
C File View Debug Plugins Options Window Help Tools BreakPoint->
Paused
004012A5  $ 55            PUSH EBP
004012A6  . 8BEC          MOV EBP,ESP
004012A8  . 60            PUSHAD
004012A9  . 8B0D 40304000 MOV ECX,DWORD PTR DS:[403040]
004012AF  . 8B1D 3C304000 MOV EBX,DWORD PTR DS:[40303C]
004012B5  . A1 38304000   MOV EAX,DWORD PTR DS:[403038]
004012BA  . 807D 08 01    CMP BYTE PTR SS:[EBP+8],1
004012BE  .v 75 10        JNZ SHORT crackme1.004012D0
004012C0  . 81C1 4B050000 ADD ECX,54B
004012C6  . 0FAFD8        IMUL EBX,EAX
004012C9  . 33C1          XOR EAX,ECX
004012CB  .v E9 17010000  JMP crackme1.004013E7
004012D0  > 807D 08 02    CMP BYTE PTR SS:[EBP+8],2
004012D4  .v 75 12        JNZ SHORT crackme1.004012E8
004012D6  . 81E9 33020000 SUB ECX,233
004012DC  . 6BDB 14       IMUL EBX,EBX,14
004012DF  . 03C8          ADD ECX,EAX
004012E1  . 23D8          AND EBX,EAX
004012E3  .v E9 FF000000  JMP crackme1.004013E7
004012E8  > 807D 08 03    CMP BYTE PTR SS:[EBP+8],3
004012EC  .v 75 0F        JNZ SHORT crackme1.004012FD
004012EE  . 05 82050000   ADD EAX,582
004012F3  . 6BC9 16       IMUL ECX,ECX,16
004012F6  . 33D8          XOR EBX,EAX
004012F8  .v E9 EA000000  JMP crackme1.004013E7
004012FD  > 807D 08 04    CMP BYTE PTR SS:[EBP+8],4
00401301  .v 75 0F        JNZ SHORT crackme1.00401312
00401303  . 23C3          AND EAX,EBX
00401305  . 81EB 22121100 SUB EBX,111222
```

Y llegamos al final de la rutina:

```
OllyDbg - crackme12.exe - [CPU - main thread, module crackme1]
C File View Debug Plugins Options Window Help Tools BreakPoint->
Paused
004013CE  . 35 55550500   XOR EAX,55555
004013D3  . 81EB 51735800 SUB EBX,587351
004013D9  .v EB 0C        JMP SHORT crackme1.004013E7
004013DB  > 807D 08 0F    CMP BYTE PTR SS:[EBP+8],0F
004013DF  .v 75 06        JNZ SHORT crackme1.004013E7
004013E1  . 03C3          ADD EAX,EBX
004013E3  . 03D9          ADD EBX,ECX
004013E5  . 03C8          ADD ECX,EAX
004013E7  > FF05 44304000 INC DWORD PTR DS:[403044]
004013ED  . A3 38304000   MOV DWORD PTR DS:[403038],EAX
004013F2  . 891D 3C304000 MOV DWORD PTR DS:[40303C],EBX
004013F8  . 890D 40304000 MOV DWORD PTR DS:[403040],ECX
004013FE  . 61            POPAD
004013FF  . C9            LEAVE
00401400  . C2 0400       RETN 4
00401403  $ 55            PUSH EBP
00401404  . 8BEC          MOV EBP,ESP
00401406  . 50            PUSH EAX
00401407  .v EB 3F        JMP SHORT crackme1.00401448
00401409    90            NOP
0040140A    90            NOP
0040140B    42            DB 42                          CHAR 'B'
0040140C    8B            DB 8B
0040140D    02            DB 02
0040140E  . 35 0D 43 00   ASCII "5♪C",0
00401412    01            DB 01
00401413    89            DB 89
00401414    02            DB 02
00401415    83            DB 83
```

Incrementamos el contador en 403044, movemos las variables nuevas devuelta a su localidad en la memoria y regresamos a nuestro loop principal.

```
OllyDbg - crackme12.exe - [CPU - main thread, module crackme1]
File View Debug Plugins Options Window Help Tools BreakPoint->
Paused
004010BF   E9 2E010000      JMP crackme1.004011F2
004010C4   66:83F8 67       CMP AX,67
004010C8   75 0C            JNZ SHORT crackme1.004010D6
004010CA   6A 03            PUSH 3
004010CC   E8 D4010000      CALL crackme1.004012A5
004010D1   E9 1C010000      JMP crackme1.004011F2
004010D6   66:83F8 68       CMP AX,68
004010DA   75 0C            JNZ SHORT crackme1.004010E8
004010DC   6A 04            PUSH 4
004010DE   E8 C2010000      CALL crackme1.004012A5
004010E3   E9 0A010000      JMP crackme1.004011F2
004010E8   66:83F8 69       CMP AX,69
004010EC   75 0C            JNZ SHORT crackme1.004010FA
004010EE   6A 05            PUSH 5
004010F0   E8 B0010000      CALL crackme1.004012A5
004010F5   E9 F8000000      JMP crackme1.004011F2
004010FA   66:83F8 6A       CMP AX,6A
004010FE   75 0C            JNZ SHORT crackme1.0040110C
00401100   6A 06            PUSH 6
00401102   E8 9E010000      CALL crackme1.004012A5
00401107   E9 E6000000      JMP crackme1.004011F2
0040110C   66:83F8 6B       CMP AX,6B
00401110   75 0C            JNZ SHORT crackme1.0040111E
00401112   6A 07            PUSH 7
00401114   E8 8C010000      CALL crackme1.004012A5
00401119   E9 D4000000      JMP crackme1.004011F2
0040111E   66:83F8 6C       CMP AX,6C
00401122   75 0C            JNZ SHORT crackme1.00401130
00401124   6A 08            PUSH 8
```

Pulsamos F8 una vez y llegamos al final del loop principal:

```
OllyDbg - crackme12.exe - [CPU - main thread, module crackme1]
File View Debug Plugins Options Window Help Tools BreakPoint->
Paused
004011F2   833D 48304000 03  CMP DWORD PTR DS:[403048],3
004011F9   73 2D             JNB SHORT crackme1.00401228
004011FB   833D 44304000 0A  CMP DWORD PTR DS:[403044],0A
00401202   75 3D             JNZ SHORT crackme1.00401241
00401204   E8 43020000       CALL crackme1.0040144C
00401209   68 00304000       PUSH crackme1.00403000                 ASCII "An error occured"
0040120E   FF75 08           PUSH [ARG.1]
00401211   E8 ED010000       CALL crackme1.00401403
00401216   C705 44304000 00  MOV DWORD PTR DS:[403044],0
00401220   FF05 48304000     INC DWORD PTR DS:[403048]
00401226   EB 19             JMP SHORT crackme1.00401241
00401228   68 11304000       PUSH crackme1.00403011                 Text = "Trying to bruteforce?"
0040122D   6A 03             PUSH 3                                 ControlID = 3
0040122F   FF75 08           PUSH [ARG.1]                           hWnd = 000900A6 ('Crackme#12 by D
00401232   E8 89020000       CALL <JMP.&USER32.SetDlgItemTextA>     SetDlgItemTextA
00401237   C705 44304000 00  MOV DWORD PTR DS:[403044],0
00401241   EB 09             JMP SHORT crackme1.0040124C
00401243   B8 00000000       MOV EAX,0
00401248   C9                LEAVE
00401249   C2 1000           RETN 10
0040124C   B8 01000000       MOV EAX,1
00401251   C9                LEAVE
00401252   C2 1000           RETN 10
00401255   55                PUSH EBP
00401256   8BEC              MOV EBP,ESP
00401258   837D 0C 10        CMP [ARG.2],10
0040125C   75 0C             JNZ SHORT crackme1.0040126A
0040125E   6A 01             PUSH 1                                 Result = 1
00401260   FF75 08           PUSH [ARG.1]                           hWnd = 000900A6 ('Crackme#12 by D
```

Aquí comparamos 403048 (que sigue siendo cero) y saltamos al mensaje de fuerza bruta si es mayor que tres. También comparamos 403044 con 0A y saltamos al código de error si nuestro ID es mayor. Después regresamos de nuestro loop principal al loop de la ventana a la espera que hagamos algo.

Ahora que sabemos como funciona la aplicación, vamos a parchearla. Cada vez que comprobamos la dirección 403048 para ver si saltamos al mensaje de fuerza bruta, los contenidos eran cero y el salto nunca se tomaba. No obstante, la comparación en la dirección 4011FB, compara el contador que hay en la dirección 403044 y saltará despues de llegar a 0x0A. También sabemos que cada vez que pasamos por el loop, el contenido de 403044 se incrementa, por lo que podemos sospechar que este contador cuenta el número de botones que hemos pulsado.

```
004013CE  35 55555500        XOR EAX,55555
004013D3  81EB 51735800      SUB EBX,587351
004013D9  EB 0C              JMP SHORT crackme1.004013E7
004013DB  807D 08 0F         CMP BYTE PTR SS:[EBP+8],0F
004013DF  75 06              JNZ SHORT crackme1.004013E7
004013E1  03C3               ADD EAX,EBX
004013E3  03D9               ADD EBX,ECX
004013E5  03C8               ADD ECX,EAX
004013E7  FF05 44304000      INC DWORD PTR DS:[403044]
004013ED  A3 38304000        MOV DWORD PTR DS:[403038],EAX
004013F2  891D 3C304000      MOV DWORD PTR DS:[40303C],EBX
004013F8  890D 40304000      MOV DWORD PTR DS:[403040],ECX
004013FE  61                 POPAD
004013FF  C9                 LEAVE
00401400  C2 0400            RETN 4
00401403  55                 PUSH EBP
00401404  8BEC               MOV EBP,ESP
00401406  50                 PUSH EAX
00401407  EB 3F              JMP SHORT crackme1.00401448
00401409  90                 NOP
0040140A  90                 NOP
0040140B  42                 DB 42                        CHAR 'B'
0040140C  8B                 DB 8B
0040140D  02                 DB 02
0040140E  35 0D 43 00        ASCII "5♪C",0
00401412  01                 DB 01
00401413  89                 DB 89
00401414  02                 DB 02
00401415  83                 DB 83
```

Esta sección parace ser sospechoso, asi que pararemos a estudiarlo con más profundidad. Sabemos que llegamos a este código haciendo clic en al menos 0x0A (10) botones. Pongamos pues un Breakpoint en 401204 y borramos todos los demás. Reiniciamos la aplicación.

```
00401204  E8 43020000        CALL crackme1.0040144C
00401209  68 00304000        PUSH crackme1.00403000                 ASCII "An error occured"
0040120E  FF75 08            PUSH [ARG.1]
00401211  E8 ED010000        CALL crackme1.00401403
00401216  C705 44304000 00   MOV DWORD PTR DS:[403044],0
00401220  FF05 48304000      INC DWORD PTR DS:[403048]
00401226  EB 19              JMP SHORT crackme1.00401241
00401228  68 11304000        PUSH crackme1.00403011                 Text = "Trying to bruteforce?"
0040122D  6A 03              PUSH 3                                 ControlID = 3
0040122F  FF75 08            PUSH [ARG.1]                           hWnd = 000C00A6 ('Crackme#12 by D
00401232  E8 89020000        CALL <JMP.&USER32.SetDlgItemTextA>     SetDlgItemTextA
00401237  C705 44304000 00   MOV DWORD PTR DS:[403044],0
00401241  EB 09              JMP SHORT crackme1.0040124C
00401243  B8 00000000        MOV EAX,0
00401248  C9                 LEAVE
00401249  C2 1000            RETN 10
0040124C  B8 01000000        MOV EAX,1
00401251  C9                 LEAVE
00401252  C2 1000            RETN 10
00401255  55                 PUSH EBP
00401256  8BEC               MOV EBP,ESP
00401258  837D 0C 10         CMP [ARG.2],10
0040125C  75 0C              JNZ SHORT crackme1.0040126A
0040125E  6A 01              PUSH 1                                 Result = 1
00401260  FF75 08            PUSH [ARG.1]                           hWnd = 000C00A6 ('Crackme#12 by D
00401263  E8 52020000        CALL <JMP.&USER32.EndDialog>           EndDialog
00401268  EB 32              JMP SHORT crackme1.0040129C
0040126A  817D 0C 11010000   CMP [ARG.2],111
00401271  75 20              JNZ SHORT crackme1.00401293
00401273  8B45 10            MOV EAX,[ARG.3]
00401276  8B55 10            MOV EDX,[ARG.3]
00401279  C1EA 10            SHR EDX,10
```

Después de hacer clic 10 veces, la aplicación se detiene en nuestro Breakpoint. Vemos que hay un CALL hacia la dirección 40144C. Pulsamos F7 y miramos lo que hace:

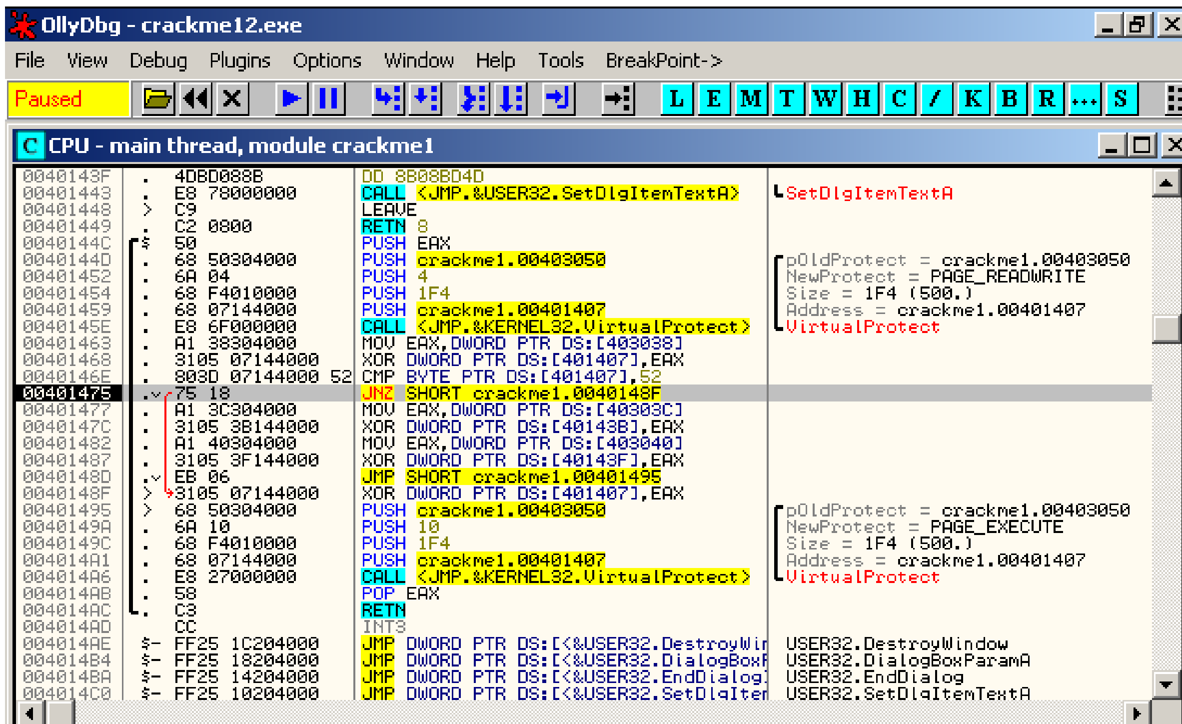
Vemos que se configura VirtualProtect para ser llamado más adelante. Lo que hace VirtualProtect es cambiar los atributos de una sección en la memoria. Utilizariamos esta función por ejemplo si queremos que una parte del código aparte de ser ejecutable también sea grabable. De esta forma escribiriamos código sobre esa sección de la memoria “sobre la marcha”. Y esto es lo que se conoce como “auto-modificación de código”: llamamos a la función VirtualProtect en una sección de la memoria, añadimos el atributo de escritura, se cambia el código y después se vuelve a llamar a VirtualProtect para cambiar el atributo a su estado original, solo ejecutable; el código ha sido modificado sobre la marcha.

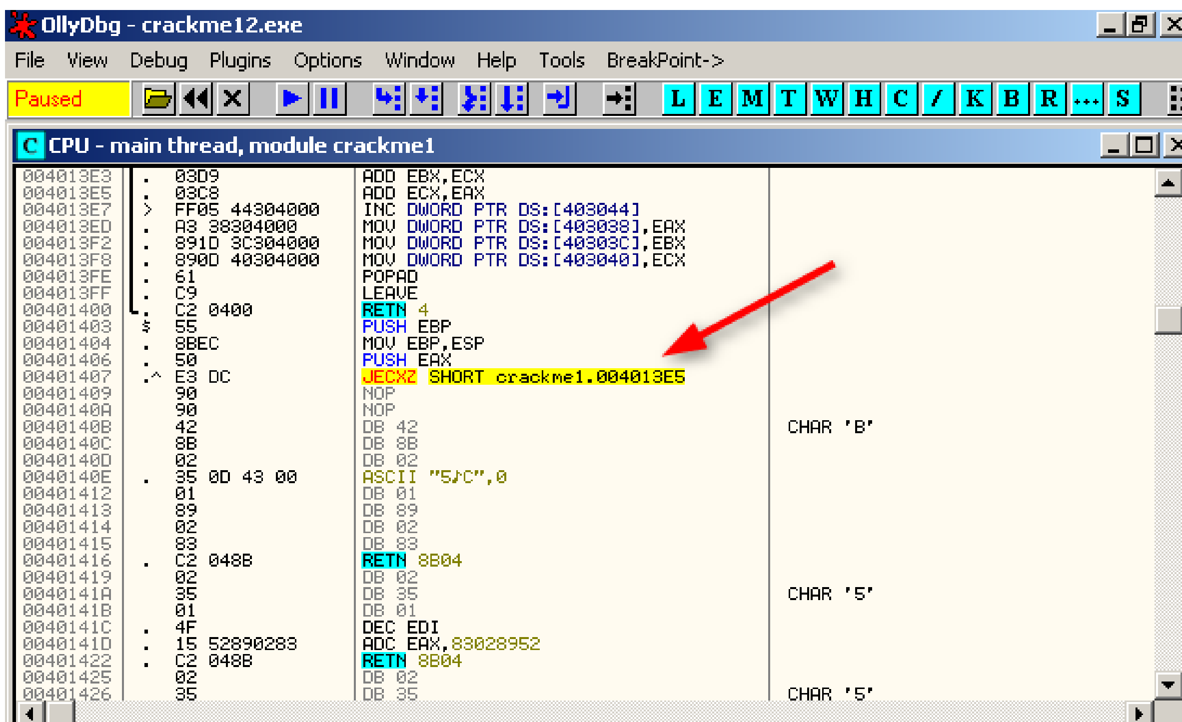
En esta aplicación el último argumento de VirtualProtect es la dirección en memoria a la que queremos cambiarle los atributos y el tercer valor es la longitud en bytes de la sección que queremos cambiar. En este caso podemos ver que la dirección de inicio es 401407 y que la longitud es de 0x1F4 (500). También podemos ver que el segundo argumento es PAGE_READWRITE, haciendo que en esta sección se pueda leer y escribir. Situemonos en esta sección de memoria, empezando por la dirección 401407, para ver lo que va cambiar:

¡No parece ser código! Seguiremos pulsando F8 para ver lo que la aplicación va cambiar en esta sección de la memoria.

Después de pasar del CALL a VirtualProtect, lo primero que hacemos es mover el contenido de la dirección 403038 en memoria a EAX y aplicar XOR con la dirección en memoria 401407, almacenando el resultado de vuelta en 401407. Sabemos que 401407 es la primera dirección de la sección en memoria a la que hemos cambiado los atributos para escritura. Y que 403038 se inició como 0xDEAD pero fue cambiado dependiendo del botón que hemos pulsado (y en que orden). Asi que estas sequencias de instrucciones cambian el espacio

de la memoria basandose en los botones y en el orden en los que son pulsados. Pulsamos F8 hasta llegar a la instrucción JNZ en la dirección 401475 y echemos un vistazo a la dirección 401407:





Vemos que la dirección 401407 ha sido cambiada y ahora tiene una instrucción válida: **JECXZ SHORT crackme1.004013E5.** ¡ La aplicación acaba de añadir un salto condicional a su propio código ! Lo hizo cambiando los opcodes en ese espacio de la memoria.

Volvemos a la instrucción anterior. Vemos que compara el primer byte en 401407 con 0x52 y salta si no es igual, a la dirección 40148F. Podemos ver que el valor del opcode en 401407 es “E3”, que no es igual a 52, asi que saltaremos. El salto es hacia otro VirtualProtect .

```
OllyDbg - crackme12.exe
CPU - main thread, module crackme1
00401463  .  A1 38304000     MOV EAX,DWORD PTR DS:[403038]
00401468  .  3105 07144000   XOR DWORD PTR DS:[401407],EAX
0040146E  .  803D 07144000 52 CMP BYTE PTR DS:[401407],52
00401475  .v 75 18           JNZ SHORT crackme1.0040148F
00401477  .  A1 3C304000     MOV EAX,DWORD PTR DS:[40303C]
0040147C  .  3105 3B144000   XOR DWORD PTR DS:[40143B],EAX
00401482  .  A1 40304000     MOV EAX,DWORD PTR DS:[403040]
00401487  .  3105 3F144000   XOR DWORD PTR DS:[40143F],EAX
0040148D  .v EB 06           JMP SHORT crackme1.00401495
0040148F  >  3105 07144000   XOR DWORD PTR DS:[401407],EAX
00401495  >  68 50304000     PUSH crackme1.00403050          pOldProtect = crackme1.00403050
0040149A  .  6A 10           PUSH 10                         NewProtect = PAGE_EXECUTE
0040149C  .  68 F4010000     PUSH 1F4                        Size = 1F4 (500.)
004014A1  .  68 07144000     PUSH crackme1.00401407          Address = crackme1.00401407
004014A6  .  E8 27000000     CALL <JMP.&KERNEL32.VirtualProtect>  VirtualProtect
004014AB  .  58              POP EAX
004014AC  .  C3              RETN
004014AD     CC              INT3
004014AE  $- FF25 1C204000   JMP DWORD PTR DS:[<&USER32.DestroyWi   USER32.DestroyWindow
004014B4  $- FF25 18204000   JMP DWORD PTR DS:[<&USER32.DialogBoxP  USER32.DialogBoxParamA
004014BA  $- FF25 14204000   JMP DWORD PTR DS:[<&USER32.EndDialog]  USER32.EndDialog
004014C0  $- FF25 10204000   JMP DWORD PTR DS:[<&USER32.SetDlgIte   USER32.SetDlgItemTextA
004014C6  .- FF25 08204000   JMP DWORD PTR DS:[<&KERNEL32.ExitProc  kernel32.ExitProcess
004014CC  $- FF25 04204000   JMP DWORD PTR DS:[<&KERNEL32.GetModu   kernel32.GetModuleHandleA
004014D2  $- FF25 00204000   JMP DWORD PTR DS:[<&KERNEL32.VirtualP  kernel32.VirtualProtect
004014D8     00              DB 00
004014D9     00              DB 00
004014DA     00              DB 00
004014DB     00              DB 00
004014DC     00              DB 00
004014DD     00              DB 00
004014DE     00              DB 00
```

Vemos que se aplicó un XOR en la memoria 401407 de la dirección 40148F, y que la instrucción de la dirección 401407 ha vuelto a cambiar:

```
OllyDbg - crackme12.exe
CPU - main thread, module crackme1
004013E7  >  FF05 44304000   INC DWORD PTR DS:[403044]
004013ED  .  A3 38304000     MOV DWORD PTR DS:[403038],EAX
004013F2  .  891D 3C304000   MOV DWORD PTR DS:[40303C],EBX
004013F8  .  890D 40304000   MOV DWORD PTR DS:[403040],ECX
004013FE  .  61              POPAD
004013FF  .  C9              LEAVE
00401400  .  C2 0400         RETN 4
00401403  $  55              PUSH EBP
00401404  .  8BEC            MOV EBP,ESP
00401406  .  50              PUSH EAX
00401407  .v EB 3F           JMP SHORT crackme1.00401448
00401409     90              NOP
0040140A     90              NOP
0040140B     42              DB 42                           CHAR 'B'
0040140C     8B              DB 8B
0040140D     02              DB 02
0040140E  .  35 0D 43 00     ASCII "5♪C",0
00401412     01              DB 01
00401413     89              DB 89
00401414     02              DB 02
00401415     83              DB 83
00401416  .  C2 048B         RETN 8B04
00401419     02              DB 02
0040141A     35              DB 35                           CHAR '5'
0040141B     01              DB 01
0040141C  .  4F              DEC EDI
0040141D  .  15 52890283     ADC EAX,83028952
00401422  .  C2 048B         RETN 8B04
00401425     02              DB 02
00401426     35              DB 35                           CHAR '5'
00401427     0E              DB 0E
00401428     0D              DB 0D
```

Ahora tenemos un JMP en lugar del JECXZ. La aplicación ha cambiado su propia memoria dos veces. Pulsamos F8 y volvemos a regresar a nuestro loop principal.

A continuación empujamos un valor (F08E2) en la pila y llamamos a otra rutina en la dirección 401403. Pulsamos F7 para entrar en la siguiente función:

Hemos saltado al área de la memoria que habia cambiado la aplicación. Podemos reconozer la nueva instrucción JMP en la dirección 401407. Pulsamos F8 hasta el salto para saber hacia donde nos lleva:

```
OllyDbg - crackme12.exe
File View Debug Plugins Options Window Help Tools BreakPoint->
Paused   L E M T W H C / K B R ... S
CPU - main thread, module crackme1
00401406  .   50              PUSH EAX
00401407  .v  EB 3F           JMP SHORT crackme1.00401448
00401409      90              NOP
0040140A      90              NOP
0040140B      42              DB 42                              CHAR 'B'
0040140C      8B              DB 8B
0040140D      02              DB 02
0040140E  .   35 0D 43 00     ASCII "5♪C",0
00401412      01              DB 01
00401413      89              DB 89
00401414      02              DB 02
00401415      83              DB 83
00401416  .   C2 048B         RETN 8B04
00401419      02              DB 02
0040141A      35              DB 35                              CHAR '5'
0040141B      01              DB 01
0040141C  .   4F              DEC EDI
0040141D  .   15 52890283     ADC EAX,83028952
00401422  .   C2 048B         RETN 8B04
00401425      02              DB 02
00401426      35              DB 35                              CHAR '5'
00401427      0E              DB 0E
00401428      0D              DB 0D
00401429      17              DB 17
0040142A      10              DB 10
0040142B      89              DB 89
0040142C      02              DB 02
0040142D      83              DB 83
0040142E  .   C2 048B         RETN 8B04
00401431      02              DB 02
00401432      35              DB 35                              CHAR '5'
00401433      16              DB 16
```

```
00401434  .   45 45 00        ASCII "EE",0
00401437      89              DB 89
00401438      02              DB 02
00401439  .   5A 58           ASCII "ZX"
0040143B  .   0466E7BB        DD BBE76604
0040143F  .   4DBD088B        DD 8B08BD4D
00401443  .   E8 78000000     CALL <JMP.&USER32.SetDlgItemTextA>    SetDlgItemTextA
00401448  >   C9              LEAVE
00401449  .   C2 0800         RETN 8
0040144C  $   50              PUSH EAX
0040144D  .   68 50304000     PUSH crackme1.00403050                pOldProtect = crackme1.00403050
00401452  .   6A 04           PUSH 4                                NewProtect = PAGE_READWRITE
00401454  .   68 F4010000     PUSH 1F4                              Size = 1F4 (500.)
00401459  .   68 07144000     PUSH crackme1.00401407                Address = crackme1.00401407
0040145E  .   E8 6F000000     CALL <JMP.&KERNEL32.VirtualProtect>   VirtualProtect
00401463  .   A1 38304000     MOV EAX,DWORD PTR DS:[403038]
00401468  .   3105 07144000   XOR DWORD PTR DS:[401407],EAX
0040146E  .   803D 07144000 52 CMP BYTE PTR DS:[401407],52
00401475  .v  75 18           JNZ SHORT crackme1.0040148F
00401477  .   A1 3C304000     MOV EAX,DWORD PTR DS:[40303C]
0040147C  .   3105 3B144000   XOR DWORD PTR DS:[40143B],EAX
00401482  .   A1 40304000     MOV EAX,DWORD PTR DS:[403040]
00401487  .   3105 3F144000   XOR DWORD PTR DS:[40143F],EAX
0040148D  .v  EB 06           JMP SHORT crackme1.00401495
0040148F  >   3105 07144000   XOR DWORD PTR DS:[401407],EAX
00401495  >   68 50304000     PUSH crackme1.00403050                pOldProtect = crackme1.00403050
0040149A  .   6A 10           PUSH 10                               NewProtect = PAGE_EXECUTE
0040149C  .   68 F4010000     PUSH 1F4                              Size = 1F4 (500.)
004014A1  .   68 07144000     PUSH crackme1.00401407                Address = crackme1.00401407
004014A6  .   E8 27000000     CALL <JMP.&KERNEL32.VirtualProtect>   VirtualProtect
004014AB  .   58              POP EAX
004014AC  .   C3              RETN
```

Está saltando hacia un regreso. Parece que no va hacer nada especial por lo que volveremos a nuestro programa principal.

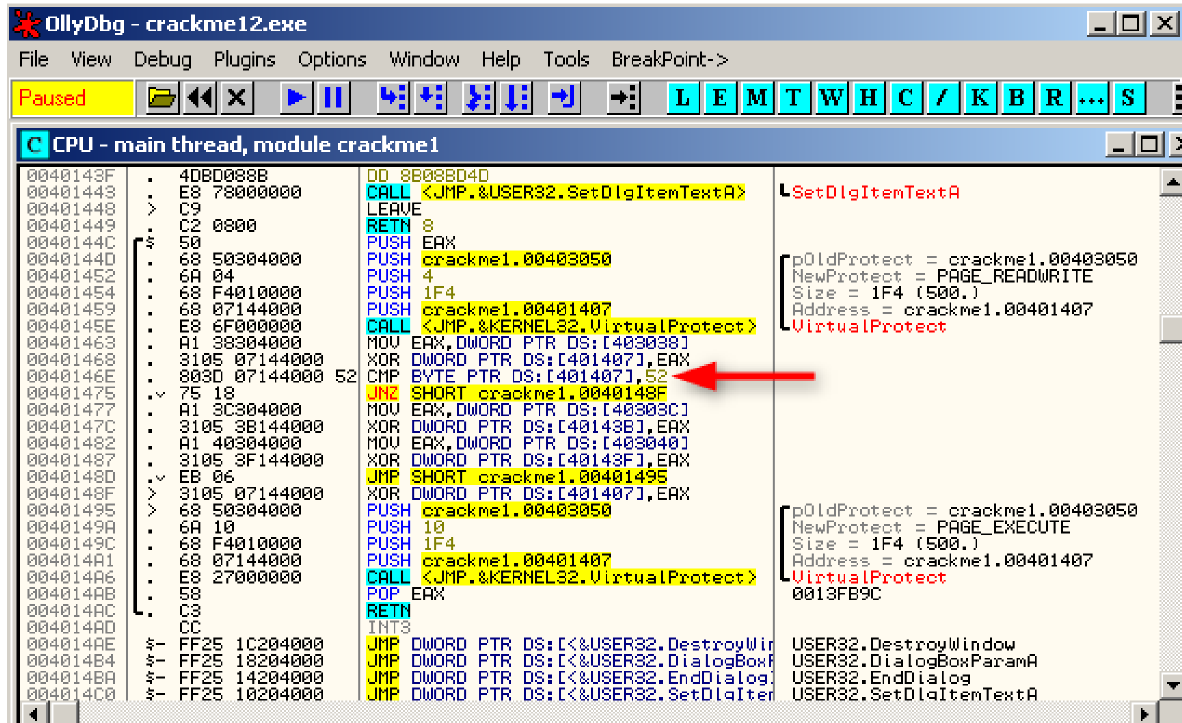
```
OllyDbg - crackme12.exe
File  View  Debug  Plugins  Options  Window  Help  Tools  BreakPoint->
Paused
CPU - main thread, module crackme1
004011F2  >   833D 48304000 03 CMP DWORD PTR DS:[403048],3
004011F9  .v  73 2D            JNB SHORT crackme1.00401228
004011FB  .   833D 44304000 0A CMP DWORD PTR DS:[403044],0A
00401202  .v  75 3D            JNZ SHORT crackme1.00401241
00401204  .   E8 43020000      CALL crackme1.0040144C
00401209  .   68 00304000      PUSH crackme1.00403000           ASCII "An error occured"
0040120E  .   FF75 08          PUSH [ARG.1]
00401211  .   E8 ED010000      CALL crackme1.00401403
00401216  .   C705 44304000 00 MOV DWORD PTR DS:[403044],0
00401220  .   FF05 48304000    INC DWORD PTR DS:[403048]
00401226  .v  EB 19            JMP SHORT crackme1.00401241
00401228  >   68 11304000      PUSH crackme1.00403011           Text = "Trying to bruteforce?"
0040122D  .   6A 03            PUSH 3                           ControlID = 3
0040122F  .   FF75 08          PUSH [ARG.1]                     hWnd = 000C00A6 ('Crackme#12 by D
00401232  .   E8 89020000      CALL <JMP.&USER32.SetDlgItemTextA> SetDlgItemTextA
00401237  .   C705 44304000 00 MOV DWORD PTR DS:[403044],0
00401241  >v  EB 09            JMP SHORT crackme1.0040124C
00401243  >   B8 00000000      MOV EAX,0
00401248  .   C9               LEAVE
00401249  .   C2 1000          RETN 10
0040124C  >   B8 01000000      MOV EAX,1
00401251  .   C9               LEAVE
00401252  .   C2 1000          RETN 10
00401255  .   55               PUSH EBP
00401256  .   8BEC             MOV EBP,ESP
00401258  .   837D 0C 10       CMP [ARG.2],10
0040125C  .v  75 0C            JNZ SHORT crackme1.0040126A
0040125E  .   6A 01            PUSH 1                           Result = 1
00401260  .   FF75 08          PUSH [ARG.1]                     hWnd = 000C00A6 ('Crackme#12 by D
00401263  .   E8 52020000      CALL <JMP.&USER32.EndDialog>     EndDialog
00401268  .v  EB 32            JMP SHORT crackme1.0040129C
0040126A  >   817D 0C 11010000 CMP [ARG.2],111
```

Lo siguiente que haremos es resetear nuestro contador de 0x0A a cero. Luego incrementamos nuestro contador en uno para la comprobación de la fuerza bruta. Ahora ya sabemos como funciona la fuerza bruta: si introducimos un dígito erroneo (de diez caracteres) más de tres veces, (el contenido de la dirección 403048 va a estar por encima de tres) saltaremos al mensaje de la fuerza bruta.

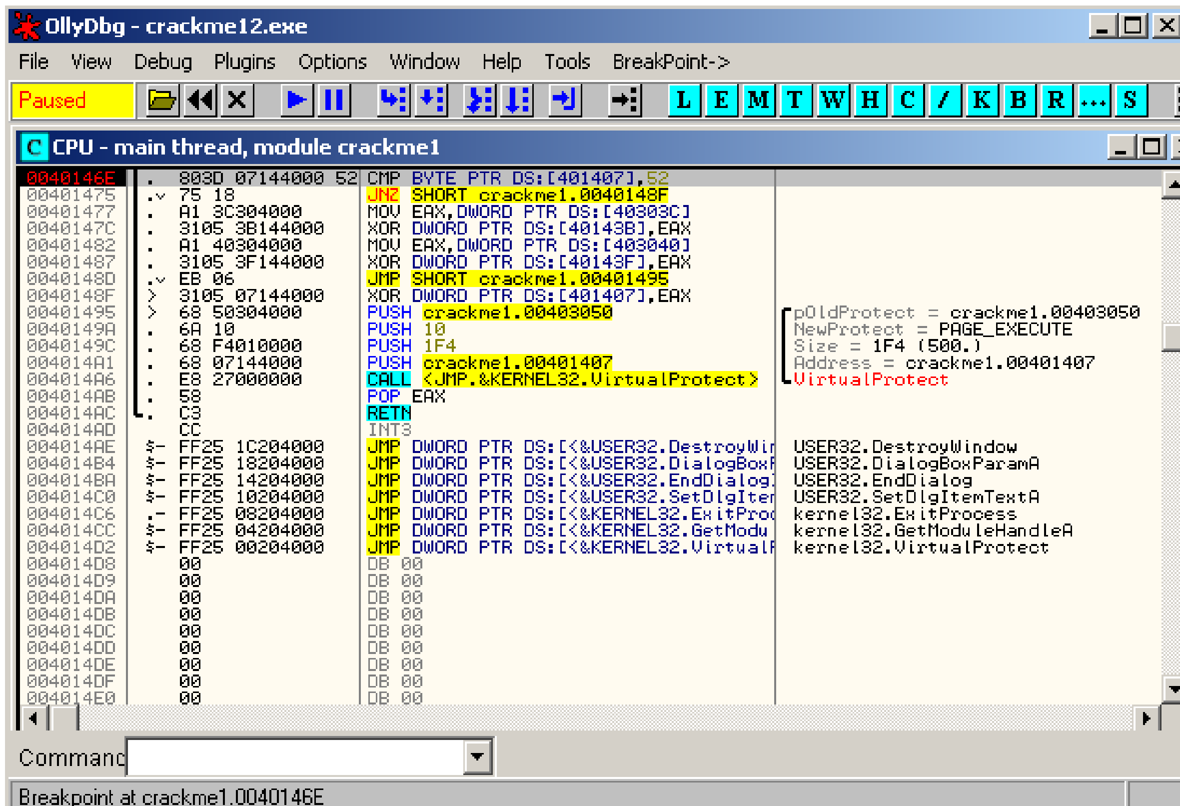
Resumiendo, lo que tenemos hasta ahora:

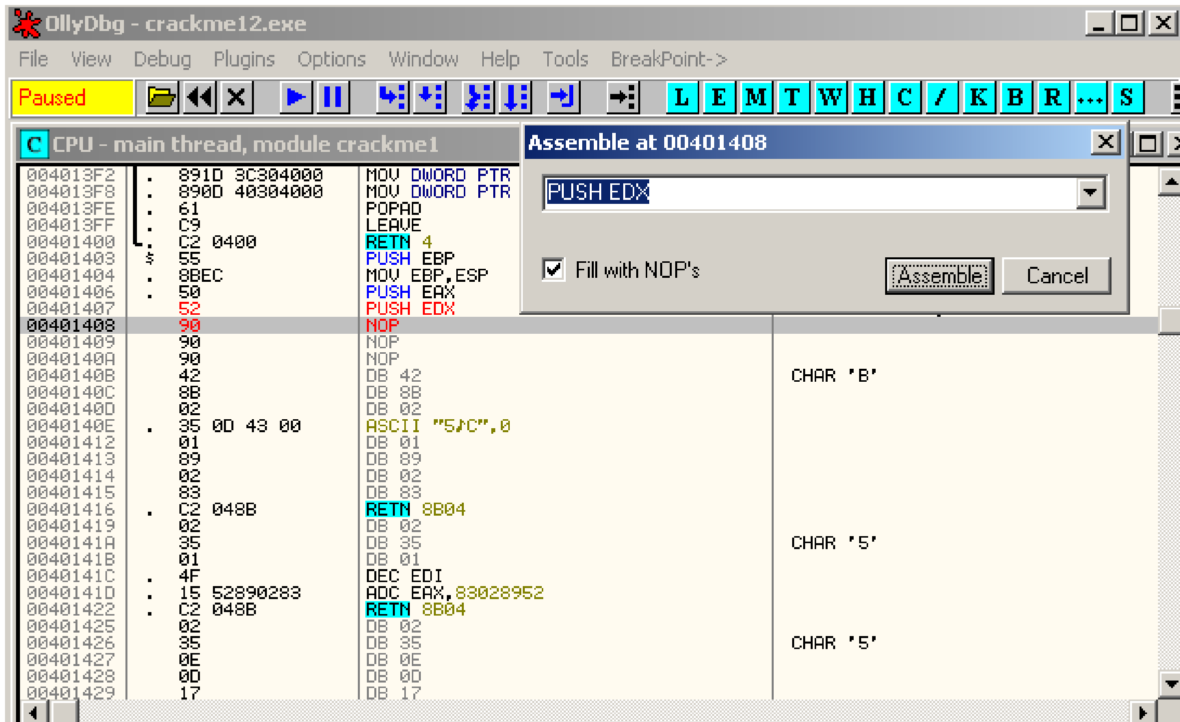
1. La contraseña tiene 10 digitos.
2. Si introducimos una contraseña erronea más de tres veces obtendremos el mensaje de la fuerza bruta y tendremos que reiniciar la aplicación.
3. Cada vez que pulsamos un botón, las áreas de la memoria 403038, 40303C y 403040 se modificarán de forma distinta según el botón que hayamos pulsado.
4. Después de pulsar diez botones, entraremos en unos cuantos CALL’s para modificar una instrucción de salto en la sección de código que corresponde a la dirección en memoria 401407.
5. Si la contraseña no es correcta, el salto que ha sido creado apuntará hacia un regreso que nos llevará de vuelta a nuesro loop principal.
6. Si introducimos la contraseña correcta, cambiaremos ese salto a otra cosa distinta. Bien a un salto a un area distinto de la memoria donde podría estar nuestro “good boy” o haciendo más cambios en nuestro código de esa área para crear el “good boy” en esa sección de la memoria en lugar de crear un salto.

Ahora sabemos que tenemos que poner ceros en la sección de código que realiza las auto modificaciones del código, es decir el código que comienza en la dirección 40144C. Volvemos echar un vistazo a esa sección:



Podemos concluir que la intstrucción de comparación con 0x52 en la dirección 40146E es bastante importante. Le dice al programa que los cambios que hicimos en el código son los correctos. ¿Pero que significa el opcode 0x52? Significa: PUSH EDX. Así que este código lo que hace es comprobar si la primera instrucción es un “PUSH EDX”, y si no lo es, lo descarta. ¿Qué pasaría si forzamos la instrucción para que sea un “PUSH EDX”? Pongamos un Breakpoint en la dirección 40146E donde el código comprueba si se trata de una instrucción PUSH y pulsamos F9. Cuando Olly se detiene en el Breakpoint iremos a la dirección 401407 para cambiar el valor a 0x52:





Si pulsamos ahora F8 vemos que no saltamos:

Ahora, lo que hace el código es mover el valor de la dirección 40303C a EAX y aplica XOR con el area de la memoria 40143B. Miremos lo que hay en esa dirección:

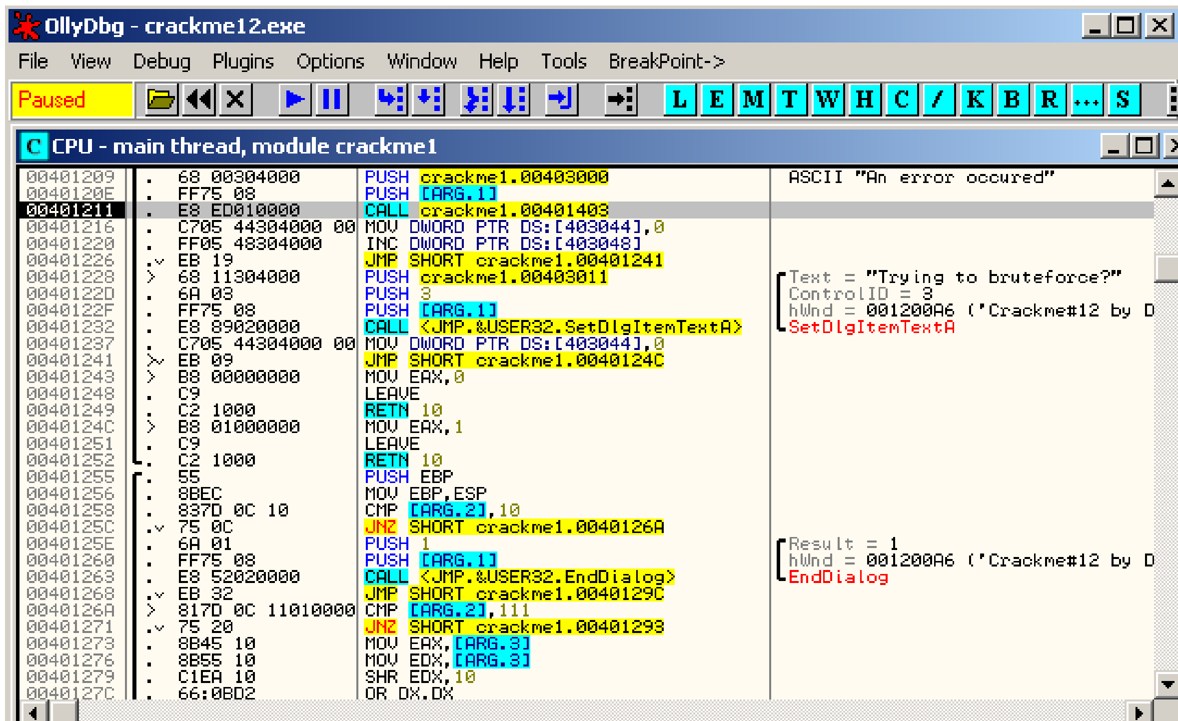
Solo se trata de un area de la memoria hacia el final de la sección de nuestro código auto modificado. Después de aplicar XOR tenemos que:

```
0040143B  .  046628DD        DD DD286604
0040143F  .  4DBD08BB        DD 8B08BD4D
00401443  .  E8 78000000     CALL <JMP.&USER32.SetDlgItemTextA>     SetDlgItemTextA
00401448  >  C9              LEAVE
00401449  .  C2 0800         RETN 8
0040144C  $  50              PUSH EAX
0040144D  .  68 50304000     PUSH crackme1.00403050                 pOldProtect = crackme1.00403050
00401452  .  6A 04           PUSH 4                                 NewProtect = PAGE_READWRITE
00401454  .  68 F4010000     PUSH 1F4                               Size = 1F4 (500.)
00401459  .  68 07144000     PUSH crackme1.00401407                 Address = crackme1.00401407
0040145E  .  E8 6F000000     CALL <JMP.&KERNEL32.VirtualProtect>    VirtualProtect
00401463  .  A1 38304000     MOV EAX,DWORD PTR DS:[403038]
00401468  .  3105 07144000   XOR DWORD PTR DS:[401407],EAX
0040146E  .  803D 07144000 52 CMP BYTE PTR DS:[401407],52
00401475  .v 75 18           JNZ SHORT crackme1.0040148F
00401477  .  A1 3C304000     MOV EAX,DWORD PTR DS:[40303C]
0040147C  .  3105 3B144000   XOR DWORD PTR DS:[40143B],EAX
00401482  .  A1 40304000     MOV EAX,DWORD PTR DS:[403040]
```

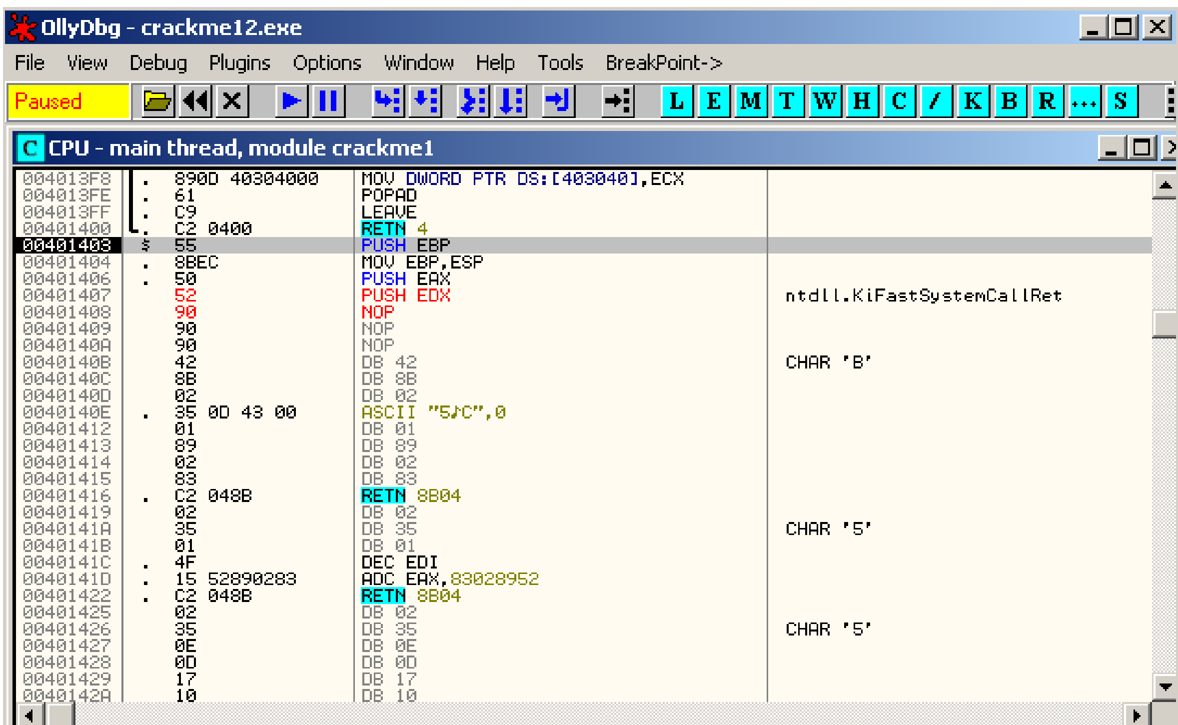
Y cambiamos la siguiente localización en 40143F aplicando XOR con el contenido en 403040:

```
OllyDbg - crackme12.exe
File  View  Debug  Plugins  Options  Window  Help  Tools  BreakPoint->
Paused
CPU - main thread, module crackme1

00401433     16              DB 16
00401434  .  45 45 00        ASCII "EE",0
00401437     89              DB 89
00401438  .  02              DB 02
00401439  .  5A 58           ASCII "ZX"
0040143B  .  046628DD        DD DD286604
0040143F  .  7DCA4AC9        DD C94ACA7D
00401443  .  E8 78000000     CALL <JMP.&USER32.SetDlgItemTextA>     SetDlgItemTextA
00401448  >  C9              LEAVE
00401449  .  C2 0800         RETN 8
0040144C  $  50              PUSH EAX
0040144D  .  68 50304000     PUSH crackme1.00403050                 pOldProtect = crackme1.00403050
00401452  .  6A 04           PUSH 4                                 NewProtect = PAGE_READWRITE
00401454  .  68 F4010000     PUSH 1F4                               Size = 1F4 (500.)
00401459  .  68 07144000     PUSH crackme1.00401407                 Address = crackme1.00401407
0040145E  .  E8 6F000000     CALL <JMP.&KERNEL32.VirtualProtect>    VirtualProtect
00401463  .  A1 38304000     MOV EAX,DWORD PTR DS:[403038]
00401468  .  3105 07144000   XOR DWORD PTR DS:[401407],EAX
0040146E  .  803D 07144000 52 CMP BYTE PTR DS:[401407],52
00401475  .v 75 18           JNZ SHORT crackme1.0040148F
00401477  .  A1 3C304000     MOV EAX,DWORD PTR DS:[40303C]
0040147C  .  3105 3B144000   XOR DWORD PTR DS:[40143B],EAX
00401482  .  A1 40304000     MOV EAX,DWORD PTR DS:[403040]
00401487  .  3105 3F144000   XOR DWORD PTR DS:[40143F],EAX
0040148D  .v EB 06           JMP SHORT crackme1.00401495
0040148F  >  3105 07144000   XOR DWORD PTR DS:[401407],EAX
00401495  >  68 50304000     PUSH crackme1.00403050                 pOldProtect = crackme1.00403050
0040149A  .  6A 10           PUSH 10                                NewProtect = PAGE_EXECUTE
0040149C  .  68 F4010000     PUSH 1F4                               Size = 1F4 (500.)
004014A1  .  68 07144000     PUSH crackme1.00401407                 Address = crackme1.00401407
004014A6  .  E8 27000000     CALL <JMP.&KERNEL32.VirtualProtect>    VirtualProtect
004014AB  .  58              POP EAX
```

Ahora que hemos cambiado la instrucción a PUSH EDX y visto lo que hace esta sección de código, volvamos hacia el loop principal para entrar en el CALL de la dirección 401211:



Pulsamos F7.

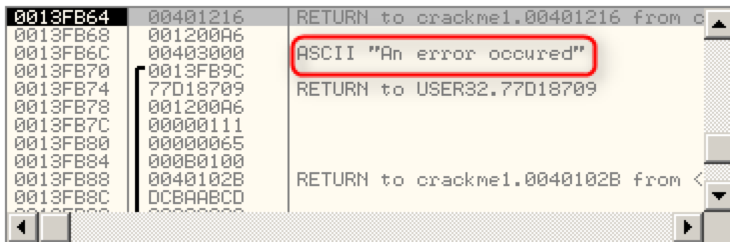


Nos encontramos al principio de la sección auto modificada, empezando por PUSH EDX. Vamos a decirle a Olly que las cosas han cambiado y que re-analize esta sección de código:

Volvemos a encontrarnos con algo que se parece a código legible:

Intentemos analizar el código en su conjunto. Tenemos un PUSH EDX al principio. Esto, junto al POP EDX del final nos indica que EDX va a ser utilizado en en este código de forma local. Tenemos NOP's vacíos que probablemente tendrán algún código. También hay una serie de áreas en la memoria a los que se aplican XOR con DWORDS. Esto normalmete significa que estamos descifrando algo, y en este caso es algo a lo que EDX está apuntando. Los NOP's probablemente son el lugar donde se encuentra EDX y EDX apuntará a algo que será descifrado u alterado con los XOR's.

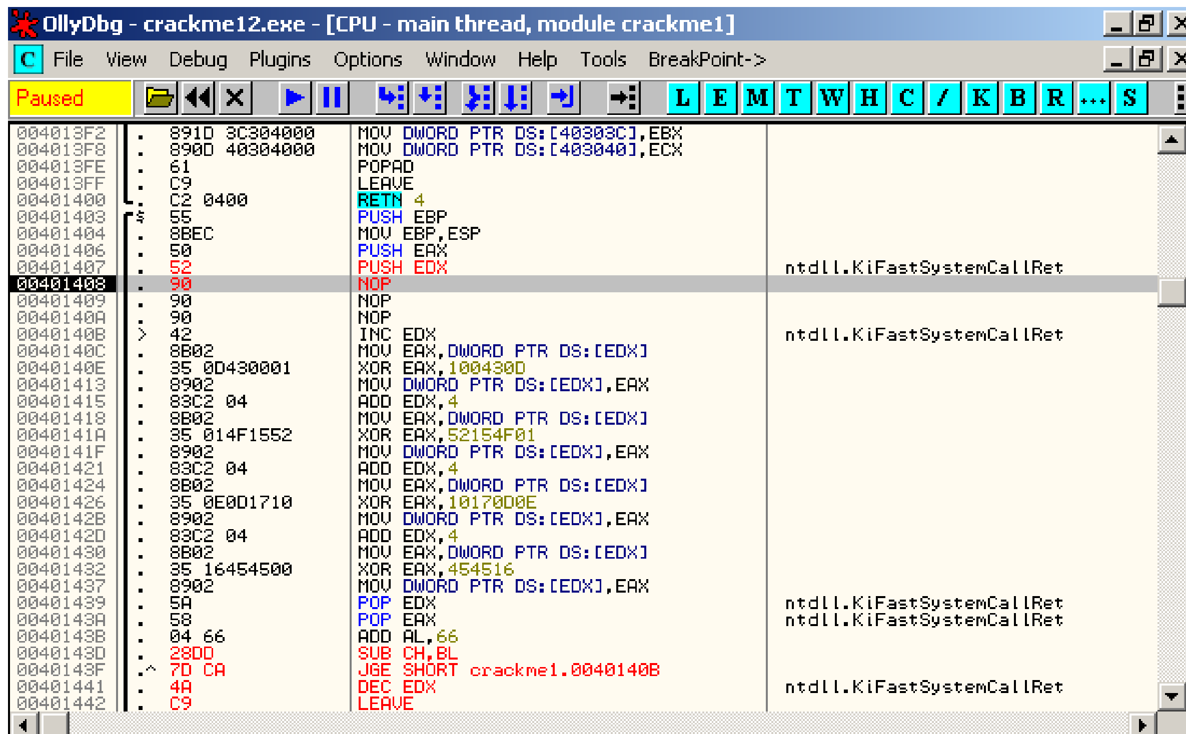
La gran pregunta en estos momentos es hacia donde apunta EDX. Si recordamos, había una cadena de texto que nunco se llego a utilizar y que viene a decir lo siguiente:"An error occured". Quizas es esto lo que va a ser descifrado…Sabemos que la cadena fue empujado a la pila, pero nunca se llego a utilizar.



Veamos si podemos confirmar nuestra hipotesis. Generalmente cargamos una variable local con una instrucción como la siguiente:

MOV EDX, [EBP + algo_#] or MOV EDX, [EBP - algo_#]

Para averiguar el número vamos a pusar F8 hasta llegar a la dirección 401408:

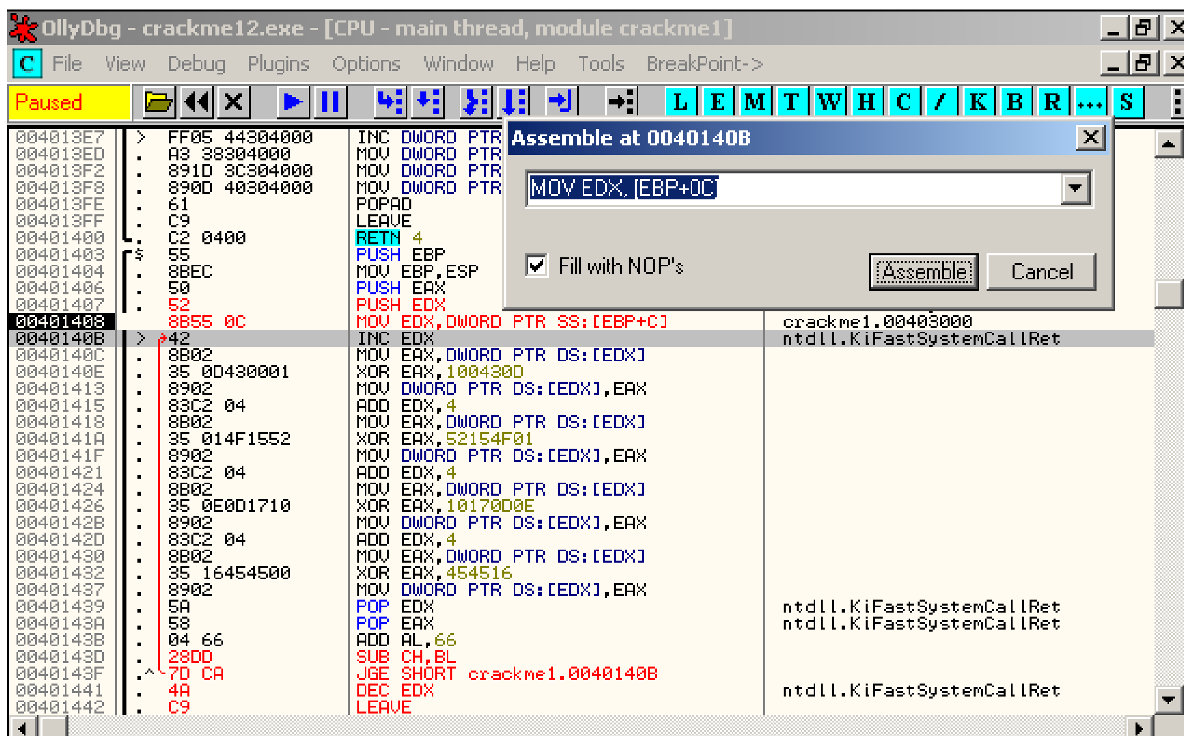


Si nos fijamos en los registros vemos que EBP apunta a 13FB60 y que la cadena de error está 12 bits más arriba que EBP (más abajo en la pila).

Asi que la insrucción que cargaría un puntero hacia la cadena de error podría ser:

MOV EDX, [EBP + 0x0C]

Intentemoslo y miremos a ver cuantos bytes necesita:



Parece que coje bien. Pulsamos F8 para ver lo que pasa a continuación. Primero, en la dirección 401408, EDX es cargado con un puntero hacia nuestro texto:

```
Registers (FPU)                                 <
EAX 00000065
ECX 0013FB10
EDX 00403000 ASCII "An error occured"
EBX 00000000
ESP 0013FB58
EBP 0013FB60
ESI 0040102B crackme1.0040102B
EDI 0013FBD8

EIP 0040140B crackme1.0040140B

C 0  ES 0023 32bit 0(FFFFFFFF)
P 0  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDE000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_ACCESS_DENIED (00000005)

EFL 00000202 (NO,NB,NE,A,NS,PO,GE,G)

ST0 empty -4.4095550575196474850e-1178
ST1 empty 0.0000000000000023290e-4933
ST2 empty +UNORM 1F80 000B0100 00000020
ST3 empty 6.3146436768772478200e-4932
ST4 empty 1.6276338743205395520e+771
ST5 empty +UNORM 00A6 00000020 000B0100
ST6 empty -1.6653055059025684530e+1621
ST7 empty 0.5263546642232484440e-4933
               3 2 1 0      E S P U O Z D I
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0 0
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1
```

A continuación EDX se incrementa para apuntar al segundo carácter de nuestra cadena (‘n’). Cuatro bytes (un dword) son cargados dentro de EAX comenzando en la ‘n’ de “A**n** error…”. Despues se aplica XOR con EAX y 0x100430D lo que da como resultado 0x73656363. Este valor es guardado en la dirección donde se encuentra la cadena de error (403000). Podemos ver nuestra cadena antes de ser almacenada:

```
Address  Hex dump                 ASCII
00403000 41 6E 20 65 72 72 6F 72  An error
00403008 20 6F 63 63 75 72 65 64   occured
00403010 00 54 72 79 69 6E 67 20  .Trying
00403018 74 6F 20 62 72 75 74 65  to brute
00403020 66 6F 72 63 65 3F 00 00  force?..
00403028 00 00 40 00 00 00 00 00  ..@.....
00403030 00 00 00 00 00 00 00 00  ........
00403038 08 E3 00 00 00 00 CF 66  ■ã....Ïf
00403040 30 77 42 42 0A 00 00 00  0wBB....
00403048 00 00 00 00 00 30 40 00  .....0@.
```

Y después de ser almacenada:

```
Address  Hex dump                 ASCII
00403000 41 63 63 65 73 72 6F 72  Accesror
00403008 20 6F 63 63 75 72 65 64   occured
00403010 00 54 72 79 69 6E 67 20  .Trying
00403018 74 6F 20 62 72 75 74 65  to brute
00403020 66 6F 72 63 65 3F 00 00  force?..
00403028 00 00 40 00 00 00 00 00  ..@.....
00403030 00 00 00 00 00 00 00 00  ........
00403038 08 E3 00 00 00 00 CF 66  ■ã....Ïf
00403040 30 77 42 42 0A 00 00 00  0wBB....
00403048 00 00 00 00 00 30 40 00  .....0@.
```

Nuestra cadena ha sido modificada. Continuemos: cargamos los cuatro bytes siguientes, XOR con 0x52154F01, y los volvemos guardar en memoria. Lo que convierte nuestra cadena en:

```
Address  Hex dump                 ASCII
00403000 41 63 63 65 73 73 20 67  Access g
00403008 72 6F 63 63 75 72 65 64  roccured
00403010 00 54 72 79 69 6E 67 20  .Trying
00403018 74 6F 20 62 72 75 74 65  to brute
00403020 66 6F 72 63 65 3F 00 00  force?..
00403028 00 00 40 00 00 00 00 00  ..@.....
00403030 00 00 00 00 00 00 00 00  ........
00403038 08 E3 00 00 00 00 CF 66  ■Ð....Ïf
00403040 30 77 42 42 0A 00 00 00  0wBB....
00403048 00 00 00 00 00 30 40 00  .....0@.
```

Volvemos a pulsar F8 hasta alcanzar nuestro siguiente bit de código lo que nos dará los siguientes cuatro bytes:

```
Address  Hex dump                 ASCII
00403000 41 63 63 65 73 73 20 67  Access g
00403008 72 61 6E 74 65 72 65 64  rantered
00403010 00 54 72 79 69 6E 67 20  .Trying
00403018 74 6F 20 62 72 75 74 65  to brute
00403020 66 6F 72 63 65 3F 00 00  force?..
00403028 00 00 40 00 00 00 00 00  ..@.....
00403030 00 00 00 00 00 00 00 00  ........
00403038 08 E3 00 00 00 00 CF 66  ■Ð....Ïf
00403040 30 77 42 42 0A 00 00 00  0wBB....
00403048 00 00 00 00 00 30 40 00  .....0@.
```

Finalmente obtenemos la cadena entera: "Acces granted":

```
Address  Hex dump                 ASCII
00403000 41 63 63 65 73 73 20 67  Access g
00403008 72 61 6E 74 65 64 20 21  ranted !
00403010 00 54 72 79 69 6E 67 20  .Trying
00403018 74 6F 20 62 72 75 74 65  to brute
00403020 66 6F 72 63 65 3F 00 00  force?..
00403028 00 00 40 00 00 00 00 00  ..@.....
00403030 00 00 00 00 00 00 00 00  ........
00403038 08 E3 00 00 00 00 CF 66  ■Ð....Ïf
00403040 30 77 42 42 0A 00 00 00  0wBB....
00403048 00 00 00 00 00 30 40 00  .....0@.
```

¡Hemos acertado con nuestra hipotesis! Sabemos el contenido de la cadena, el problema que tenemos ahora es que, como hemos introducido una contraseña erronea y las últimas declaraciones han sido descifradas incorrectamente, nuestro mensaje nunca aparecerá. Lo que tenemos que hacer es reconstruir los argumentos que van ser empujados hacia la pila para el SetDigItemTextA. Con la ayuda de Olly sabemos que SetDigItemTextA necesita tres argumentos para ser empujados a la pila:

LPCTSTR lpString // text to set

int nIDDlgItem, // identifier of control

HWND hDlg, // handle of dialog box

El primero es facil:

PUSH [EBP + 0x0c]

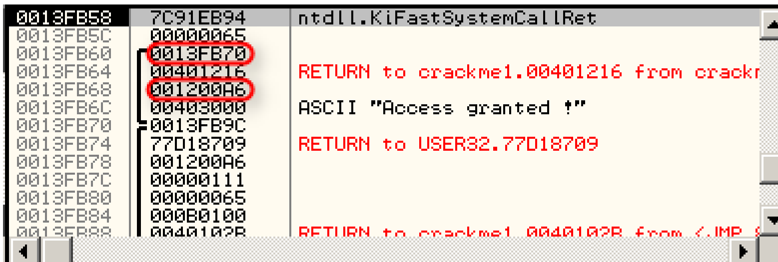
Es el que apunta a nuestra nueva cadena de texto. El segundo y el tercer son algo más dificiles, pero afortunadamente podemos contar con referencias. Hay un SetDigItemTextA cuando aparece el mensaje de la fuerza bruta:

```
00401226  .v EB 19              JMP SHORT crackme1.00401241
00401228  >  68 11304000        PUSH crackme1.00403011                 ┌Text = "Trying to bruteforce?"
0040122D  .  6A 03              PUSH 3                                 │ControlID = 3
0040122F  .  FF75 08            PUSH [ARG.1]                           │hWnd = 001200A6 ('Crackme#12 by Det
00401232  .  E8 89020000        CALL <JMP.&USER32.SetDlgItemTextA>     └SetDlgItemTextA
00401237  .  C705 44304000 00   MOV DWORD PTR DS:[403044],0
00401241  >v EB 09              JMP SHORT crackme1.0040124C
```

Podemos ver que el ControlID es igual a tres y el controlador de la ventana es 1200A6. El ControlID es facil:

PUSH 3

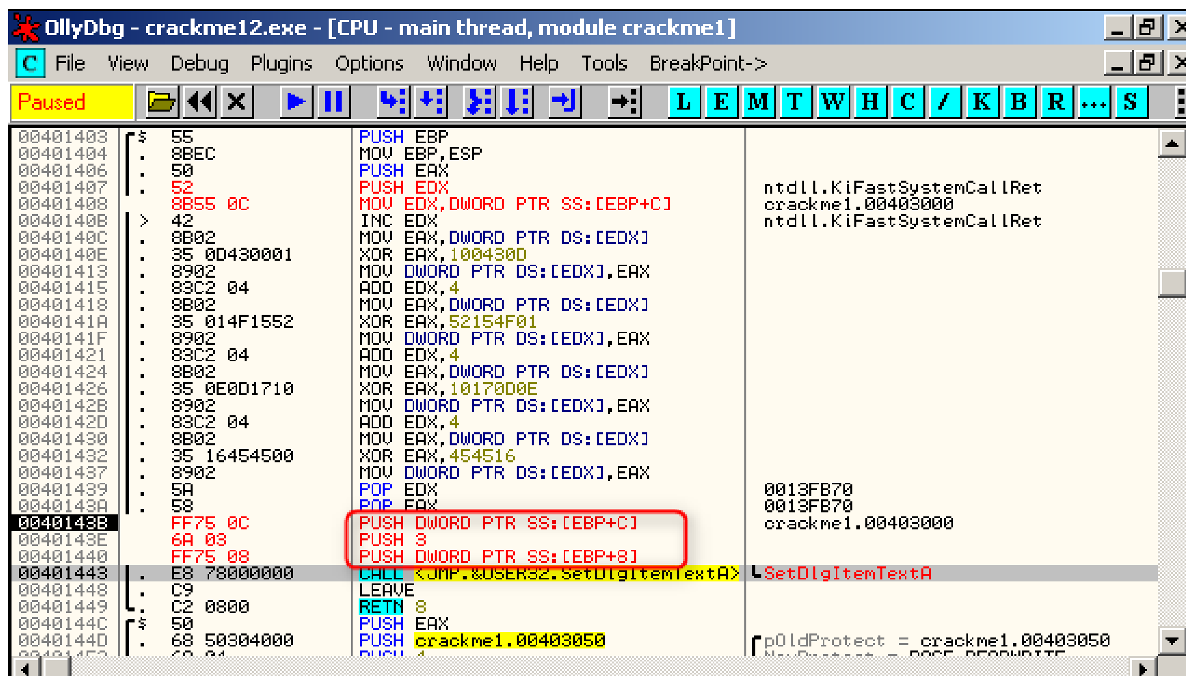
Para el controlador miramos en la pila:



Vemos que el controlador (1200A6) es igual a EBP+8. Asi que tenemos:

PUSH DWORD PTR [EBP + 8]

El código final quedaría de al siguiente manera:



Ejecutamos la aplicación:

Guardando el binario con nuestros parches hará posible introducir cualquier contraseña con 10 digitos y asi obtener siempre el "good boy".

7.15 Caso práctico 15: Fuerza bruta

La fuerza bruta es una forma de extraer un serial o una constraseña de un binario si sabemos el input y el output de una rutina de cifrado / descifrado, pero no sabemos como parchear el software o no tenemos el tiempo suficiente para hacerlo. Se trata de la diferencia entre un software crackeado con un parche (o una copia del ejecutable parcheado) y un software que ya viene con un nombre de usuario y serial que funcione. Si alguna vez hemos descargado un software crackeado que incluye el nombre de usuario y el serial, probablemente se utilizó la técnica de fuerza bruta.

Sabiendo pues el input y el output de la rutina a cifrar/descifrar, se intentarán todas las posibilidades que conviertan al input en el output hasta que uno coincida. Por ejemplo, si introducimos un serial de 1212121212 y la aplicación envía esto a la rutina de descifrado para que lo compare con "j6^^gD7-L". Lo que intentaremos averiguar es como se convierte el serial introducido en esa cadena de caracteres, y que serial tenemos que introducir para que la aplcación nos registre.

Nota: *Este método solo funciona en binarios que contengan el nombre de usuarios y sus contraseñas, pero no en aquellas aplicaciónes que esten conectados a una base de datos online.*

Para realizar el ejercicio seguiremos utilizando el crackme12.exe de ejercicios anteriores asi como un programa de fuerza bruta.

A continuación se muestra un listado de las operaciones que se realizan una vez pulsado cualquier botón del crackme:

```
004012A9 mov ecx, dword_403040            ; variable 'a'
004012AF mov ebx, dword_40303C            ; variable 'b'
004012B5 mov eax, dword_403038            ; variable 'c'
004012BA cmp [ebp+arg_0], 1               ; ***** Button 1
004012BE jnz short loc_4012D0
004012C0 add ecx, 54Bh                    ; c += 54Bh
004012C6 imul ebx, eax                    ; b *= a
004012C9 xor eax, ecx                     ; a^= c
004012CB jmp loc_4013E7

004012D0 cmp [ebp+arg_0], 2               ; ***** Button 2
004012D4 jnz short loc_4012E8
004012D6 sub ecx, 233h                    ; c -= 233h
004012DC imul ebx, 14h                    ; b *= 14h
004012DF add ecx, eax                     ; c += a
004012E1 and ebx, eax                     ; b &= a
004012E3 jmp loc_4013E7

004012E8 cmp [ebp+arg_0], 3               ; ***** Button 3
004012EC jnz short loc_4012FD
004012EE add eax, 582h                    ; a += 582h
004012F3 imul ecx, 16h                    ; c *= 16h
```

```
004012F6 xor ebx, eax                     ; b ^= a
004012F8 jmp loc_4013E7

004012FD cmp [ebp+arg_0], 4               ; ***** Button 4
00401301 jnz short loc_401312
00401303 and eax, ebx                     ; a &= b
00401305 sub ebx, 111222h                 ; b -= 111222h
0040130B xor ecx, eax                     ; c ^= a
0040130D jmp loc_4013E7

00401312 cmp [ebp+arg_0], 5               ; ***** Button 5
00401316 jnz short loc_401324
00401318 cdq
00401319 idiv ecx                         ; a /= c, division rest -->
0040131B sub ebx, edx                     ; b -= r
0040131D add eax, ecx                     ; a += c
0040131F jmp loc_4013E7

00401324 cmp [ebp+arg_0], 6               ; ***** Button 6
00401328 jnz short loc_401339
0040132A xor eax, ecx                     ; a ^= c
0040132C and ebx, eax                     ; b &= a
0040132E add ecx, 546879h                 ; c += 546879h
00401334 jmp loc_4013E7

00401339 cmp [ebp+arg_0], 7               ; ***** Button 7
0040133D jnz short loc_401351
0040133F sub ecx, 25FF5h                  ; c -= 25FF5h
00401345 xor ebx, ecx                     ; b ^= c
00401347 add eax, 401000h                 ; a += 401000h
0040134C jmp loc_4013E7
```

```
00401351 cmp [ebp+arg_0], 8                 ; ***** Button 8
00401355 jnz short loc_401367
00401357 xor eax, ecx                       ; a ^= c
00401359 imul ebx, 14h                      ; b *= 14h
0040135C add ecx, 12589h                    ; c += 12589h
00401362 jmp loc_4013E7

00401367 cmp [ebp+arg_0], 9                 ; ***** Button 9
0040136B jnz short loc_401378
0040136D sub eax, 542187h                   ; a -= 542187h
00401372 sub ebx, eax                       ; b -= a
00401374 xor ecx, eax                       ; c ^= a
00401376 jmp short loc_4013E7

00401378 cmp [ebp+arg_0], 0Ah               ; ***** Button 10
0040137C jnz short loc_40138A
0040137E cdq
0040137F idiv ebx                           ; a /= b, division rest -->
00401381 add ebx, edx                       ; b += r
00401383 imul eax, edx                      ; a *= r
00401386 xor ecx, edx                       ; c ^= r
00401388 jmp short loc_4013E7

0040138A cmp [ebp+arg_0], 0Bh               ; ***** Button 11
0040138E jnz short loc_4013A3
00401390 add ebx, 1234FEh                   ; b += 1234FEh
00401396 add ecx, 2345DEh                   ; c += 2345DEh
0040139C add eax, 9CA4439Bh                 ; a += 9CA4439Bh
004013A1 jmp short loc_4013E7
```

```
004013A3 cmp [ebp+arg_0], 0Ch               ; ***** Button 12
004013A7 jnz short loc_4013B2
004013A9 xor eax, ebx                       ; a ^= b
004013AB sub ebx, ecx                       ; b -= c
004013AD imul ecx, 12h                      ; c *= 12h
004013B0 jmp short loc_4013E7

004013B2 cmp [ebp+arg_0], 0Dh               ; ***** Button 13
004013B6 jnz short loc_4013C8
004013B8 and eax, 12345678h                 ; a &= 12345678h
004013BD sub ecx, 65875h                    ; c -= 65875h
004013C3 imul ebx, ecx                      ; b *= c
004013C6 jmp short loc_4013E7

004013C8 cmp [ebp+arg_0], 0Eh               ; ***** Button 14
004013CC jnz short loc_4013DB
004013CE xor eax, 55555h                    ; a ^= 55555h
004013D3 sub ebx, 587351h                   ; b -= 587351h
004013D9 jmp short loc_4013E7

004013DB cmp [ebp+arg_0], 0Fh               ; ***** Button 15
004013DF jnz short loc_4013E7
004013E1 add eax, ebx                       ; a += b
004013E3 add ebx, ecx                       ; b += c
004013E5 add ecx, eax                       ; c += a
```

Lo siguiente que necesitamos son los inputs y outputs. Si recordamos la sección que contenía el código auto modificable empezaba con una cosa, y luego después de aplicarle la instrucción XOR se convertía una instrucción legitima.

Asi que el input es el código antes de aplicarle el XOR y el output son las mismas areas despues de aplicarle el XOR y ser modificado contra ‘a’, ‘b’ y ‘c’.

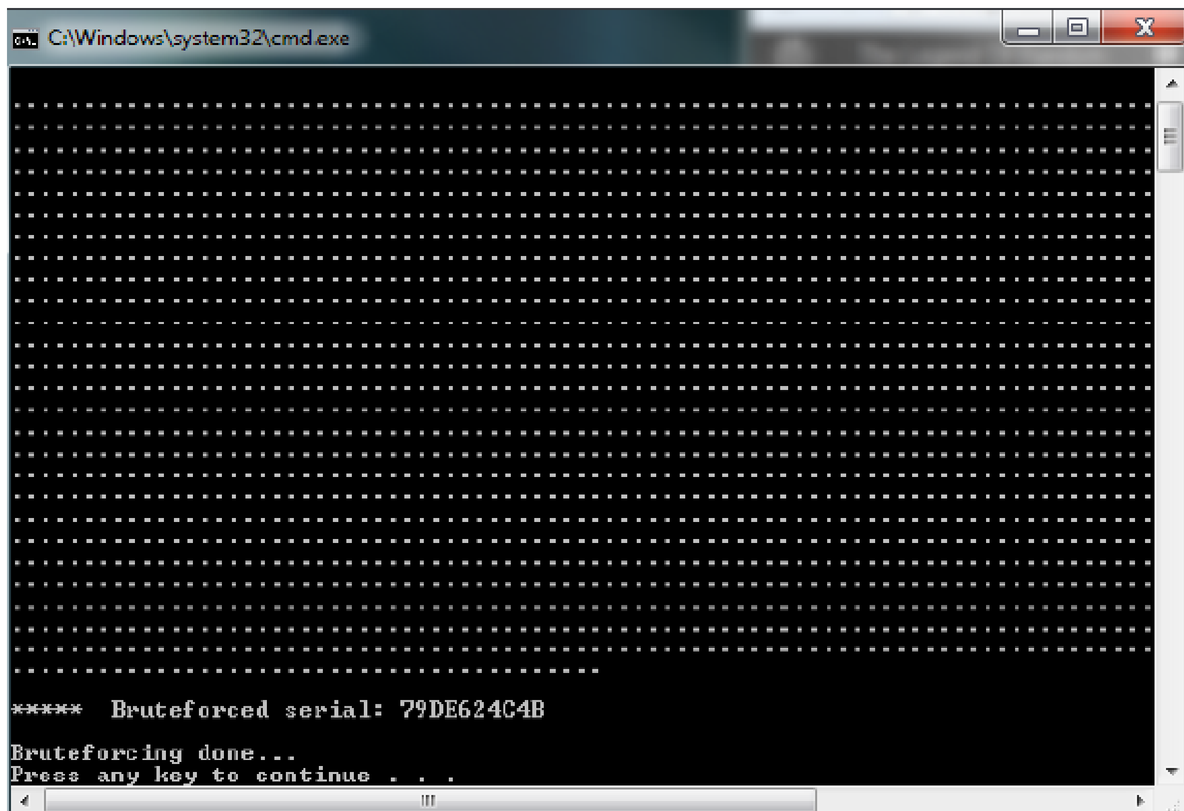
Dirección 401407 empieza como EB 3F 90 90 y se convierte en B9 B4 C5 9C después del XOR con ‘a’.

Dirección 40143B empieza como 04 66 E7 BB y se convierte en FF 75 0C 6A después del XOR con ‘b’.

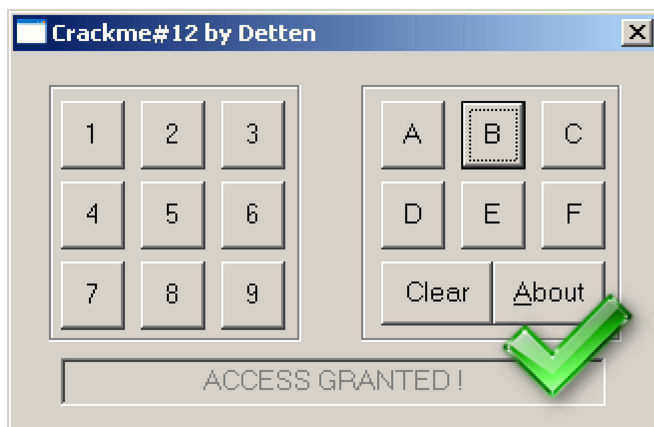
Dirección 40143F empieza como 4D BD 08 8B y se convierte en 03 FF 75 08 después del XOR con ‘c’

Para ver todas las combinaciones posibles utilizaremos el bruteforcer escrito en C cuyo código fuente se adjunta a este ejercicio (hemos incluido los dos primeros caracteres de la contraseña: ‘7’ y ‘9’ para agilizar el proceso. Un ordenador con ocho nucleos y sin ningún tipo de información previa necesitará alrededor de una hora para crackear el serial).

Nuesto output después de ejecutar el bruteforcer en la consola es el siguiente:



Introducimos el serial:



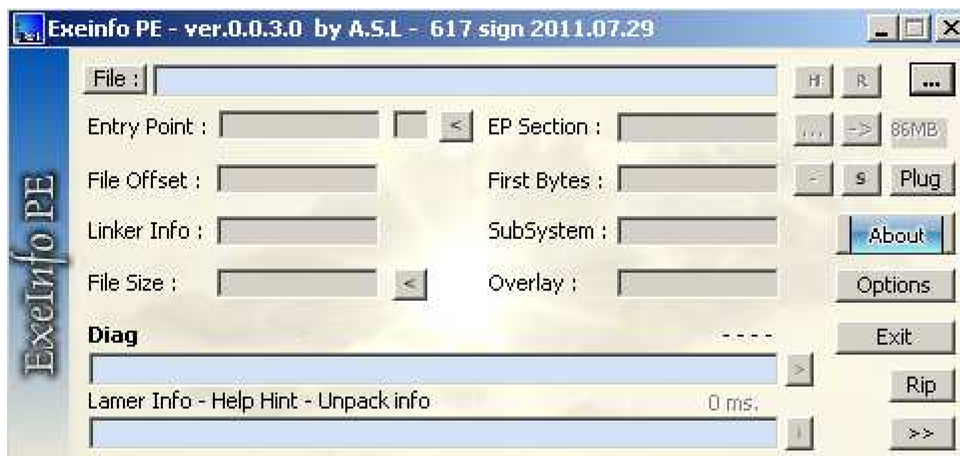
7.16 Caso práctico 16: Los binarios de delphi

Delphi es un lenguaje de programación que se creó con el propósito de agilizar la creación de software basándolo en una programación visual. En Delphi se utiliza una versión más actual del Pascal conocida como Object Pascal como lenguaje de programación.Delphi es un lenguaje muy versátil se usa para casi cualquier proyecto como por ejemplo servicios del sistema operativo, establecer comunicación entre un servidor web y un programa, aplicaciones de consola, conectividad con bases de datos, para realizar aplicaciones visuales, etc. Este lenguaje produce aplicaciones en código máquina, por lo que la computadora las interpreta inmediatamente y no precisa de un lenguaje interprete como es necesario en otros lenguajes de programación.

Ventajas del lenguaje Delphi:

- En cualquiera de sus versiones se pueden programar DLLs**.**
- En Delphi podemos programar directamente los componentes visuales e incluso crear nuevos controles que hereden características de los ya existentes.
- También podemos utilizar en Delphi componentes visuales de otros lenguajes de programación.
- Delphi utiliza Object Pascal, que es un lenguaje de programación orientado a objetos, lo que nos permite beneficiarnos de características importantes en programación como son: el encapsulamiento, polimorfismo y la herencia.

Una de las primeras preguntas que nos podemos plantear es:¿ Como sé que estoy tratando con un programa en Delphi ? Para responder a esta pregunta contamos con la ayuda del programa ExeInfoPE. Este programa se usa generalmente para averiguar que empaquetador se ha utilizado para empaquetar un binario. Si el programa no ha sido empaquetado también nos dice el lenguaje en el cual fue escrito el programa. Después de iniciar ExeInfoPE aparece la siguiente ventana de inicio:

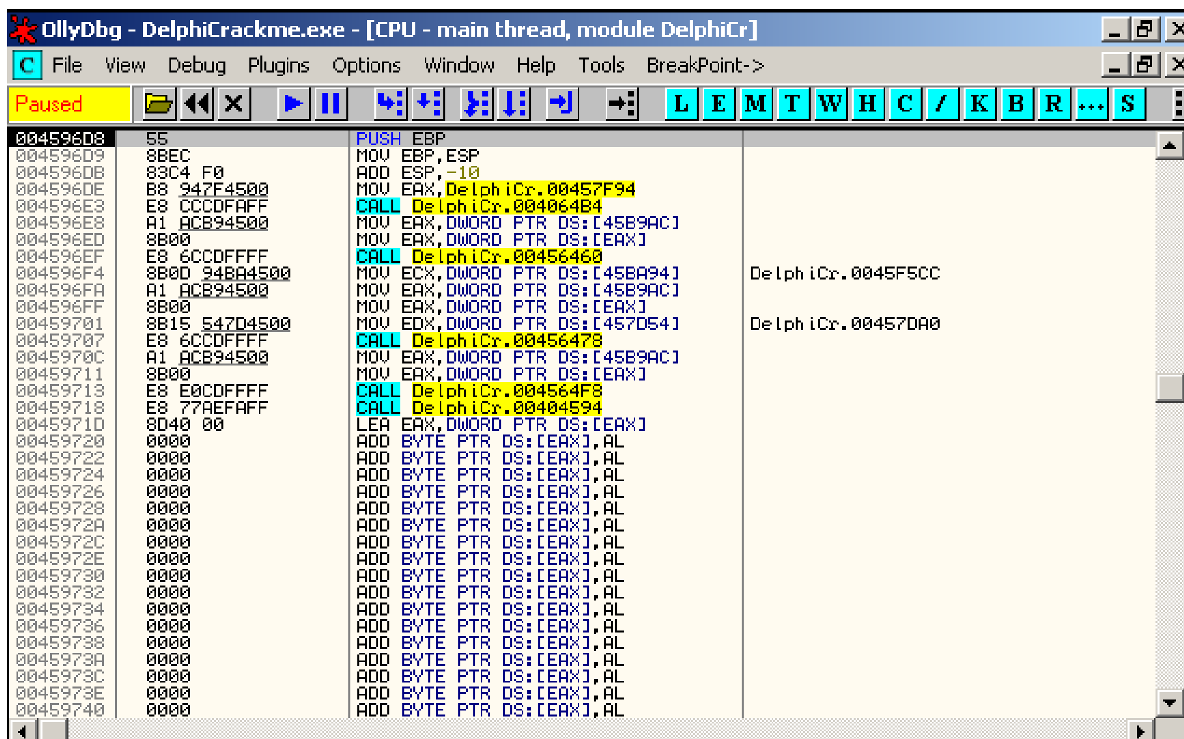


7.16.1 DelphiCrackme.exe

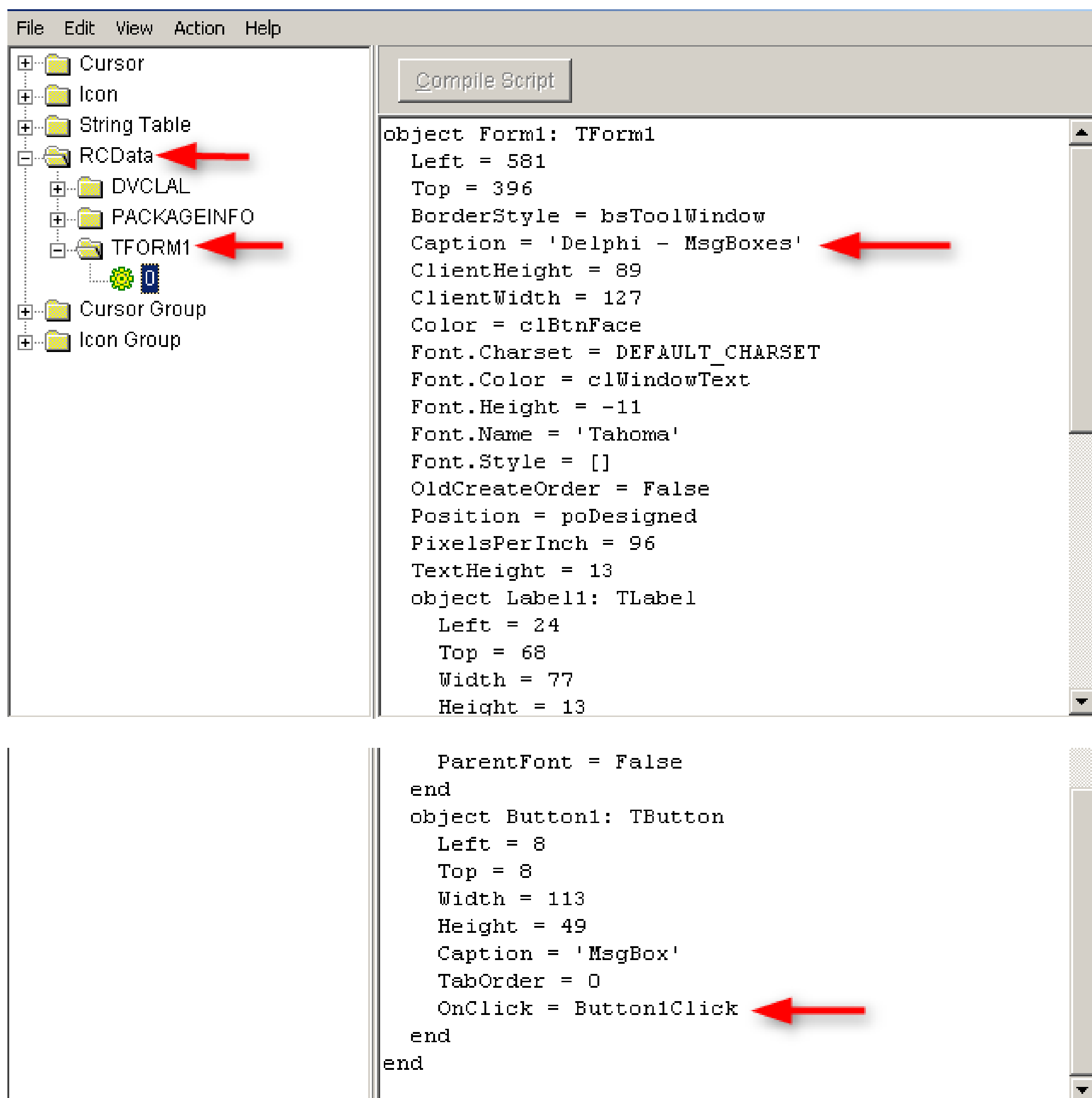
Cargamos nuestro primer programa DelphiCrackme.exe

La información que nos muestra Exeinfo PE sobre el programa DelphiCrackme.exe, es que ha sido compilado en Delphi. También podemos ver que la aplicación no ha sido empaquetada.

Asimismo, tampronto cargamos la aplicación en Olly vemos que estamo ante un programa “distinto” a los demás programas con los que hemos trabajado hasta ahora:



Una de las diferencias más importantes a la hora de trabajar con programas Delphi son sus recursos. Si cargamos la aplicación en Resource Hacker veremos una nueva carpeta llamada RCDATA. Y dentro de ella nos encontramos con la sección de recursos asociado a este binario.



Dentro de RCDATA cabe destacar la sección TFORM, donde se encuentran los cuadros de dialogo/ventanas de los programas Delphi. En el crackme que estamos estudiando, podemos ver que hay un formulario, TFORM1. Si hacemos clic en la flor que lleva dentro se abrirá en Resource Hacker el área principal de los datos correspondientes a esta sección. Estos datos nos dicen todo sobre el formulario; el tamaño, el color, la ubicación en la pantalla, el título, … resumiendo todos los campos y botones que lleva incorporado.

En la sección de ‘Caption’ encontraremos el título de la ventana. En nuestro caso es “Delphi - MsgBoxes”. Este campo es importante en aquellas aplicaciones que tienen más de un formulario. Con el título de la ventana podemos saber que formulario corresponde a que ventana.

También es de cierta relevancia el objeto botones que aparece al final. Es importante porque en nuestro analisis deseamos ‘atrapar’ la aplicación cuando pulsamos un boton, ya sea el de ‘OK’, ‘Exit’, ‘Registrar’, ‘Aceptar’, etc. Es importante que nos fijemos en el nombre del botón para el método OnClick. En nuestro caso es “Button1Click”. Como los programas de Delphi conectan todo con nombres en ASCII, cuando la aplicación vaya a ejecutar el codigo

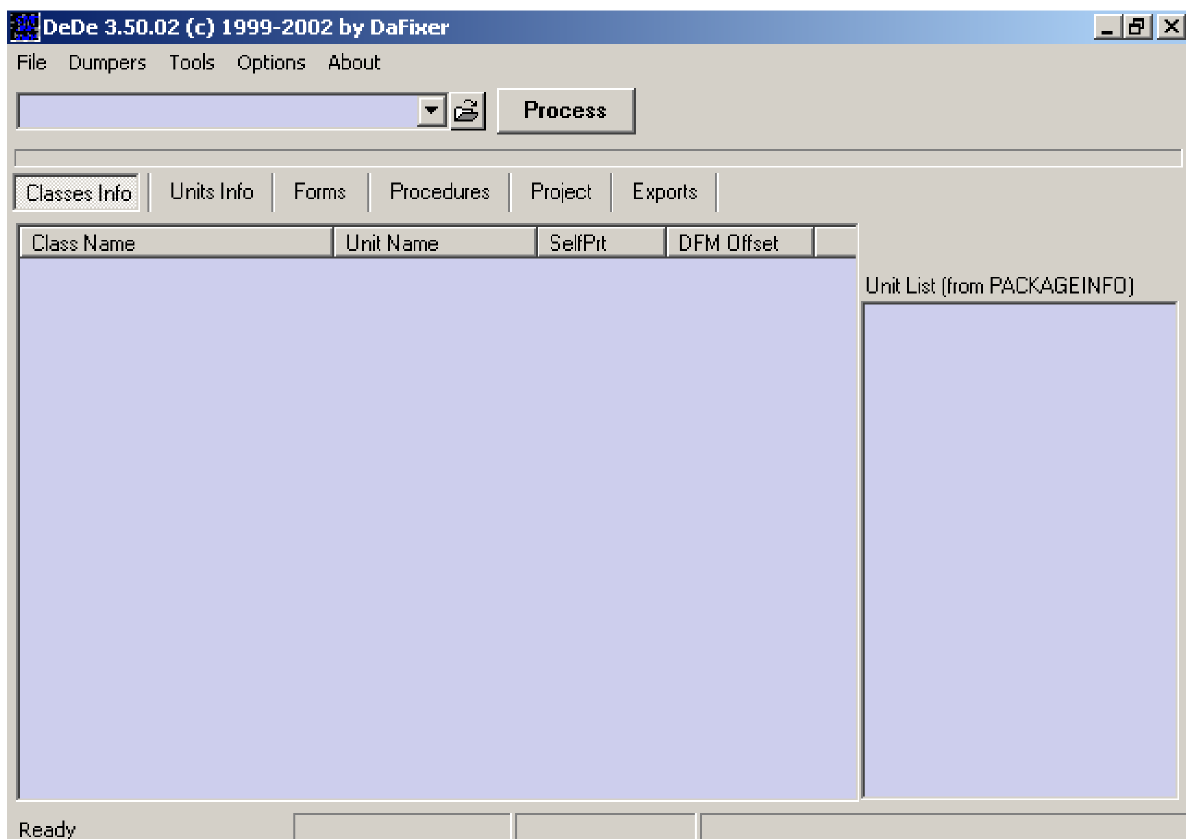
asociado con hacer clic en un botón, buscará el nombre “Button1Click” para encontrar el método.

Resumiendo la información del Resource Hacker, tenemos un formulario (ventana) con un botón. El “Caption” de la ventana es “Delphi - MsgBoxes” y la función de devolución de llamada que controla el clic del botón se llama “Button1Click”.

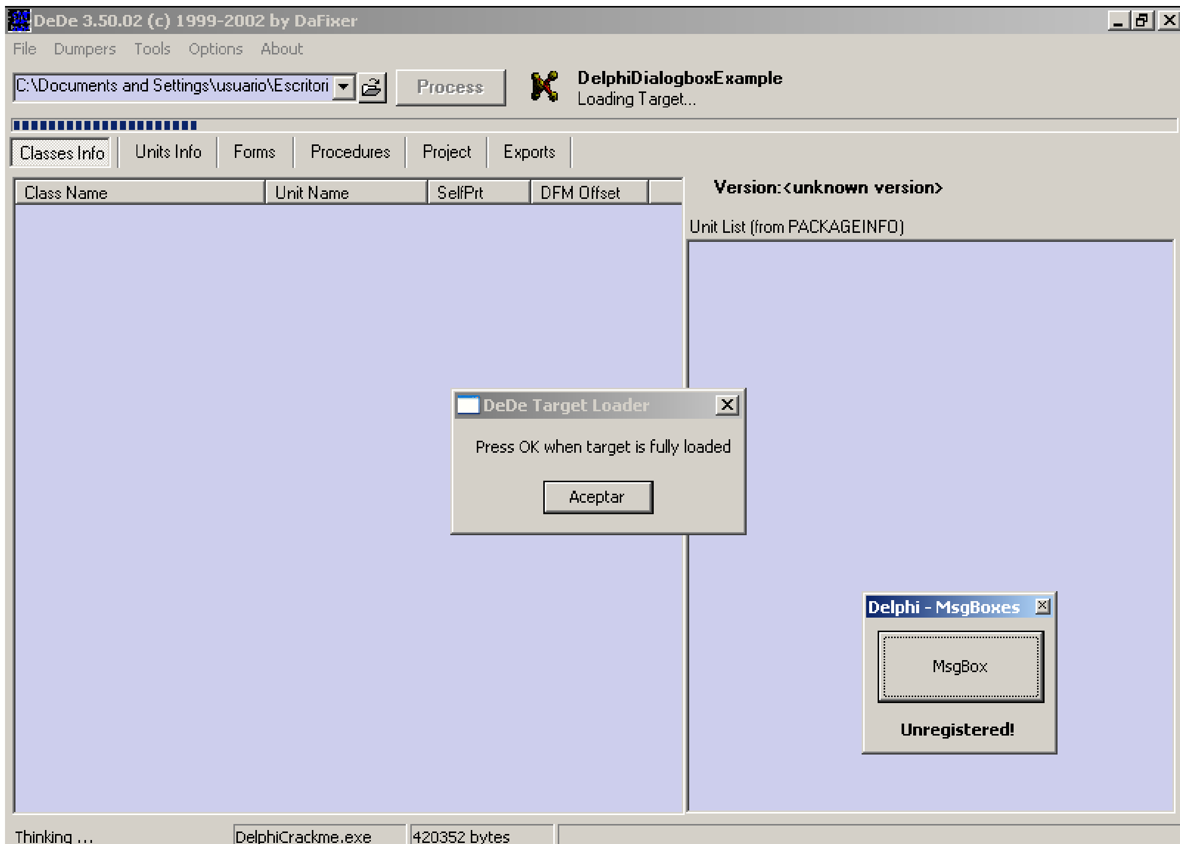
Veamos a continuación una de las herramientas más importantes a la hora de trabajar con programas Delphi: el decompilador de delphi (DeDe).

El decompilador de delphi carga un programa de delphi y lo descompone, enseñandonos todos los datos del formulario que hemos visto. También nos muestra desde donde son llamados todos los métodos, las direcciones de todos los métodos, y el nombre de los métodos. También es capaz de mostrarnos una completa decompilación del binario, así como las posibilidades de modificarlo.

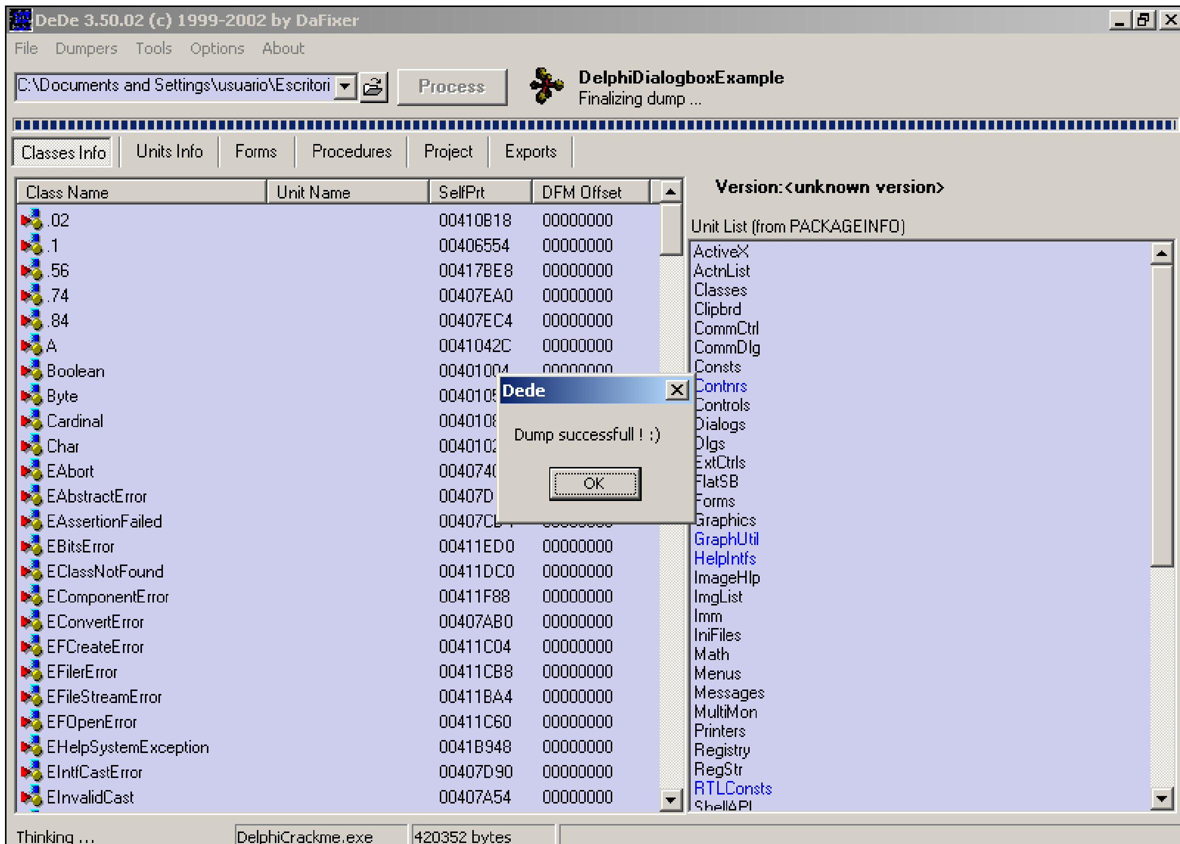
A continuación vemos la ventana principal despues de ejecutar DeDe.



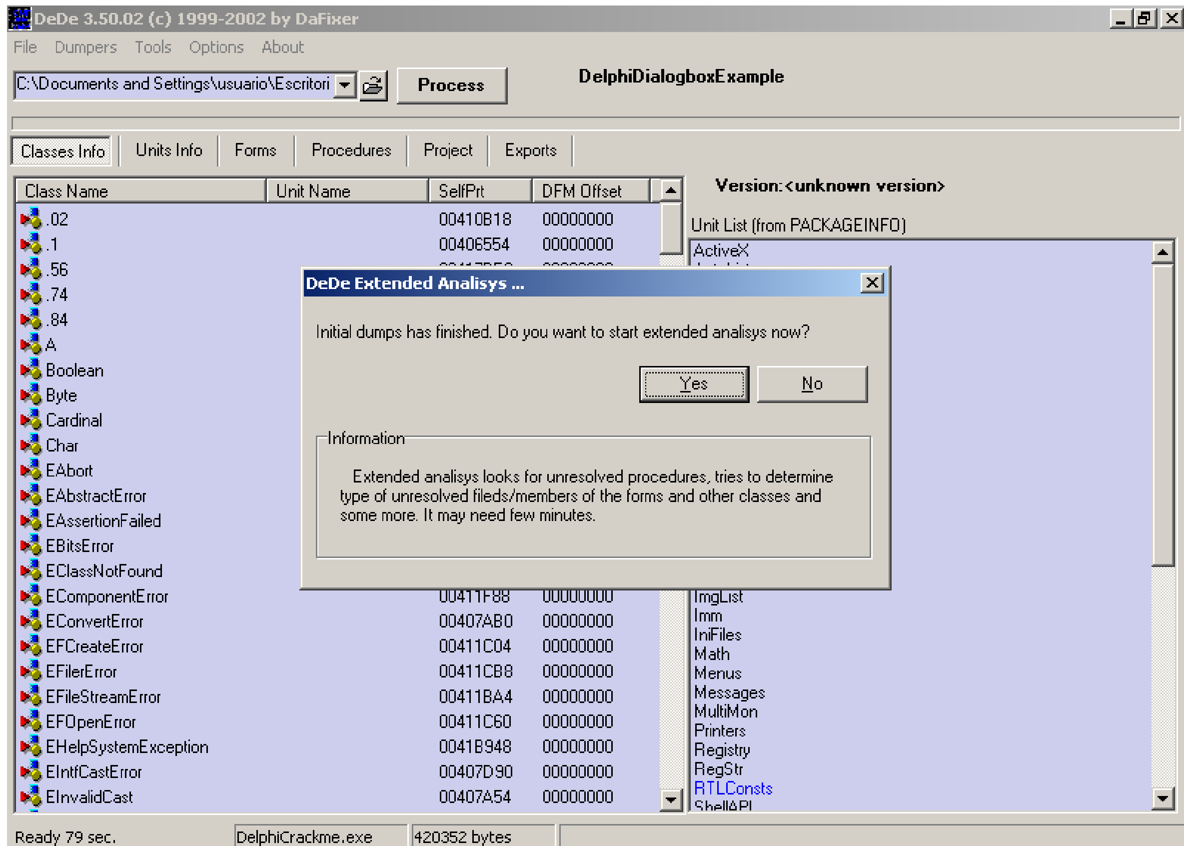
Cargamos nuestro DelphiCrackme.exe en DeDe y hacemos clic en “Process”:



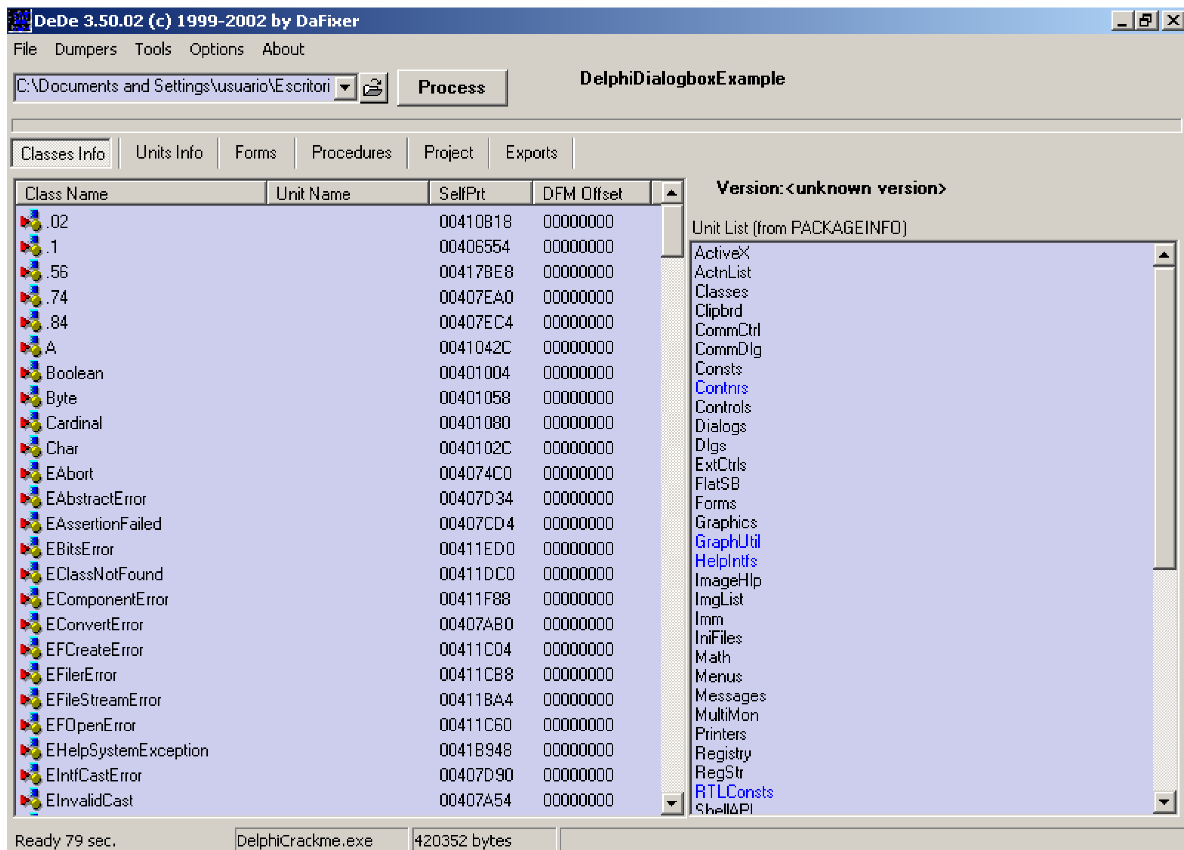
Hacemos clic en Aceptar y en OK



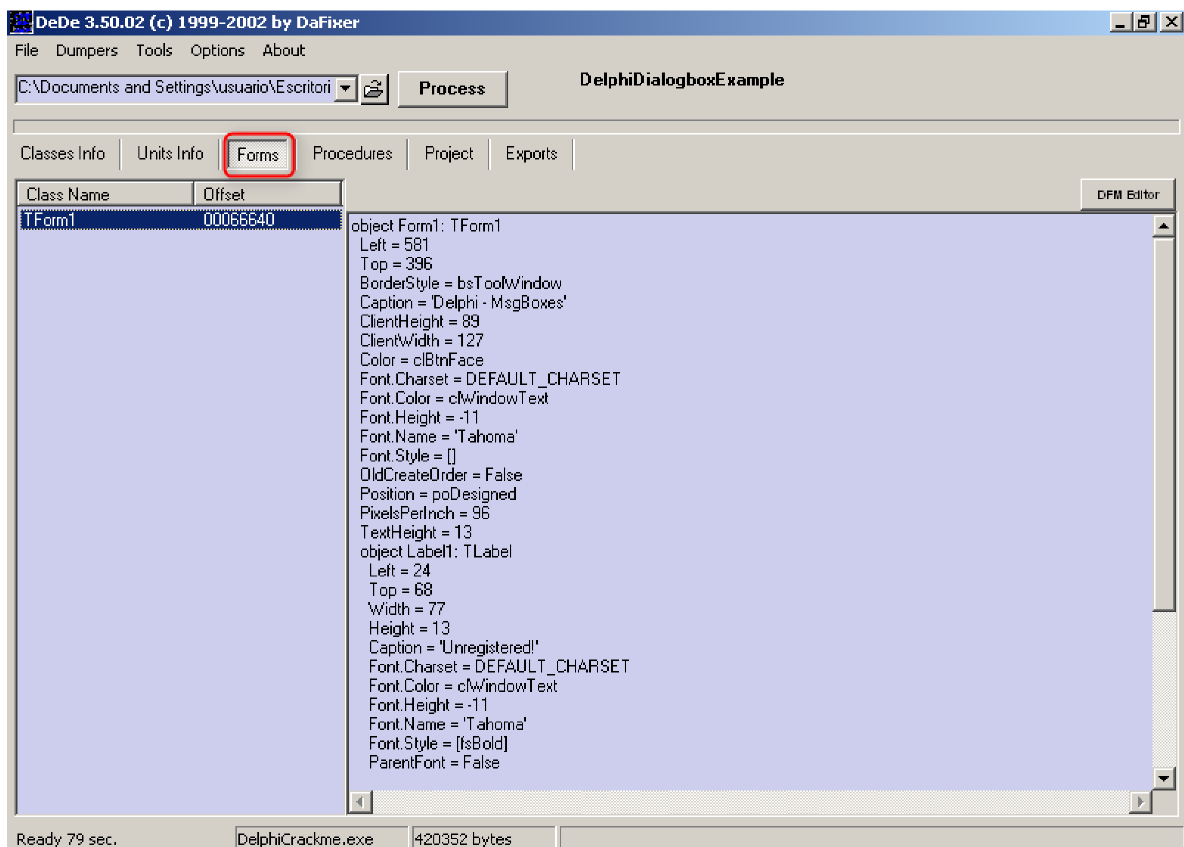
Le decimos que no al analisis extendido ya que haciendo clic en Yes no suele mostrar más información.



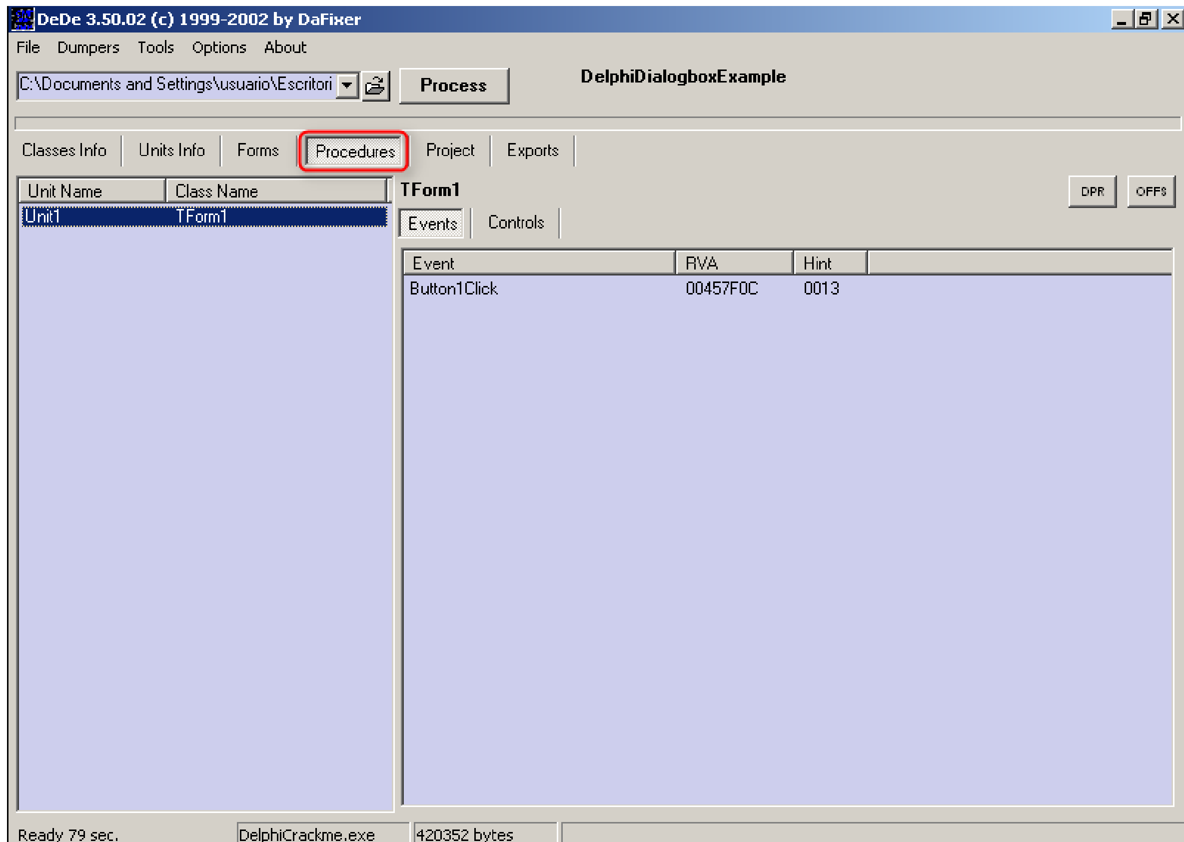
Una vez finalizado el procesamiento de la aplicación nos aparece la ventana principal de DeDe.



Por defecto aparece información sobre la clase de la aplicación, no obstante lo que nos interesa es la pestaña de "Forms".



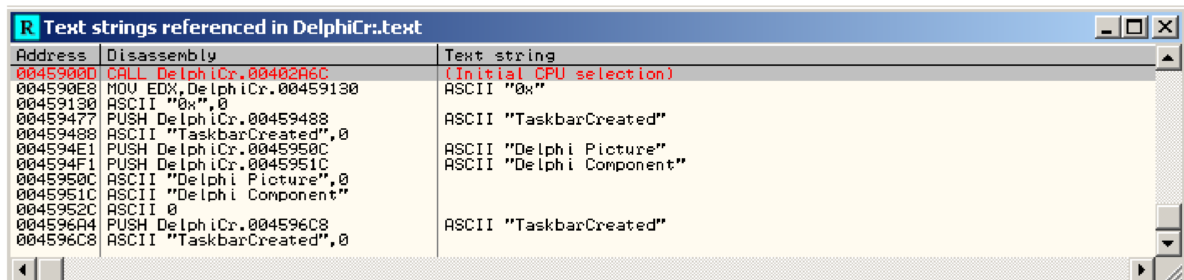
Aquí podemos ver la información que habiamos recopilado en Resource Hacker, es decir los atributos del formulario (en el futuro podemos obviar el analisis con Resource Hacker y centrarnos directamente en el analisis con DeDe). A continuación haremos clic en la pestaña “Procedures”:



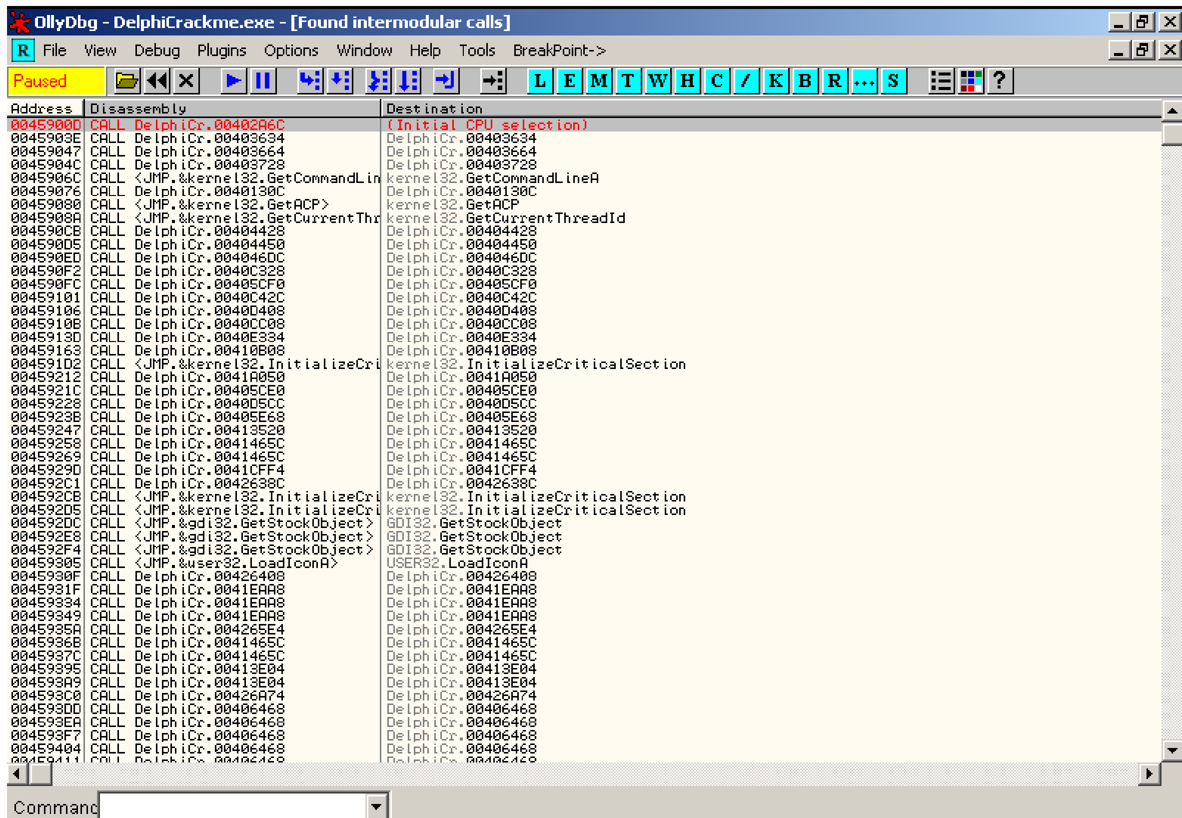
En esta pestaña DeDe nos muestra los nombres y direcciones del método devolución de llamada para el formulario TForm1. Vemos que solo hay un botón y por lo tanto solo una devolución de llamada. Lo importante aquí es que ahora sabemos la dirección de la devolución de llamada: 00457F0C.

Cragamos la aplicación en Olly:

Si buscamos por cadenas de texto vemos que no vamos a tener suerte:

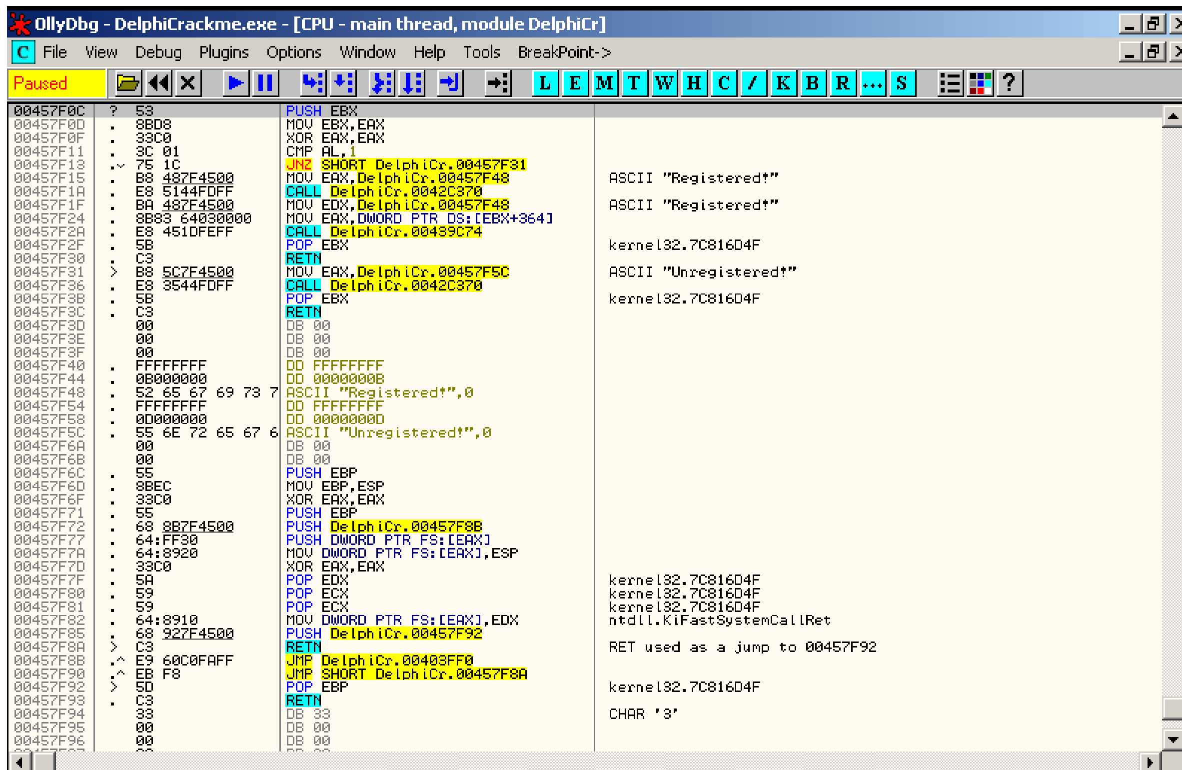


Lo mismo sucede si buscamos por “All intermodular calls”:

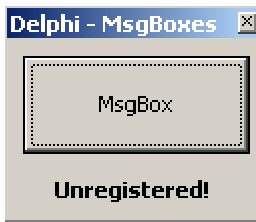


Una de las caracteristicas de los programas de Delphi es que hay incontables CALL’s.

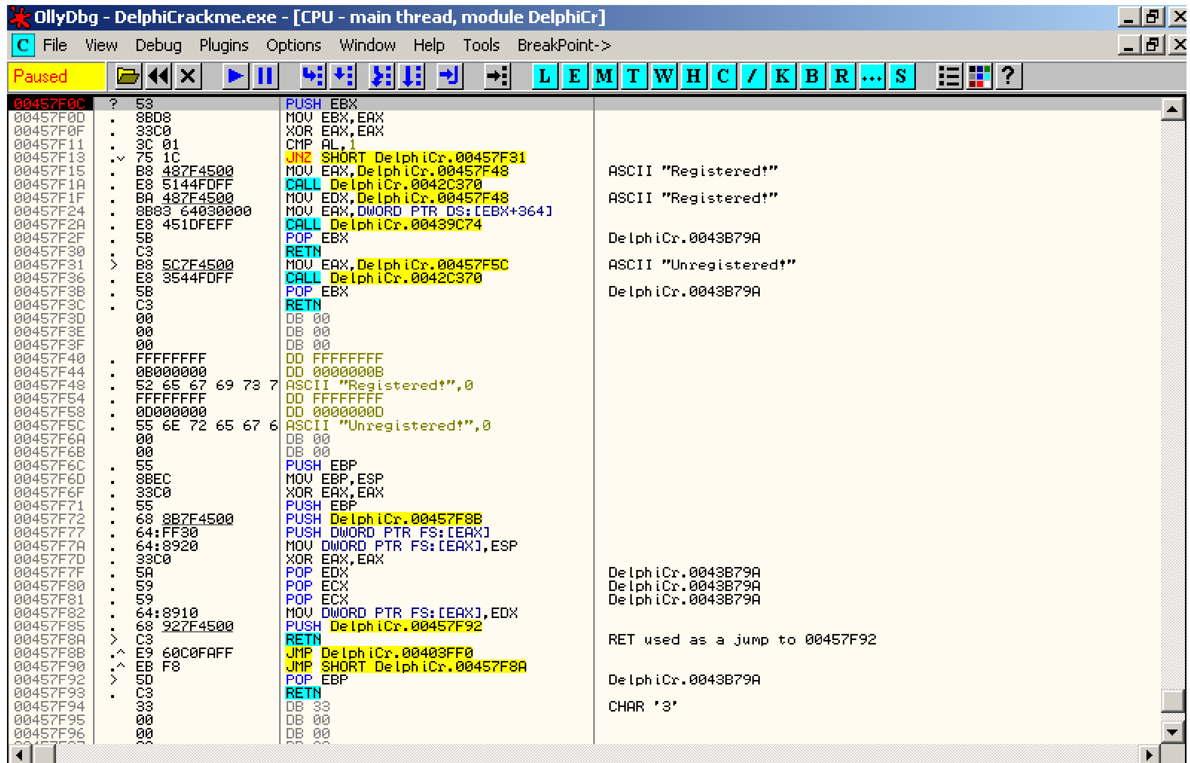
Del analisis con DeDe sabemos la dirección que va a ser llamada cuando hacemos clic sobre un botón. Busquemos pues esa dirección en Olly (457F0C):



Pongamos un Breakpoint en 457F0C y ejecutamos la aplicación:



Vemos que pone “Unregistered”, y el título de la ventana es “Delphi - MsgBoxes”. Y hay un botón. Si pulsamos el botón, Olly se detendrá en nuestro Breakpoint.



Vemos que el salto en 457F13 nos lleva al “bad boy”. Cambiamos el valor de la bandera Z a uno y volvemos a correr la aplicación.

OllyDbg - DelphiCrackme.exe - [CPU - main thread, module DelphiCr]

OllyDbg - DelphiCrackme.exe - [CPU - main thread, module DelphiCr]

Delphicra...

Registered!

OK

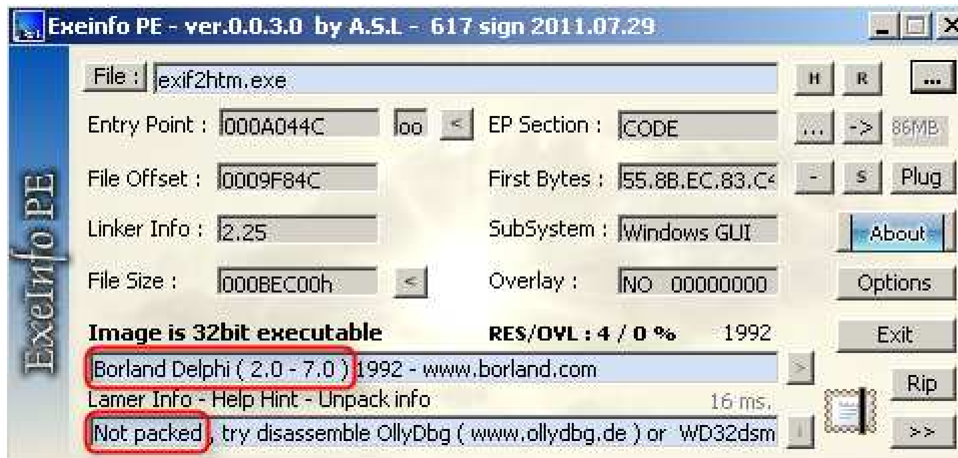
Delphi - MsgBoxes

MsgBox

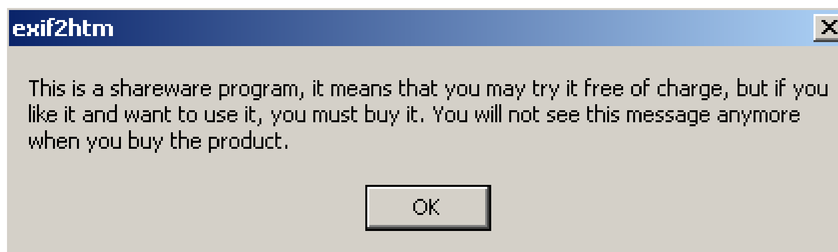
Registered!

7.16.2 exif2htm.exe

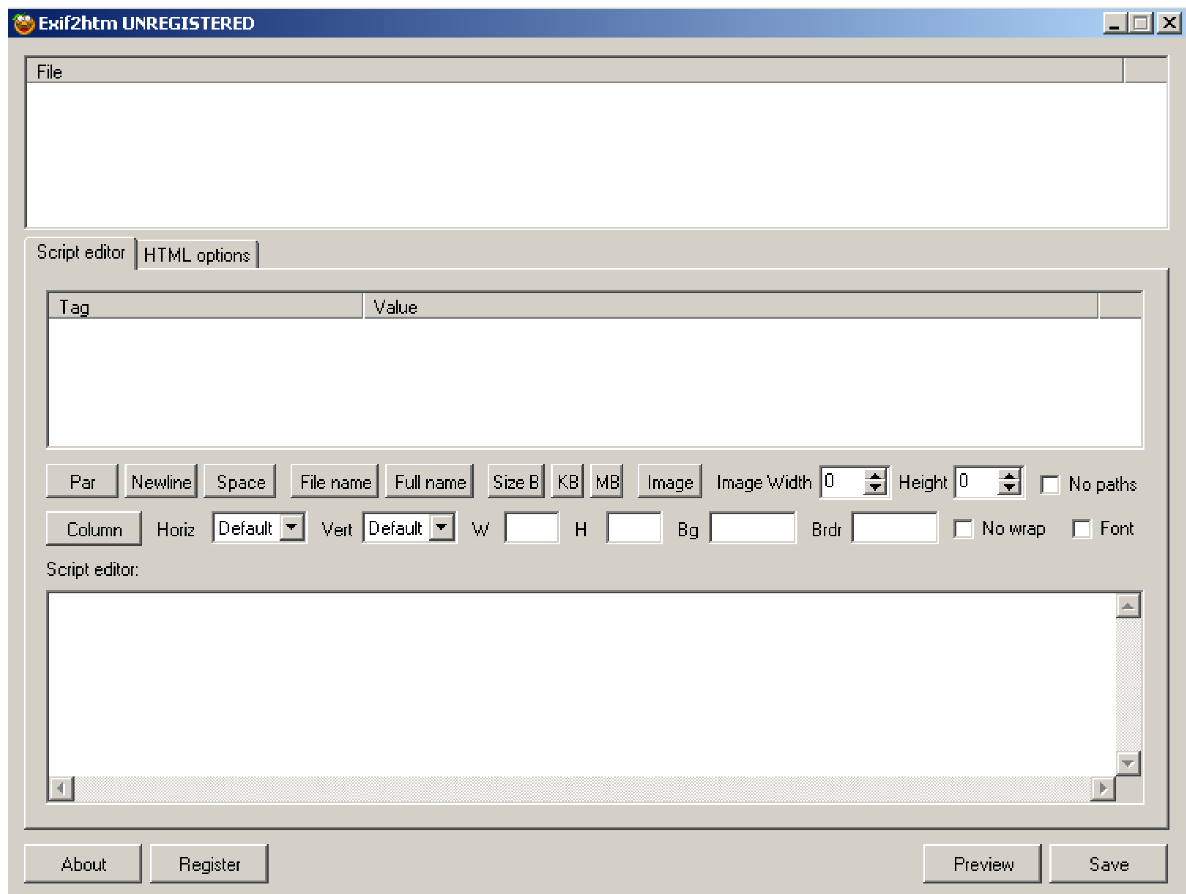
Cargamos la aplicación en ExeInfoPE y vemos que se trata de un programa Delphi sin empaquetar:



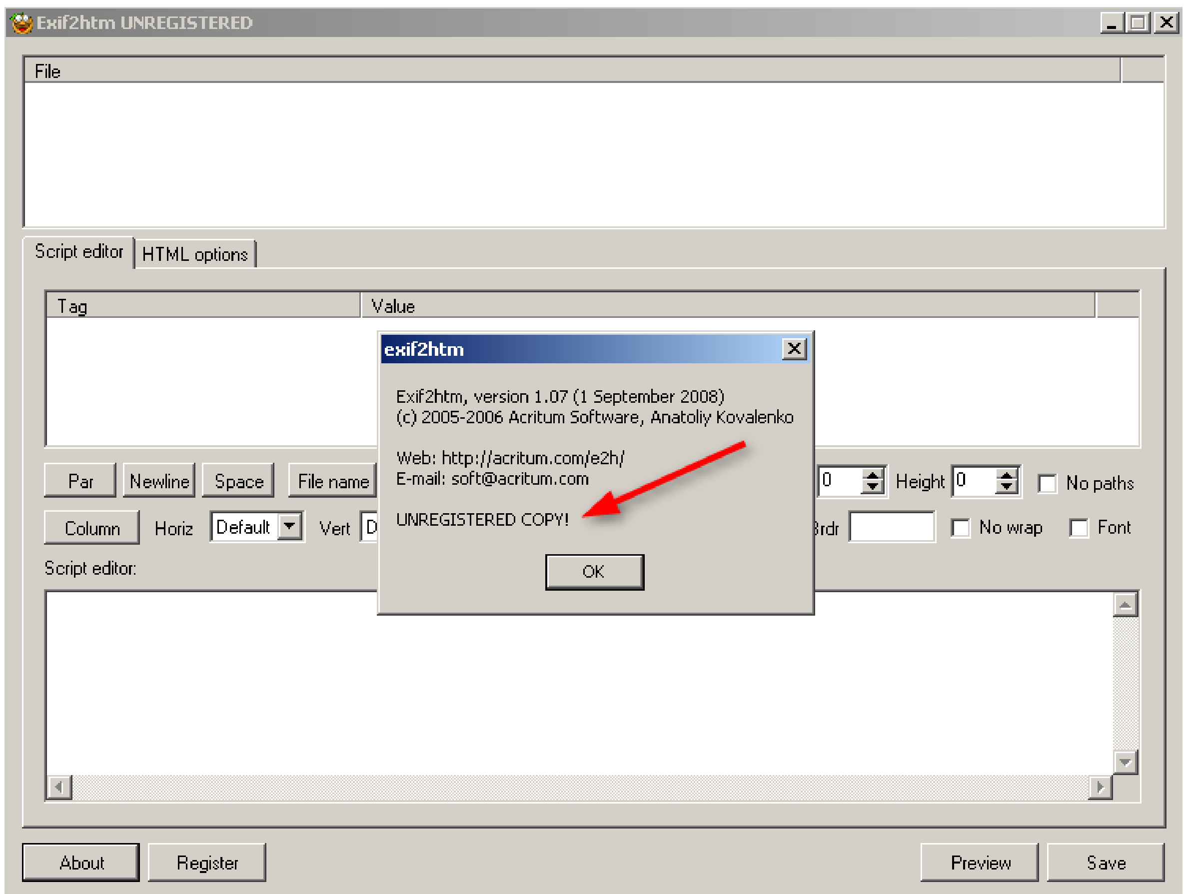
Haciendo doble clic sobre la aplicación podemos apreciar el nag.

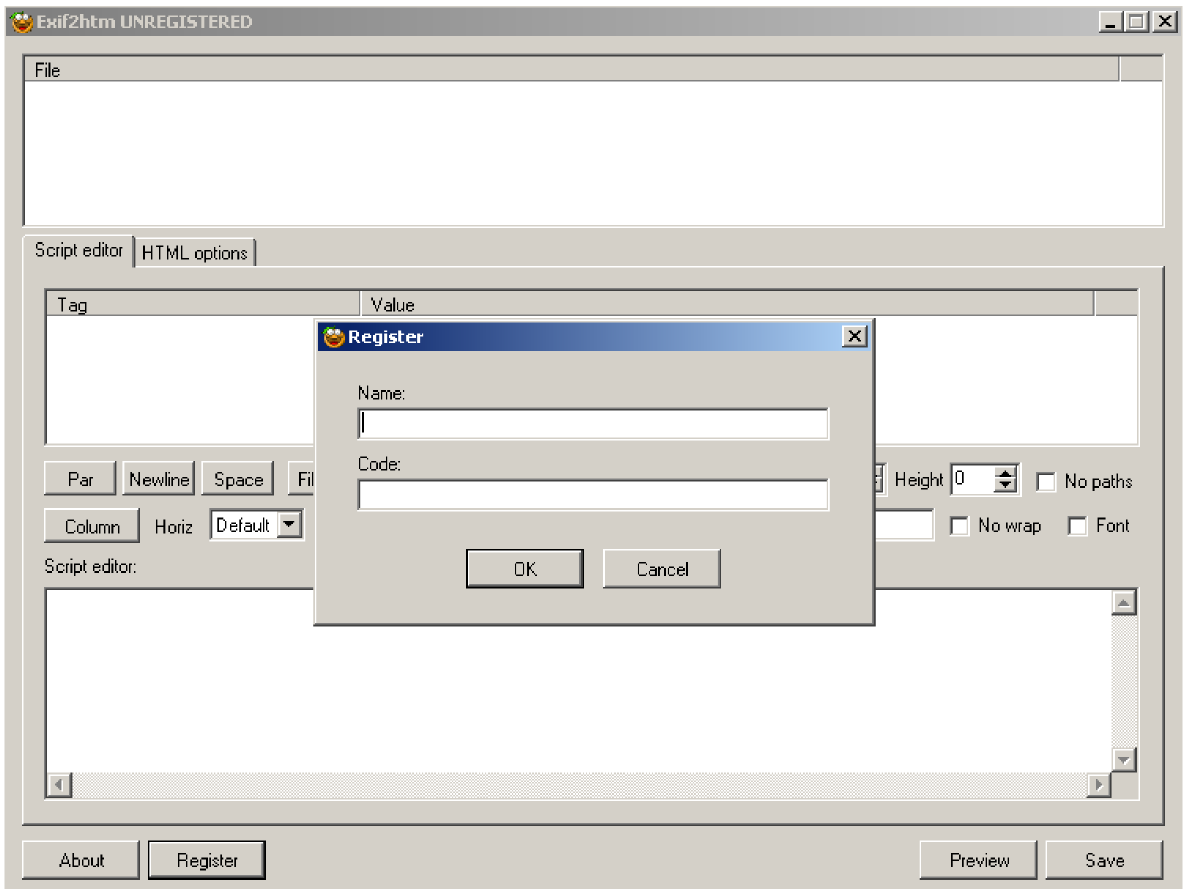


Hacemos clic en OK y se abre la pantalla principal. Vemos arriba en el título de la ventana que estamos sin registrar.

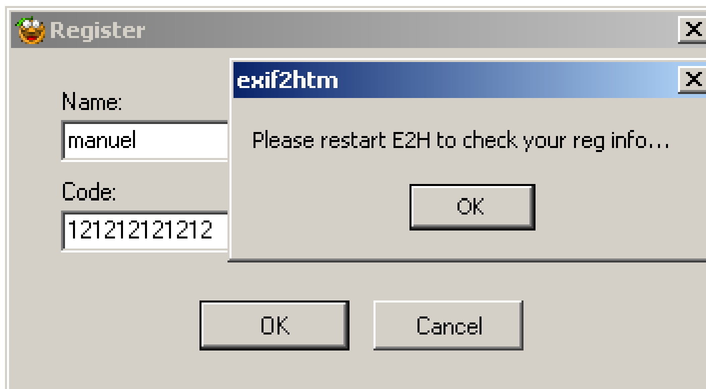


Hacemos clic en el botón “About” y en “Register” para obtener más información:

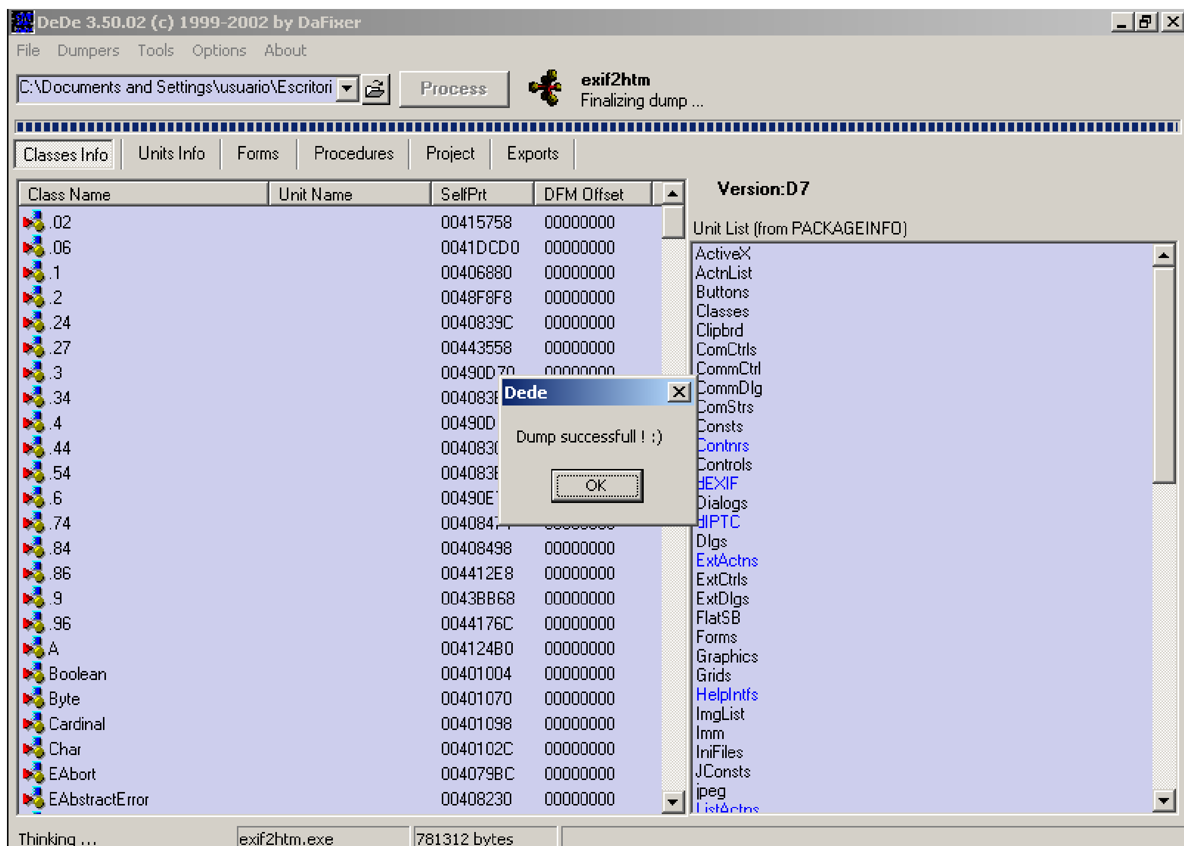

Exif2htm UNREGISTERED
File
Script editor
HTML options
Tag
Value
exif2htm
Exif2htm, version 1.07 (1 September 2008)
(c) 2005-2006 Acritum Software, Anatoliy Kovalenko
Web: http://acritum.com/e2h/
E-mail: soft@acritum.com
UNREGISTERED COPY!
OK
Par
Newline
Space
File name
Height
No paths
Column
Horiz
Default
Vert
No wrap
Font
Script editor:
About
Register
Preview
Save


Exif2htm UNREGISTERED
File
Script editor
HTML options
Tag
Value
Register
Name:
Code:
OK
Cancel
Par
Newline
Space
Height
No paths
Column
Horiz
Default
No wrap
Font
Script editor:
About
Register
Preview
Save

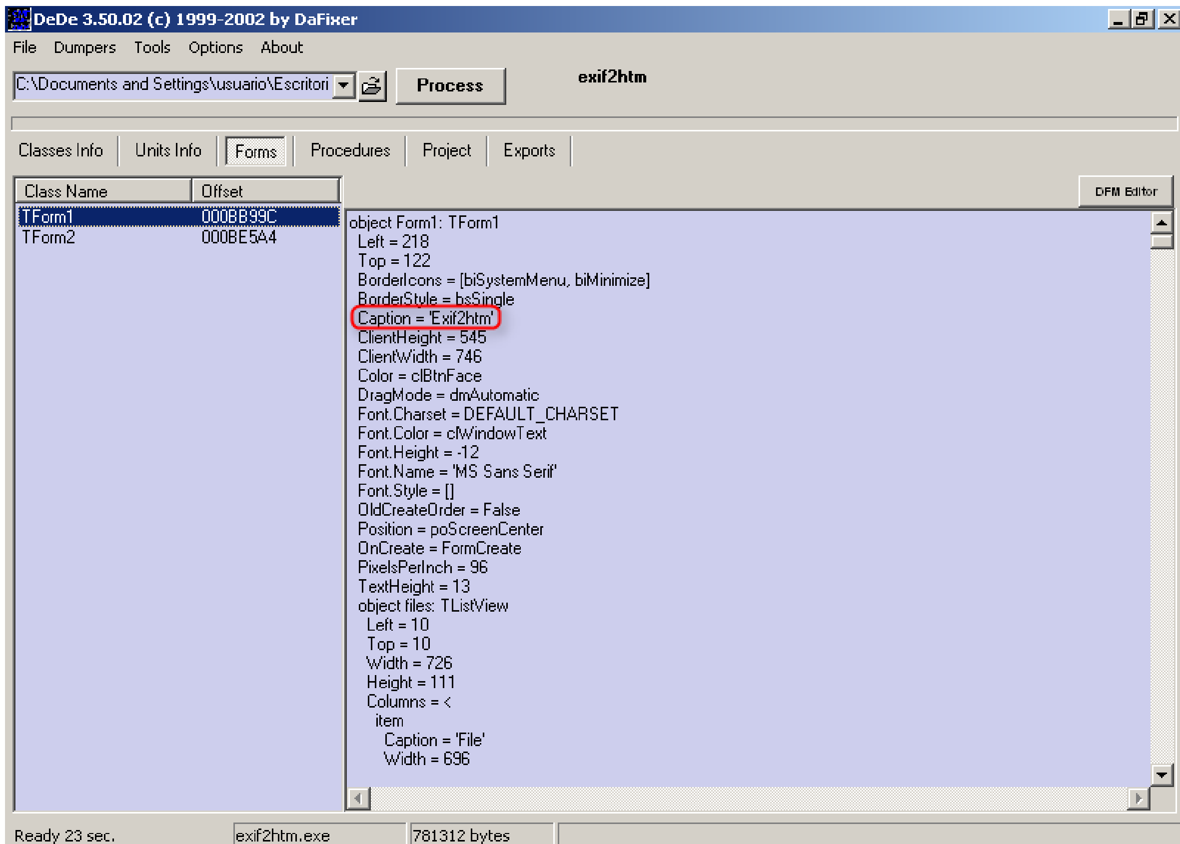
Introducimos un nombre y un código cualquiera y nos aparece el siguiente mensaje:



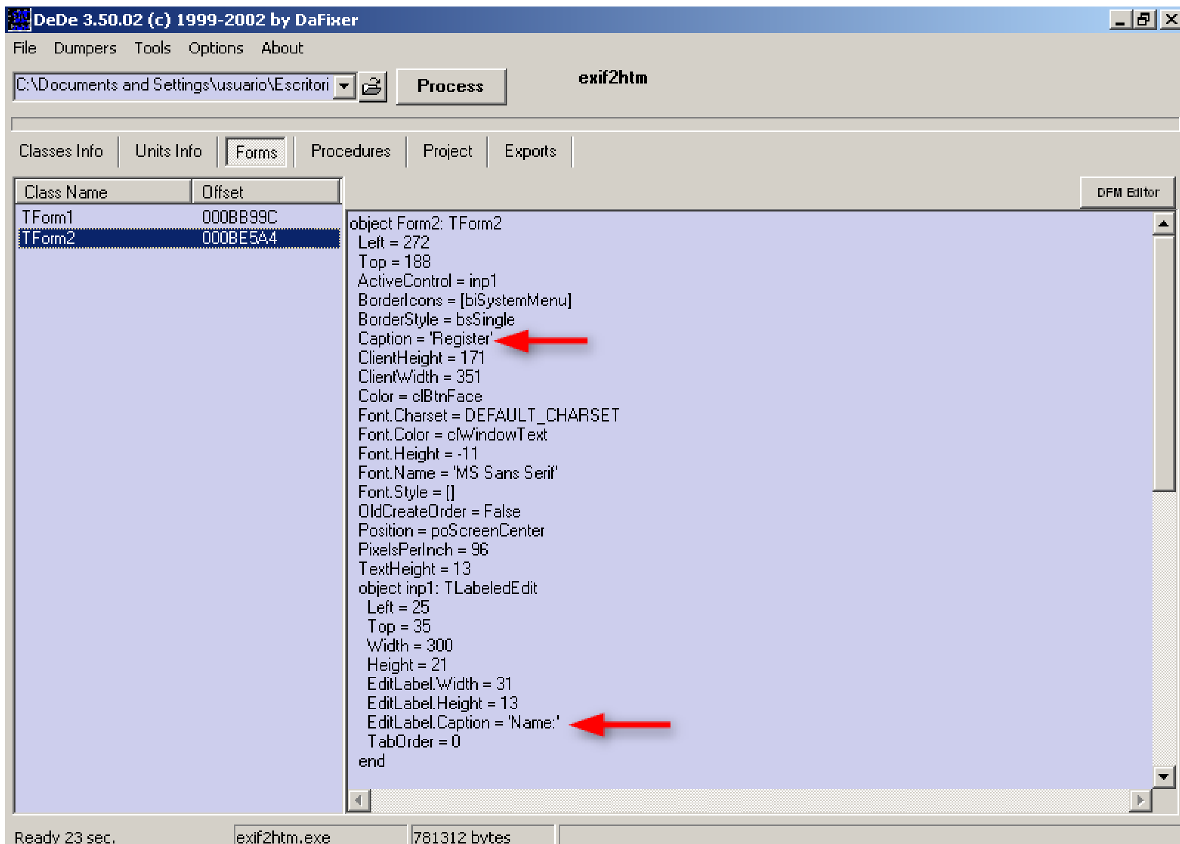
Vamos a cargar la aplicación en DeDe con el fin de sacar algo más de información:

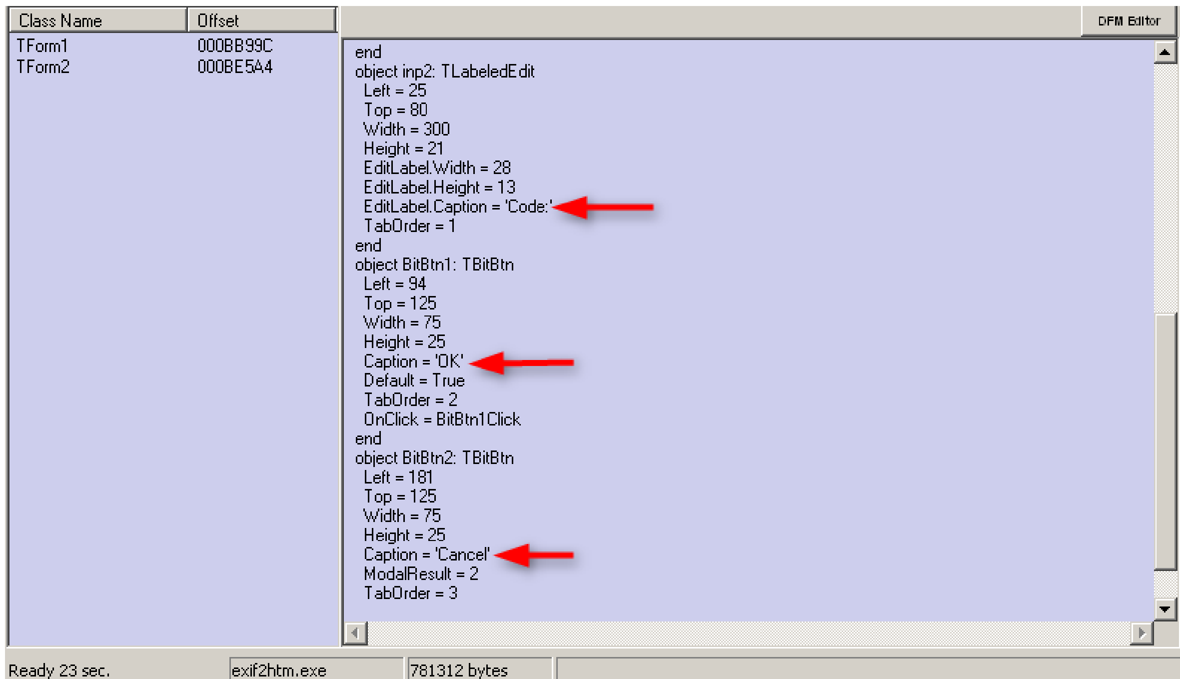


Hacemos clic en la pestaña “Forms”. Seleccionamos TForm1 y vemos que probablemente se trata de la ventana principal.

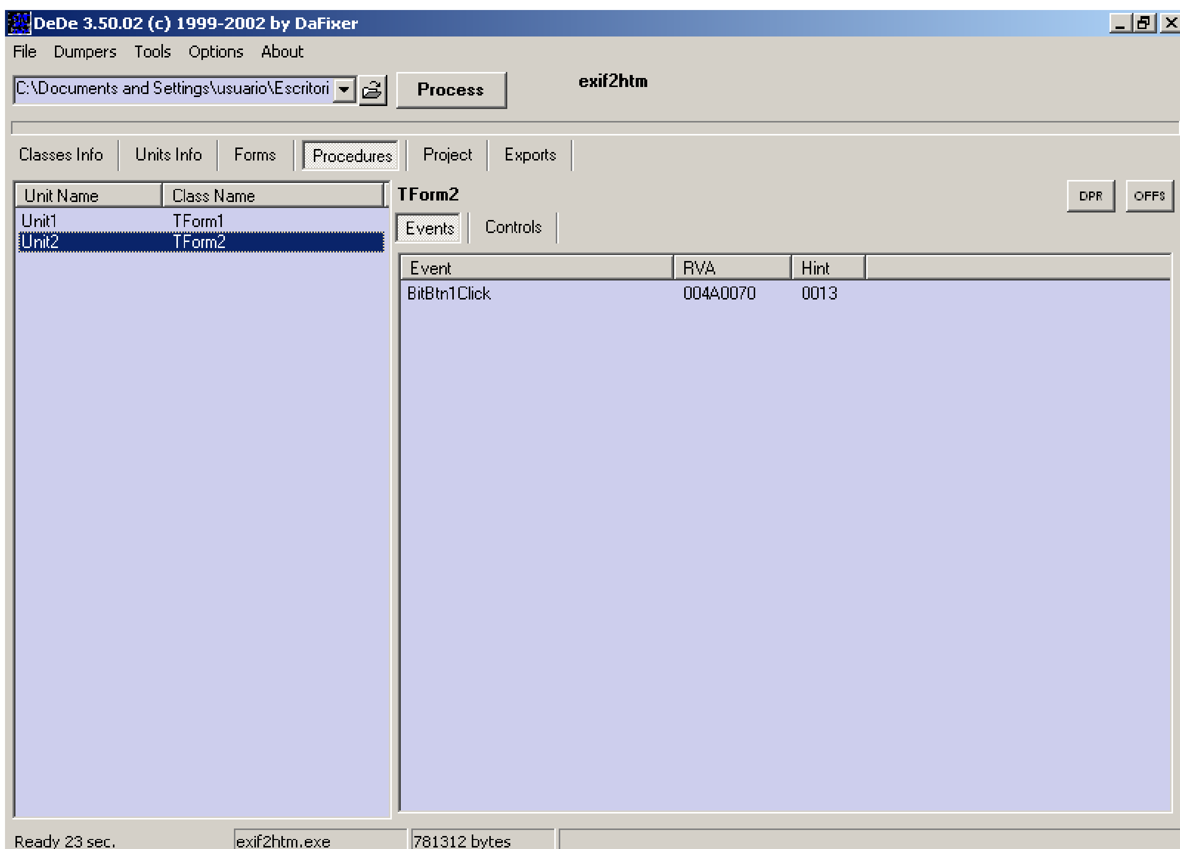


Seleccionando TForm2, y podemos ver que el caption es ‘Register’, vemos que están las etiquetas ‘Name’ y ‘Code’, además existen dos botones al final; ‘OK’ y ‘Cancel’.





A continuación seleccionamos la pestaña "Procedures":



Haciendo clic en Unit2 – Tform2, podemos apreciar que hay un método, "BitBtn1Click", que corresponde al botón "OK" de la ventana de registro, según hemos podido ver en la pestaña de Forms. Además vemos que DeDe nos provee con la dirección RVA para ese método. Carguemos pues la aplicación en Olly:

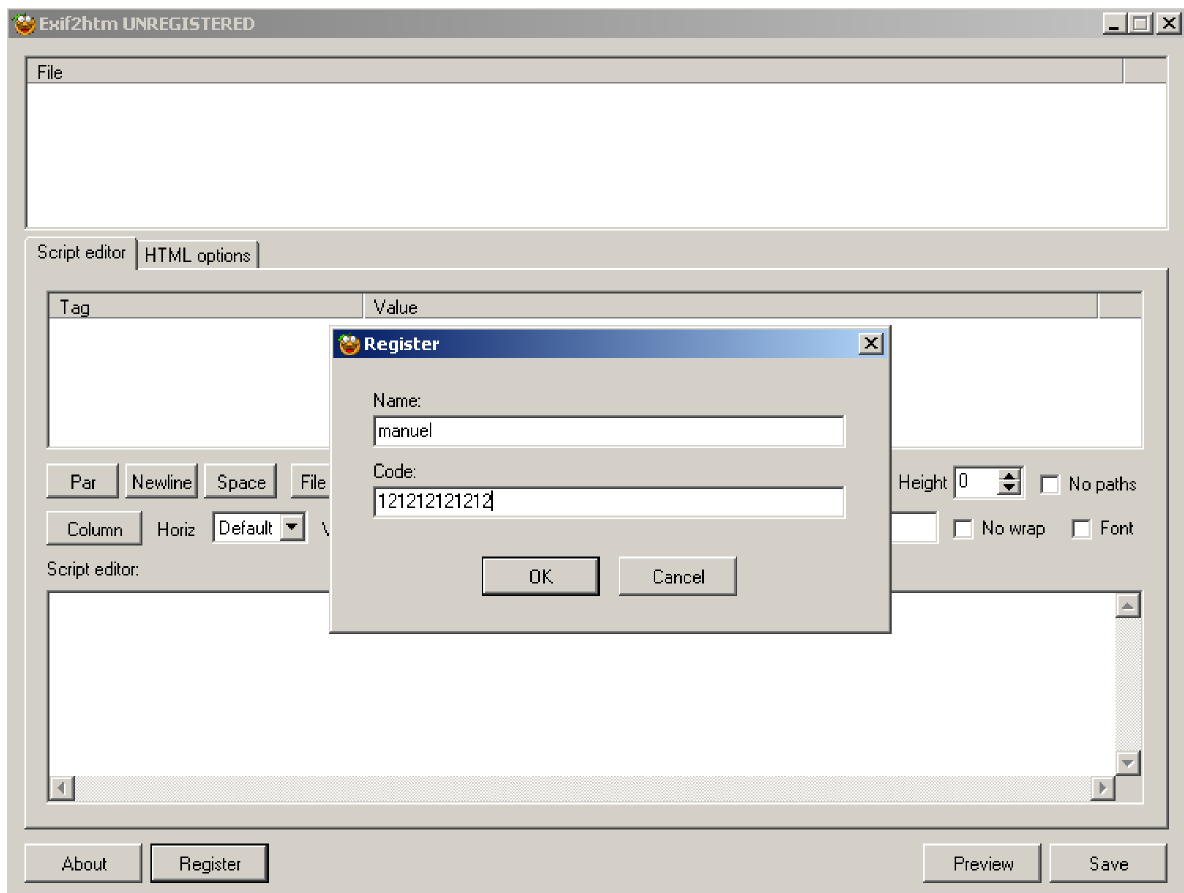
A primera vista no parece ser un binario muy amigable. Busquemos la dirección 4A0070 y pongamos un Breakpoint.

Un poco más abajo y dentro de la misma rutina podemos apreciar la cadena de texto:

```
004A0110  .  8B08           MOV ECX,DWORD PTR DS:[EAX]
004A0112  .  FF51 74        CALL DWORD PTR DS:[ECX+74]
004A0115  .  8BC6           MOV EAX,ESI
004A0117  .  E8 F836F6FF    CALL exif2htm.00403814
004A011C  .  B8 7C014A00    MOV EAX,exif2htm.004A017C        ASCII "Please restart E2H to check your reg info..."
004A0121  .  E8 BA1FF9FF    CALL exif2htm.004320E0
004A0126  .  A1 30AD4A00    MOV EAX,DWORD PTR DS:[4AAD30]
004A012B  .  E8 4C3FFDFF    CALL exif2htm.0047407C
```

Esta técniqua se utiliza bastante y la solución consiste en averiguar donde almacenará la aplicación el nombre y el código que se introdujo nada más ejecutar la aplicación. Solo hay un par de lugares donde una aplicación pueda almacenar datos entre ejecución y ejecución. Esos lugares son o bien los registros o un archivo ini. El siguiente paso pues es averiguar donde podría almacenar la aplicación esos datos de forma que cuando volvamos a ejecutar la aplicación sepamos en que lugar se está comprobando si estamos registrados o no.

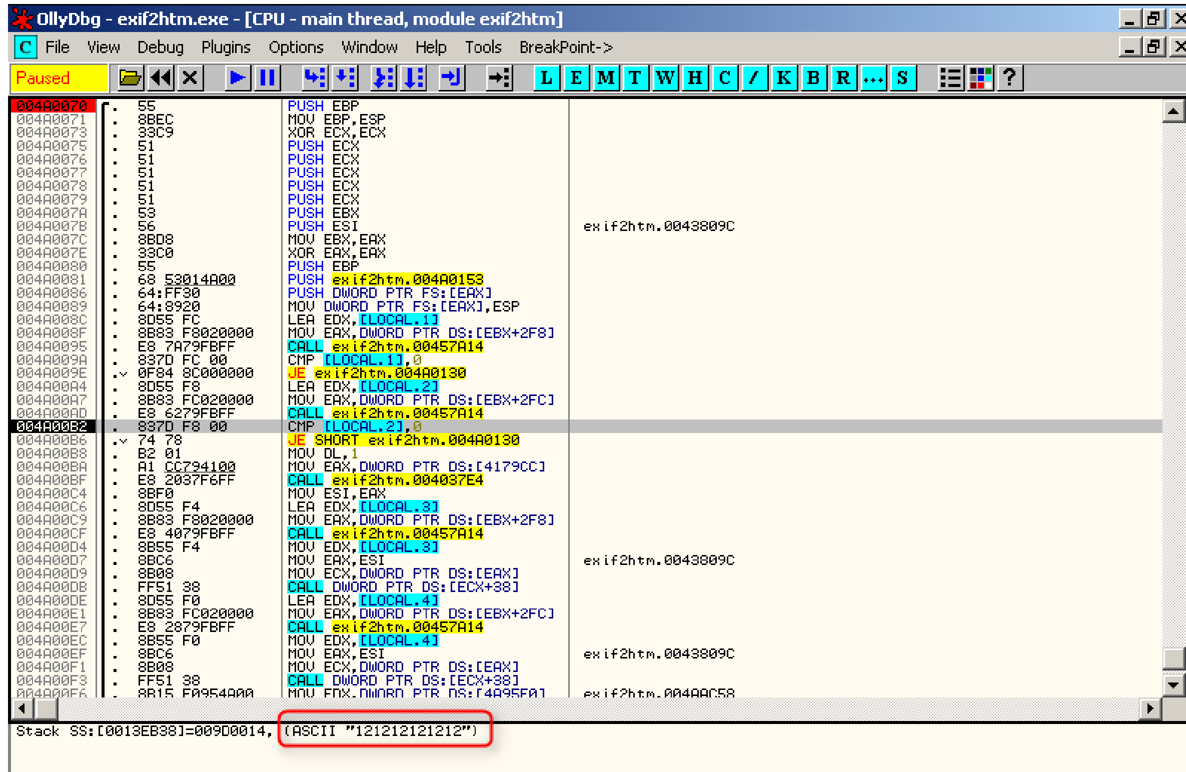
Una vez puesto nuestro Breakpoint pulsamos F9 para ejecutar la aplicación. Pulsamos el botón “Register”, introducimos un nombre y código cualquiera y hacemos clic en “OK”.



Olly se detiene en nuestro breakpoint:

Vemos que hay un conjunto de variables que son empujados en la pila. Después aparece un CALL en 4A0095. Si seguimos pasando por el código y nos detenemos en la dirección 4A009A, podemos ver algo interesante en la ventana de información:

En el 99.99% de los casos estamos ante una comprobación por parte de la aplicación para verificar si hemos introducido ‘algo’ en el campo texto. El hecho de que EAX sea igual a 6 podría indicar que estamos comprobando la longitud de la cadena de texto. Vemos que justo a continuación se comprueba si EAX es igual a cero y después viene un salto. Si continuamos pulsando F8 vemos que aparece también el código que hemos introducido:

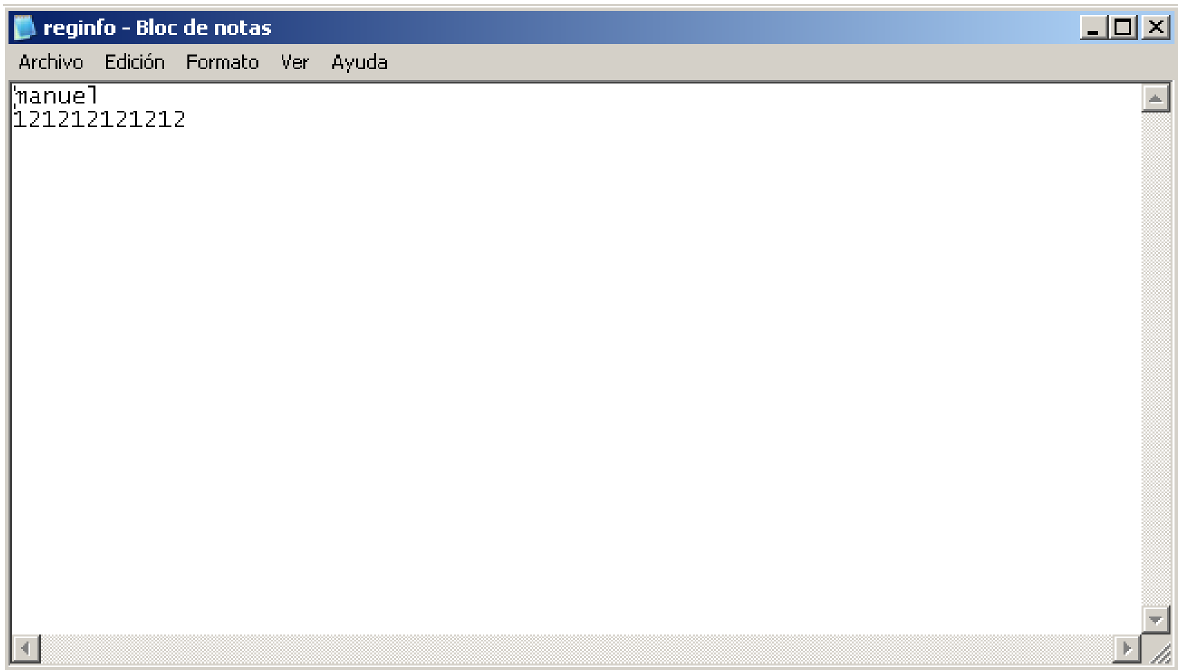


EAX vuelve a ser comparado a cero (ahora es ocho), y salta si es cero. Después pasamos un conjunto de CALL’s. Cada uno de esos CALL’s cargará otra vez nuestro nombre y el código como argumentos. Pulsamos F8 hasta llegar a 4A0101:

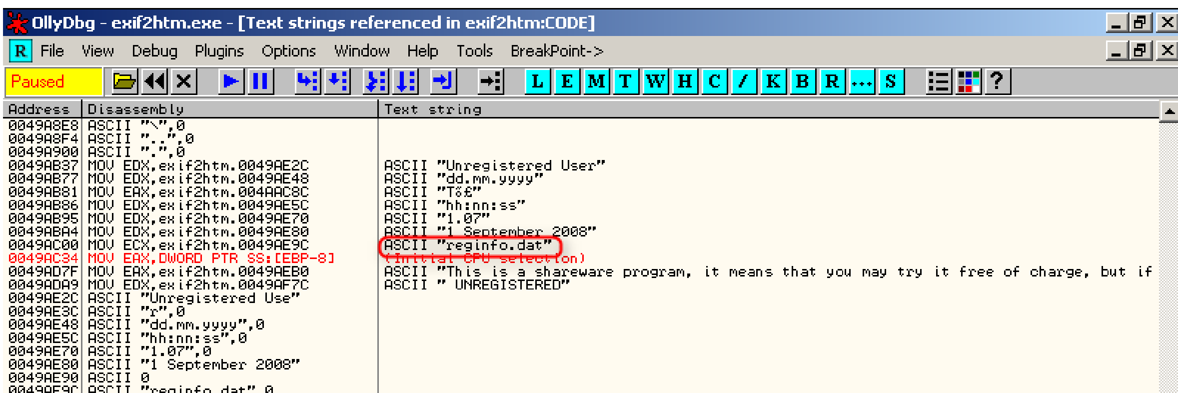
```
OllyDbg - exif2htm.exe - [CPU - main thread, module exif2htm]
C File View Debug Plugins Options Window Help Tools BreakPoint->
Paused

004A00B2  .   837D F8 00      CMP [LOCAL.2],0
004A00B6  .v  74 78           JE SHORT exif2htm.004A0130
004A00B8  .   B2 01           MOV DL,1
004A00BA  .   A1 CC794100     MOV EAX,DWORD PTR DS:[4179CC]
004A00BF  .   E8 2037F6FF     CALL exif2htm.004037E4
004A00C4  .   8BF0            MOV ESI,EAX
004A00C6  .   8D55 F4         LEA EDX,[LOCAL.3]
004A00C9  .   8B83 F8020000   MOV EAX,DWORD PTR DS:[EBX+2F8]
004A00CF  .   E8 4079FBFF     CALL exif2htm.00457A14
004A00D4  .   8B55 F4         MOV EDX,[LOCAL.3]
004A00D7  .   8BC6            MOV EAX,ESI
004A00D9  .   8B08            MOV ECX,DWORD PTR DS:[EAX]
004A00DB  .   FF51 38         CALL DWORD PTR DS:[ECX+38]
004A00DE  .   8D55 F0         LEA EDX,[LOCAL.4]
004A00E1  .   8B83 FC020000   MOV EAX,DWORD PTR DS:[EBX+2FC]
004A00E7  .   E8 2879FBFF     CALL exif2htm.00457A14
004A00EC  .   8B55 F0         MOV EDX,[LOCAL.4]
004A00EF  .   8BC6            MOV EAX,ESI
004A00F1  .   8B08            MOV ECX,DWORD PTR DS:[EAX]
004A00F3  .   FF51 38         CALL DWORD PTR DS:[ECX+38]
004A00F6  .   8B15 F0954A00   MOV EDX,DWORD PTR DS:[4A95F0]     exif2htm.004AAC58
004A00FC  .   8B12            MOV EDX,DWORD PTR DS:[EDX]
004A00FE  .   8D45 EC         LEA EAX,[LOCAL.5]
004A0101  .   B9 68014A00     MOV ECX,exif2htm.004A0168         ASCII "reginfo.dat"
004A0106  .   E8 A547F6FF     CALL exif2htm.004048B0
004A010B  .   8B55 EC         MOV EDX,[LOCAL.5]
004A010E  .   8BC6            MOV EAX,ESI
004A0110  .   8B08            MOV ECX,DWORD PTR DS:[EAX]
004A0112  .   FF51 74         CALL DWORD PTR DS:[ECX+74]
004A0115  .   8BC6            MOV EAX,ESI
004A0117  .   E8 F836F6FF     CALL exif2htm.00403814
004A011C  .   B8 7C014A00     MOV EAX,exif2htm.004A017C         ASCII "Please restart E2H to check your reg info..."
004A0121  .   E8 BA1FF9FF     CALL exif2htm.004320E0
004A0126  .   A1 30AD4A00     MOV EAX,DWORD PTR DS:[4AAD30]
004A012B  .   E8 4C3FFDFF     CALL exif2htm.0047407C
004A0130  >   33C0            XOR EAX,EAX
004A0132  .   5A              POP EDX                           0013EE78
004A0133  .   59              POP ECX                           0013EE78
004A0134  .   59              POP ECX                           0013EE78
004A0135  .   64:8910         MOV DWORD PTR FS:[EAX],EDX
004A0138  .   68 5A014A00     PUSH exif2htm.004A015A
004A013D  >   8D45 EC         LEA EAX,[LOCAL.5]
004A0140  .   E8 5F44F6FF     CALL exif2htm.004045A4
004A0145  .   8D45 F0         LEA EAX,[LOCAL.4]
004A0148  .   BA 04000000     MOV EDX,4
```

Llegados a este punto parece que la aplicación va a crear un archivo de datos. Seguimos pulsando F8 hasta llegar al call en 4A0112. Una vez pasado, vamos a la carpeta donde se encuentra nuestro ejecutable y vemos que efectivamente se ha creado un nuevo archivo:

Abrimos el archivo para ver lo que contiene:



Ahora ya sabemos donde se almacenan nuestros datos. Vamos a Olly y busquemos la cadena de texto “reginfo.dat”:



Vamos al código y ponemos un Breakpoint:

Cerrarmos la aplicación (cerrando la aplicación y no cerrando Olly), reiniciamos la aplicación a través de Olly y nos detenemos en nuestro Breakpoint.

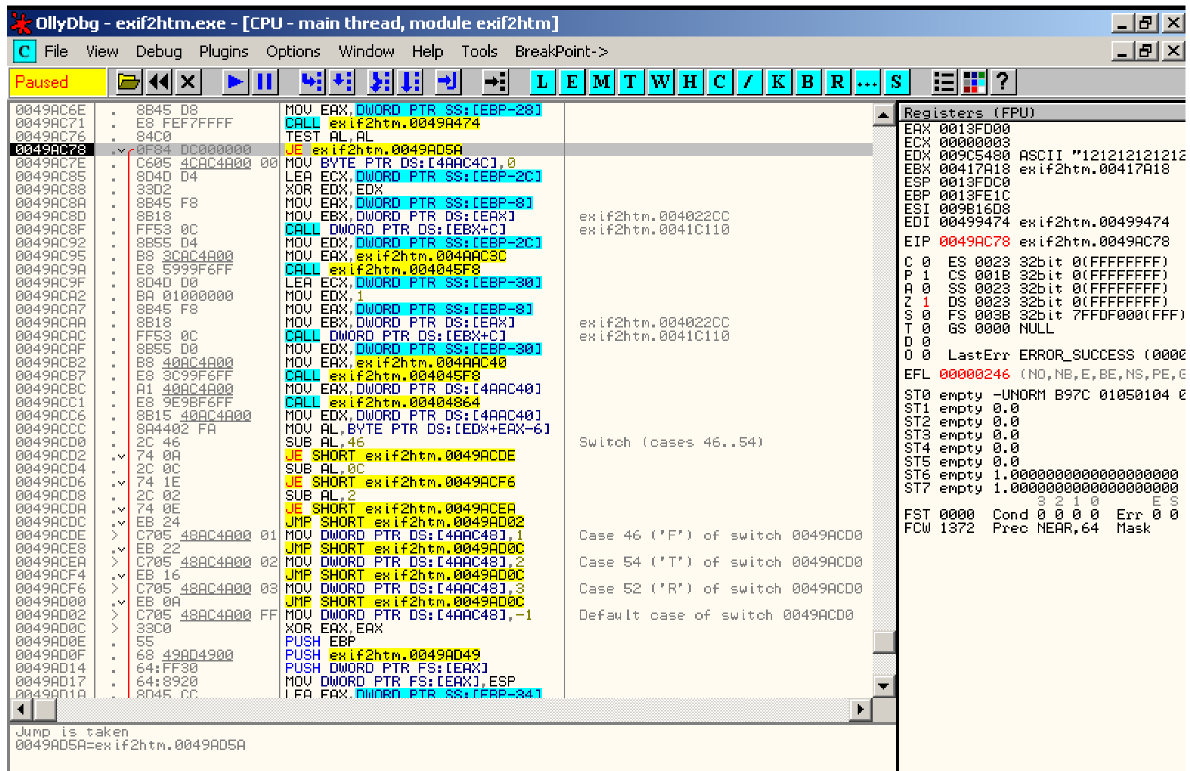
Pulsamso F8 para ejecutar el código paso a paso. En el primer salto condicional, (49AC26) no saltamos.

En el siguiente conjunto de instrucciones nuestro nombre y el código son cargados para realizar una serie de CALL's con ellos.

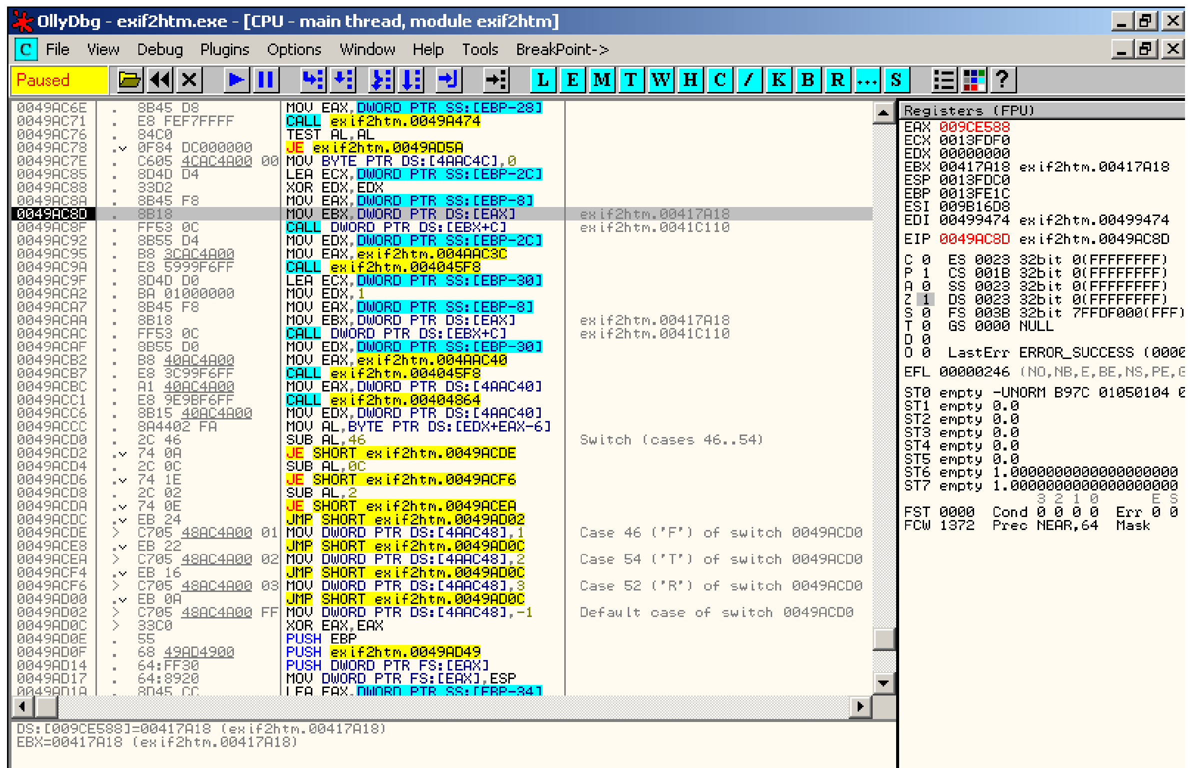
El siguinete salto condicional en la dirección 49AC58 si lo vamos a tomar. Pulsamos F8 y vemos hacia donde saltamos:

Vemos que nos estamos acercando al “bad boy”. Reiniciamos la aplicación y esta vez no tomamos el salto en la dirección 49AC58.

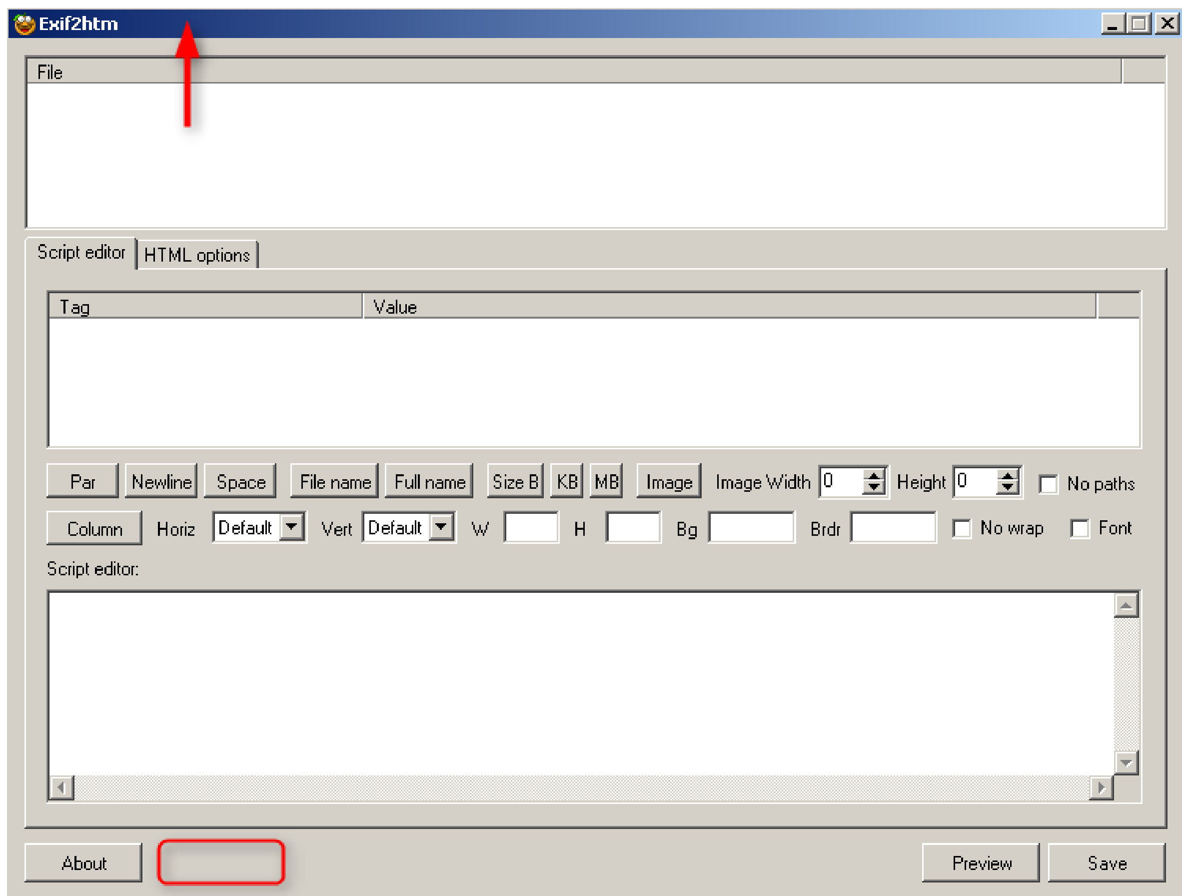
Cambiamos el valor de la bandera Z a cero, y miramos lo que pasa en el siguiente salto condicional, en la dirección 49AC78:



Siguiendo la flecha roja vemos que el salto nos lleva al mismo lugar que el salto anterior (al mensaje del “bad boy”). Lo que nos está diciendo esto es que se trata de una segunda comprobación de nuestro nombre / código. Volvemos a cambiar el valor de la bandera Z para no coger el salto y seguimos pulsando F8:



Seguimos pasando línea por línea y vemos que no sucede nada en especial. Pulsamos entonces F9 para ejecutar la aplicación. Notaremos que nuestro nag ha desaparecido. También vemos que ha desaparecido el texto "UNREGISTERED" en la parte superior de la ventana y el botón de "Register" en la parte inferior.



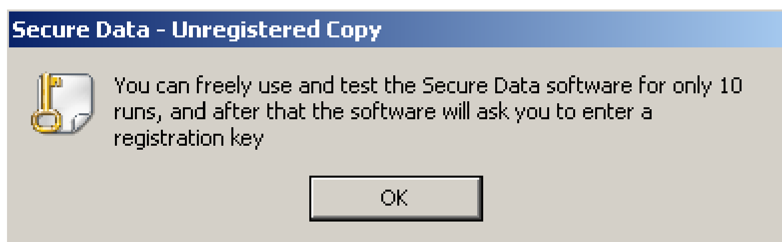
Conclusión: Hemos forzado a la aplicación para que utilice cualquier nombre y código.

.

7.17 Caso práctico 17: Periodos de prueba y Hardware Breakpoints

Muchos programas ofrecen períodos de prueba que permiten utilizar el software sin costo durante un período de tiempo predeterminado. Una vez finalizado, ya no es posible utilizar el software de forma gratuita. Para continuar utilizando el programa, debemos pagar por una suscripción o utilizar uno de los métodos para crackear el software de prueba.

Ejecutamos el programa SecureData.exe e inmediatamente aparece una ventana con el periodo de prueba.

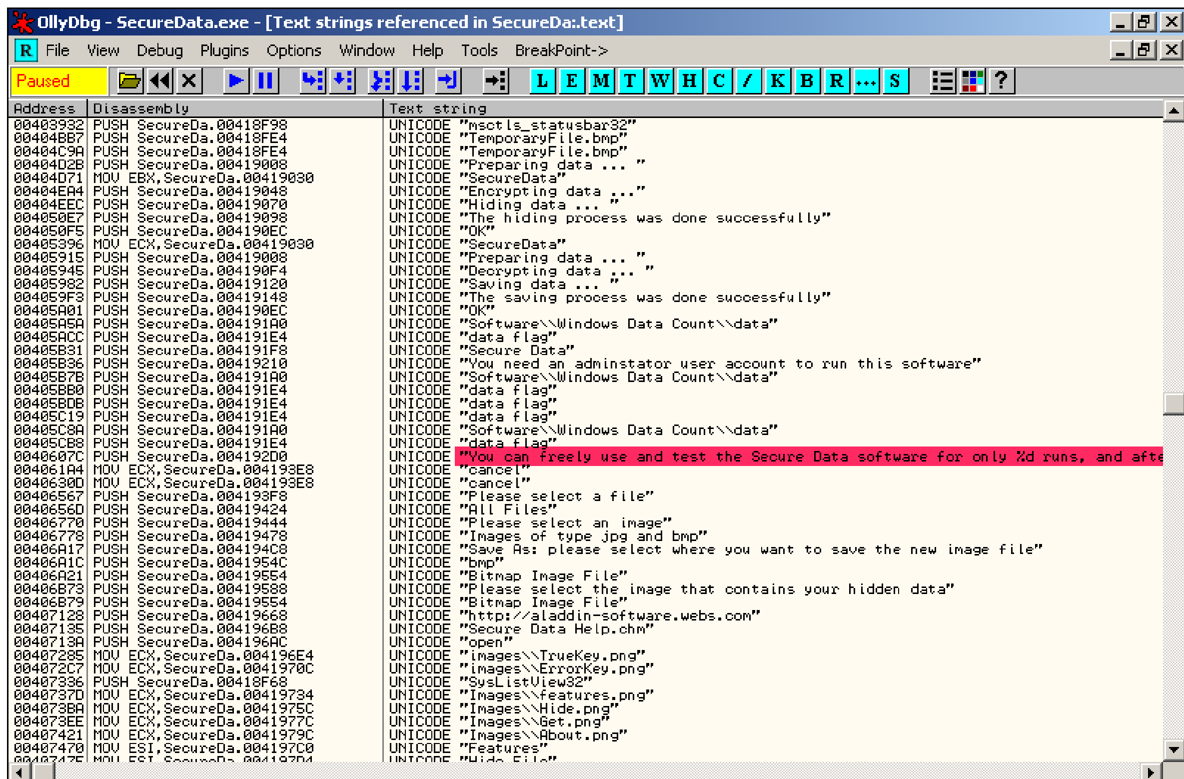


Hacemos clic en "OK" y se abre la ventana principal.

Hacemos clic en “About”:

Introducimos un código y nos sale el siguiente mensaje:

Hemos visto lo suficiente. Cargemos la aplicación en Olly y buscamos las cadenas de texto:



Vemos el texto del periodo de prueba que aparece en el nag. Sin embargo no vemos ninguna cadena de texto que haga referencia a la pantalla de registrar.

Nota: *No todas las cadenas de texto en una aplicación van a estar siempre en la memoria todo el tiempo. En algunas aplicaciones las cadenas más importantes no van a ser descifradas hasta que sean usadas por el programa. Este ejecutable puede ser uno de estos casos; cuando la panatalla de registrar necesita las cadenas para comunicarle al usuario que el código introducido es incorrecto, esas cadenas van a ser descifradas en ese preciso momento.*

Lo más importante a tener en cuenta cuando se trabaja con periodos de prueba es que la aplicación tiene que recordar la cantidad de intentos/días que restan despues de cerrar / reiniciar el programa. Esto significa que hay ciertos datos que necesitan ser almacenados en algún sitio de forma persistente. Los lugares más comunes para almacenar dichos datos son los registros o un archivo en el disco duro.

En la mayoría de los casos estos datos no son alterados por lo que suele ser bastante facil localizarlos. Lo más seguro es buscar una ruta hacia un archivo o un registro que podría ser como el siguiente:

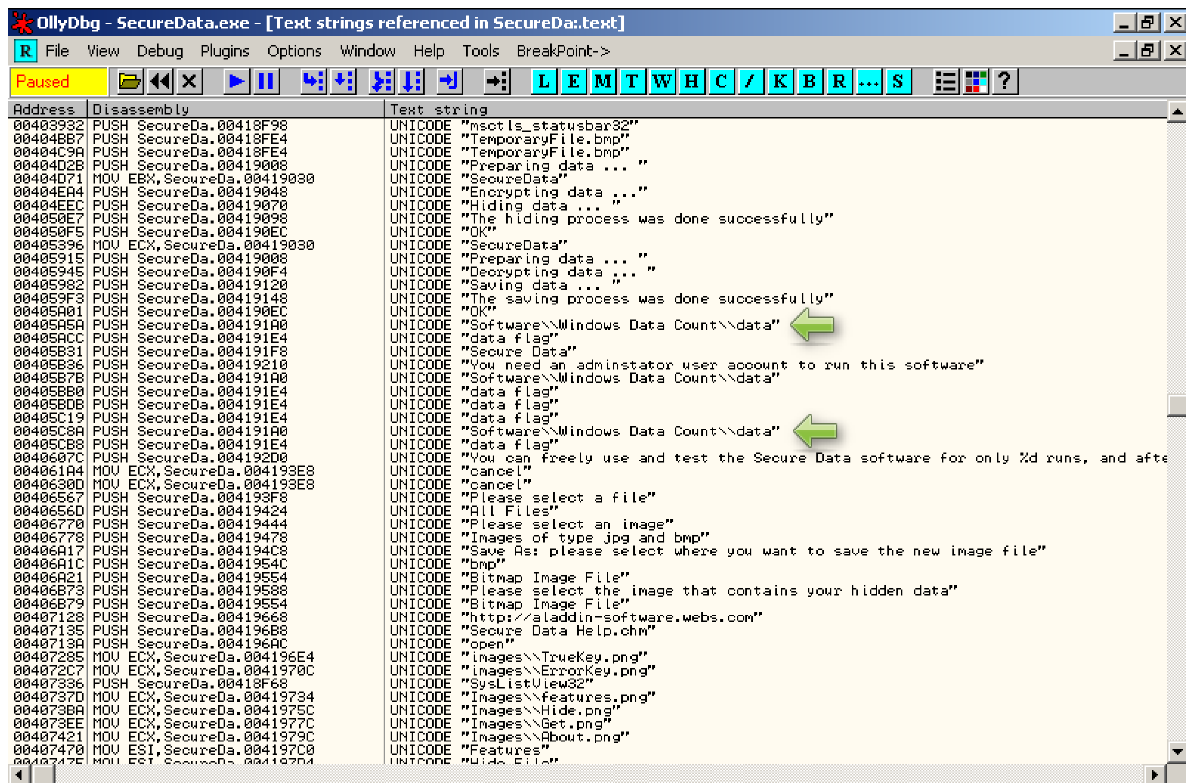
Software\\AppName\\Key // (para la ruta de un registro)

AppName\DataFileName.ini or AppName\DataFileName.dat //(para la ruta de un archivo)

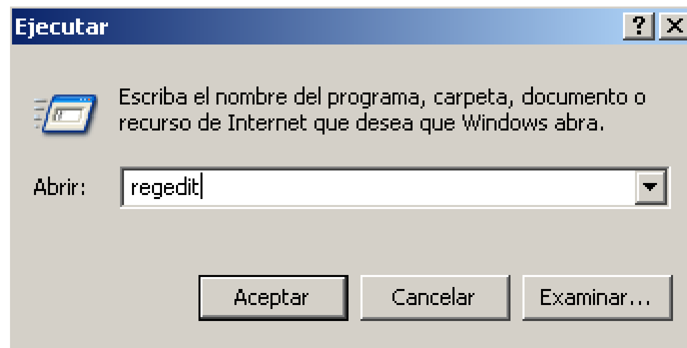
Muchas veces aparecerá algo asi como %WINDOWS% alrededor del nombre, lo que apunta al directorio de instalación de Windows.

También podemos hacer una busqueda por “intermodular calls”. En este caso nos fijaríamos en los CALL’s CreateFileExA (para archivos) y RegSetValueExA (para registros).

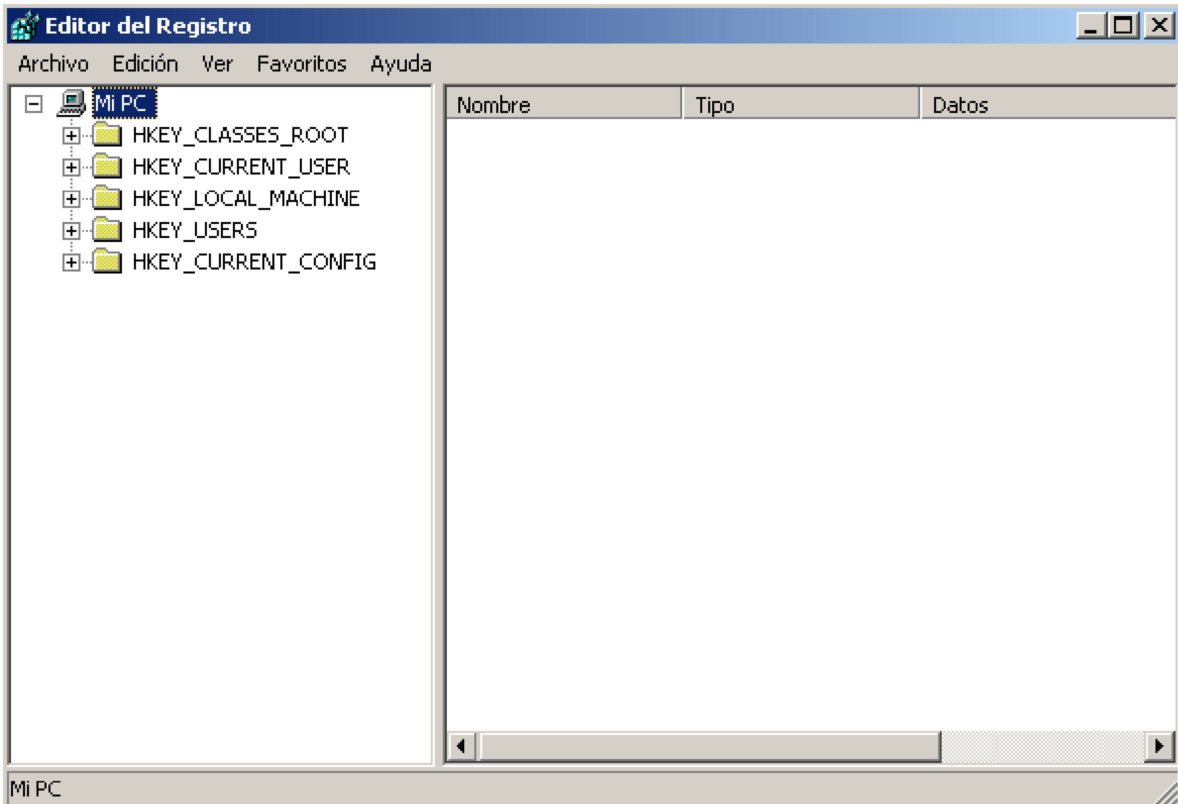
Volviendo a nuestra ventana de cadenas de texto podemos ver una referencia a una llave de registro.



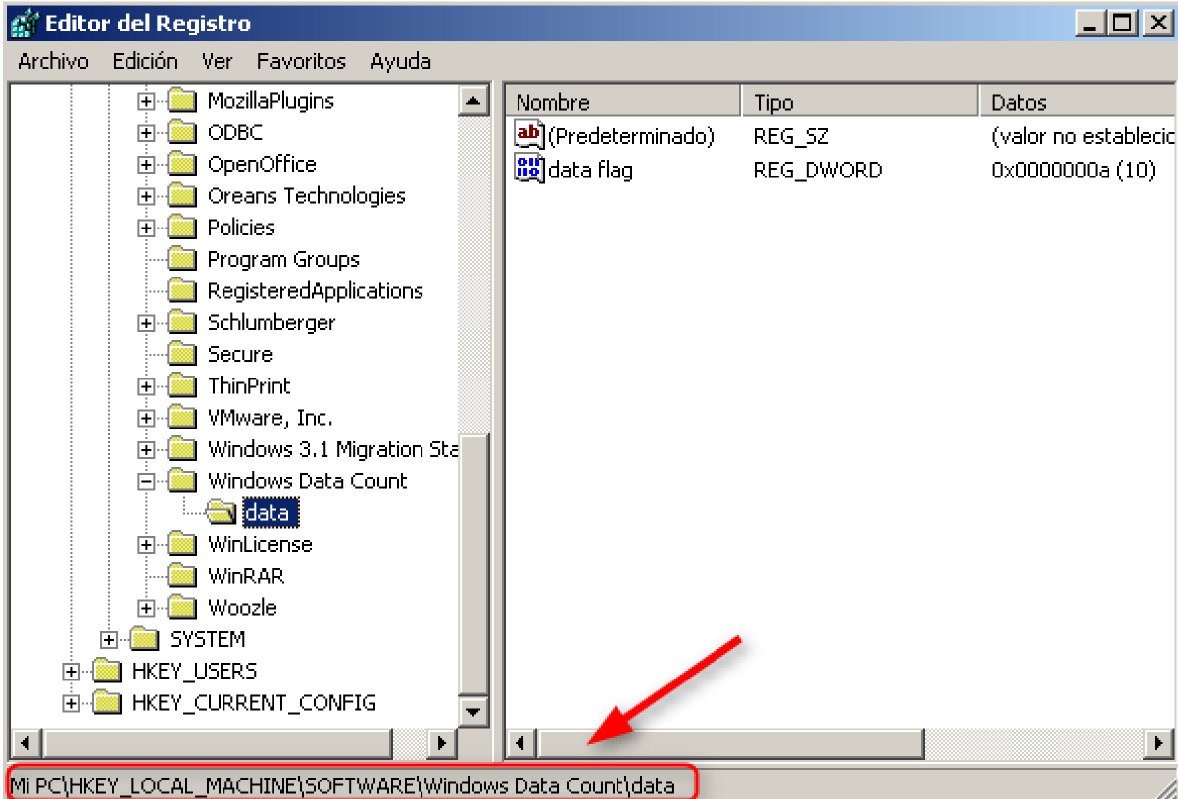
Para acceder a los regisro de Windows vamos a "Inicio" -> "Ejecutar" y escribimos 'regedit':



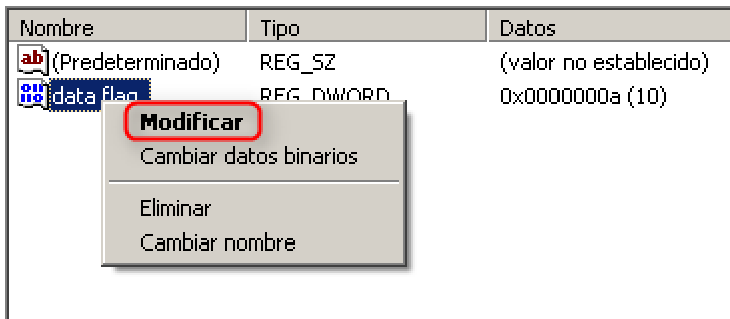
Al hacer clic en Aceptar se abre le Editor del Registro mostrandonos las cinco principales carpetas que contiene:



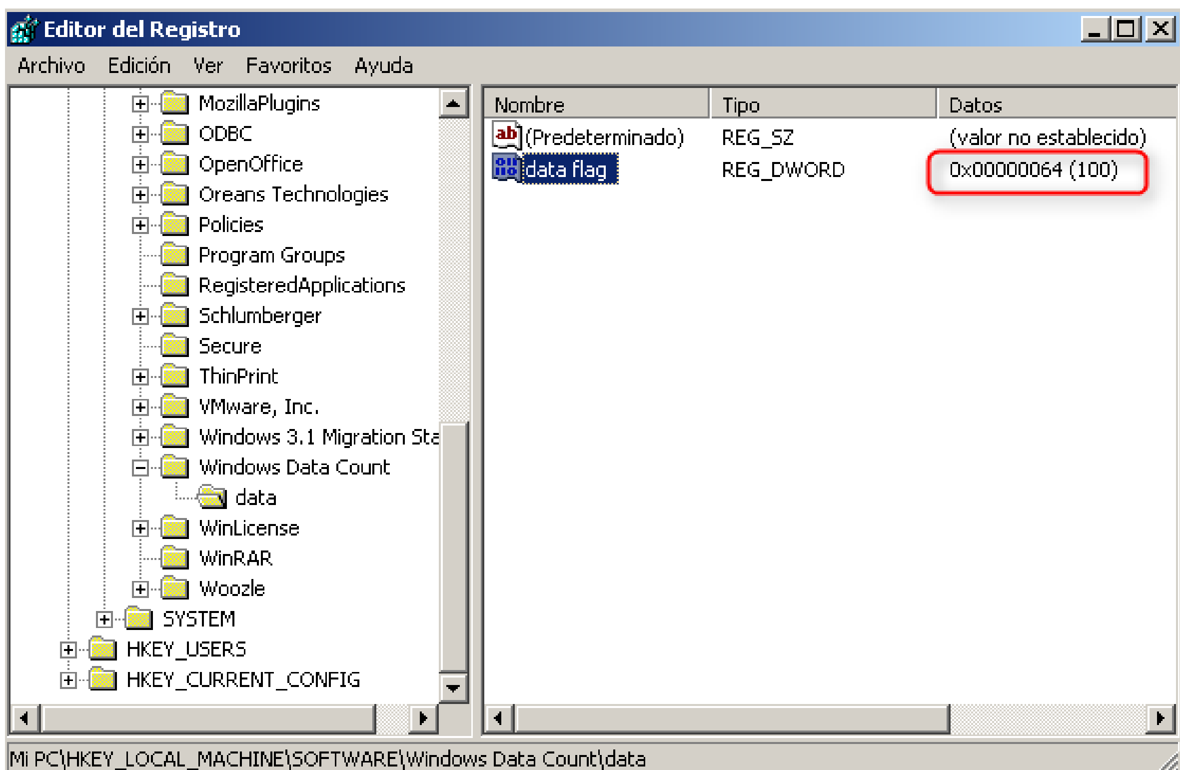
Desplegamos la carpeta 'HKEY_LOCAL_MACHINE', seleccionamos 'Software' -> 'Windows Data Count' -> 'data'



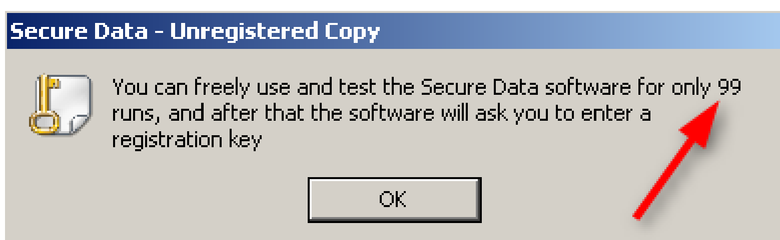
Si nos fijamos en la parte derecha de la ventana vemos que el valor de 'data flag' es igual a diez, lo que parece bastante sospechoso. Vamos a cambiarlo y mirar lo que sucede. Hacemos clic con le botón derecho y seleccionamos 'Modificar'.



Introducimos el valor de cien y guardamos:



Ejecutamos la aplicación:



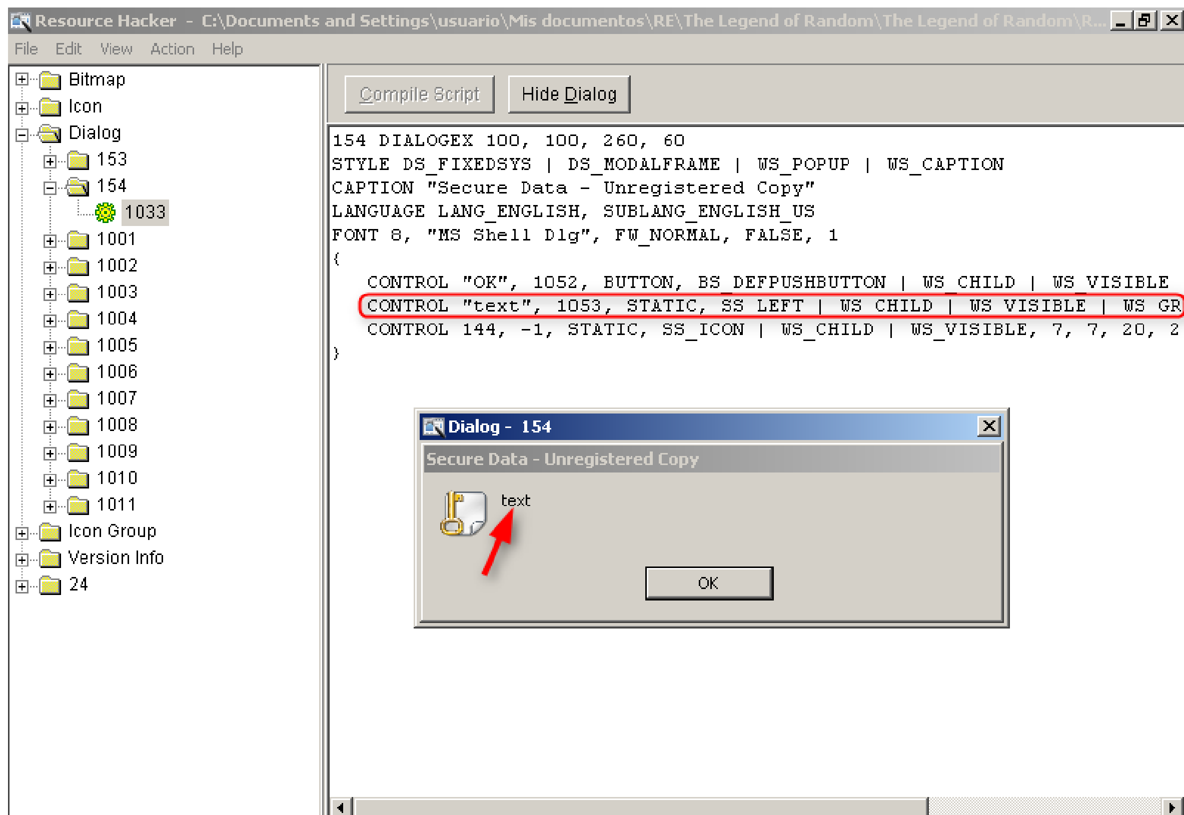
Otra forma de manipula el periodo de prueba es parcheando el código.

Volvemos a la ventana de cadenas de texto y hacemos doble clic sobre el mensaje seleccionado:

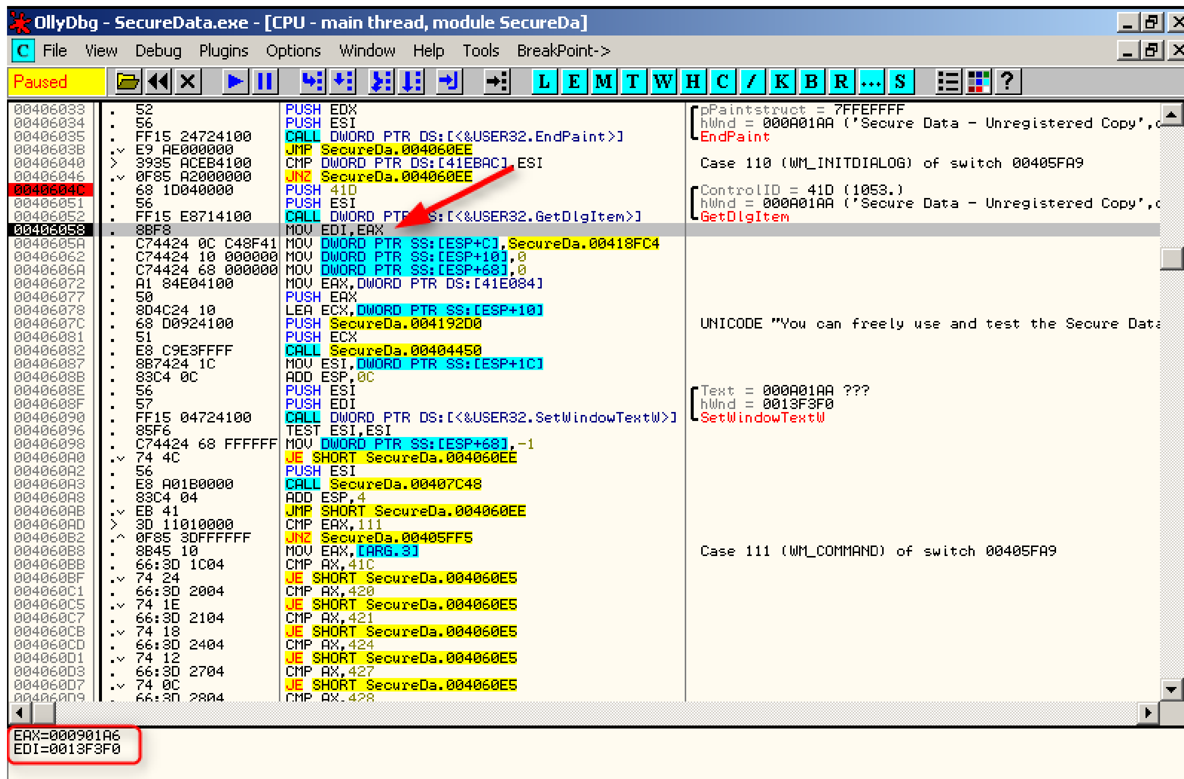
Viendo el código desde una perspectiva más amplia podemos ver que estamos ante una rutina de devolución de llamada de Windows para cuando aparezca el mensaje de WM_INITDIALOG.

Lo primero que hace es cargar un controlador con ID 0x41D (1053) para ser pasado a la API GetDegiItem. Cargando la aplicación en Resource Hacker podemos ver que el ID 1053

coresponde al texto que aparece al comienzo del nag con la información del periodo de prueba restante.



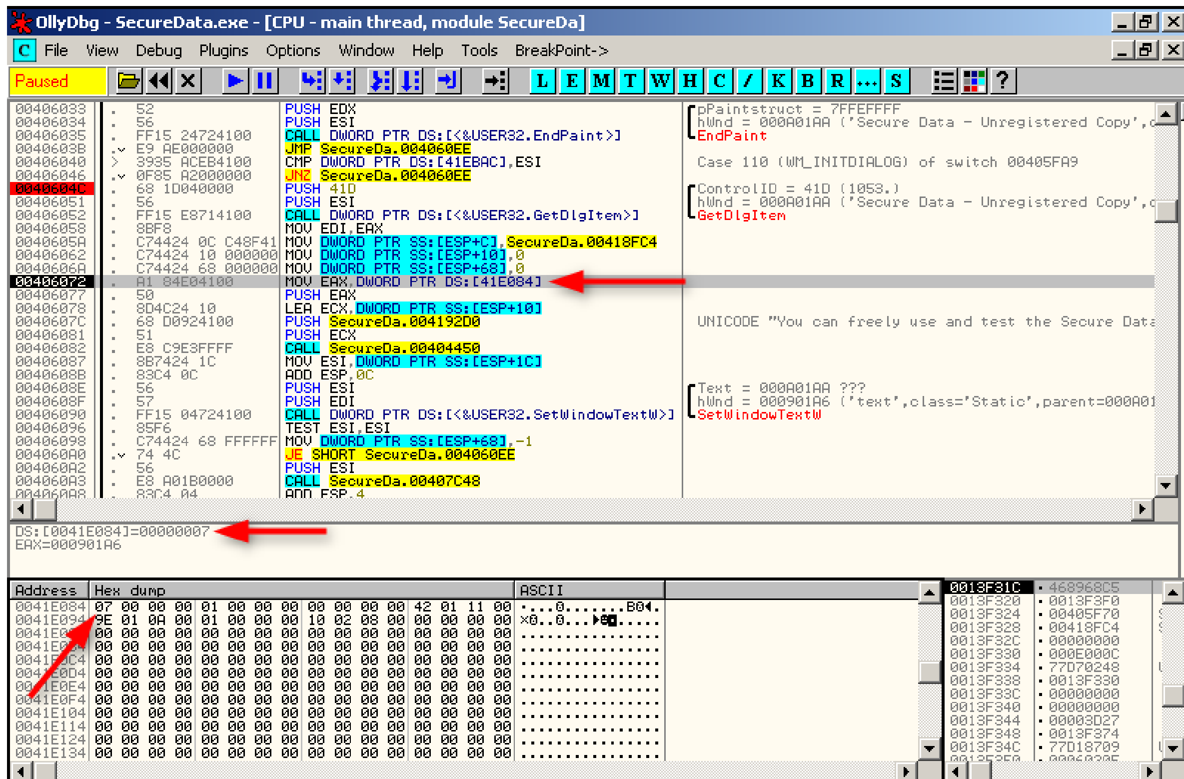
Si miramos en la ayuda de la API podemos ver que GetDigItem devuelve un controlador al control de Windows en EAX. Si ponemos un breakpoint en 40604C, reiniciamos la aplicación y pulsamos F8 hasta la dirección 406058, podemos ver que el valor de regreso es 0901A6 y se almacena en EDI:



La siguiente instrucción carga el contenido de 418FC4 en la pila en ESP+C. Si seguimos esa dirección en el dump, vemos que esa dirección almacena otra dirección, 403980, que si la seguimos podemos averiguar que se trata de una devolución de llamada. Podemos asumir que se trata de la devolución de llamada para la ventana de dialogo (por ejemplo, si se pulsa OK).

```
Address   Hex dump                                         ASCII
00418FC4  80 39 40 00 54 00 65 00 6D 00 70 00 6F 00 72 00  Ç9@.T.e.m.p.o.r.
00418FD4  61 00 72 00 79 00 46 00 69 00 6C 00 65 00 00 00  a.r.y.F.i.l.e...
00418FE4     00 65 00 6D 00 70 00 6F 00 72 00 61 00 72 00  T.e.m.p.o.r.a.r.
00418FF   79 00 46 00 69 00 6C 00 65 00 2E 00 62 00 6D 00  y.F.i.l.e...b.m.
0041  04  70 00 00 00 50 00 72 00 65 00 70 00 61 00 72 00  p...P.r.e.p.a.r.
00419014  69 00 6E 00 67 00 20 00 64 00 61 00 74 00 61 00  i.n.g. .d.a.t.a.
00419024  20 00 2E 00 2E 00 2E 00 20 00 00 00 53 00 65 00   ....... ...S.e.
00419034  63 00 75 00 72 00 65 00 44 00 61 00 74 00 61 00  c.u.r.e.D.a.t.a.
00419044  00 00 00 00 45 00 6E 00 63 00 72 00 79 00 70 00  ....E.n.c.r.y.p.
00419054  74 00 69 00 6E 00 67 00 20 00 64 00 61 00 74 00  t.i.n.g. .d.a.t.
00419064  61 00 20 00 2E 00 2E 00 2E 00 00 00 48 00 69 00  a. .........H.i.
00419074  64 00 69 00 6E 00 67 00 20 00 64 00 61 00 74 00  d.i.n.g. .d.a.t.
```

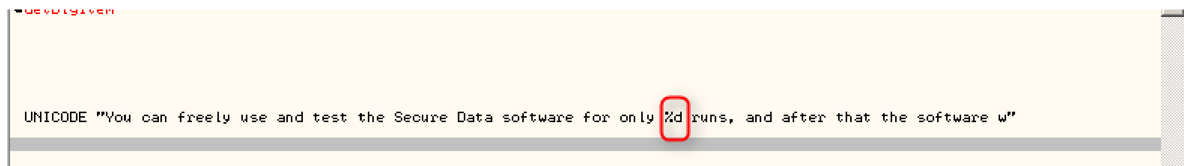
Los dos líneas siguientes empujan ceros a la pila (inizialicando algunas variables locales que serán utilizadas en el CALL) y después se carga un valor sospechoso en EAX desde el lugar en memoria 41E084:



Es sospechoso porque coincide precisamente con el número de pruebas restantes para ejecutar la aplicación. Si reiniciamos ahora la aplicación este número se vería decrementado en uno por haber utilizado una prueba más. Podemos resumir que la dirección 41E084 es el lugar donde se almacenan los intentos restantes para poder ejecutar la aplcación.

Finalmente la aplicación carga ese valor como un puntero hacia la cadena en la pila y luego hace un CALL. La razón de este CALL, en la dirección 406082 es para comprobar el número de intentos que restan por hacer e insertar el resultado en la cadena de texto.

Vemos que la cadena de texto en lugar de un valor contiene un símbolo:



Sabemos de programación que %d es una expresión de tipo *printf*. Como el número restante de pruebas es dinámico tenemos que crear una cadena generica para insertar el numero actual conforme se vaya ejecutando la aplicación. Si pulsamos F8, pasando el CALL, podemos ver que se ha insertado el valor en la cadena:

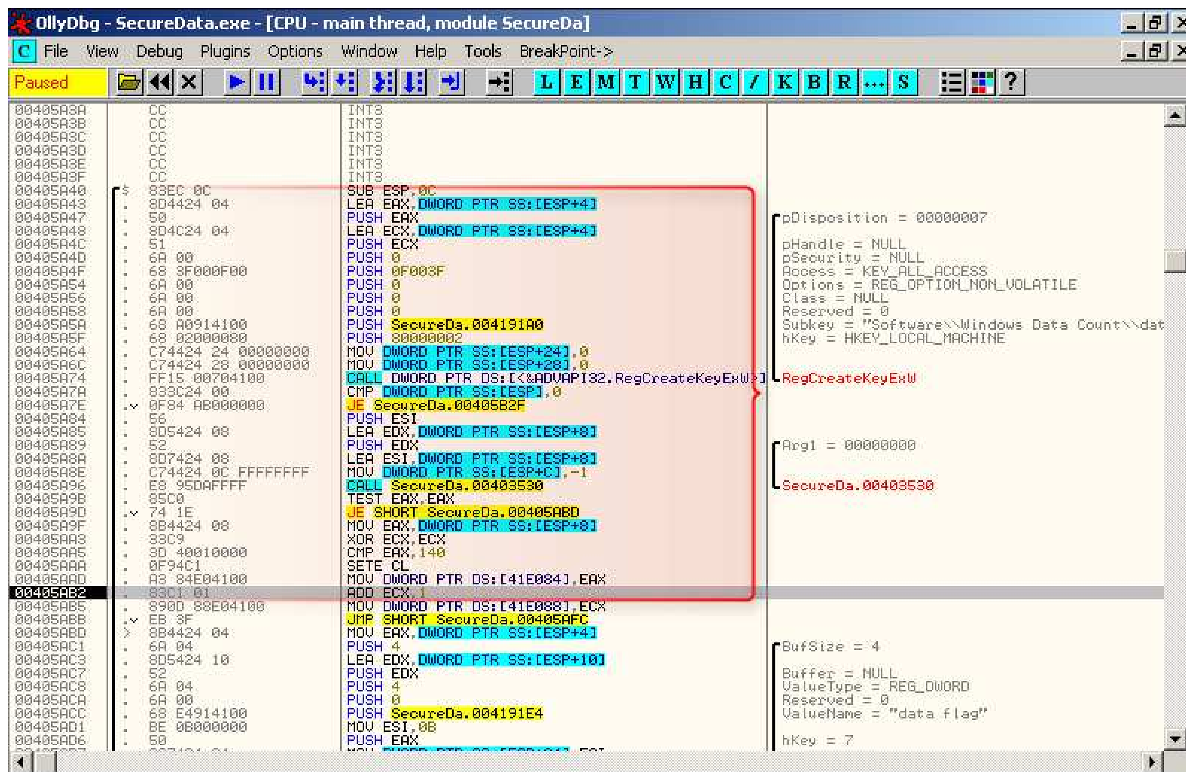
Ahora podemos afirmar con total seguridad que la dirección mencionada más arriba contiene el número de intentos restantes… Si finalizamos el area de este código, el nag volverá aparecer, esperando a que hagamos clic en OK.

Para parchea la aplicación vamos a seguir la dirección 41E084 en el dump para poner un “hardware breakpoint”. De esta forma conseguiremos que Olly se pare siempre que un nuevo valor es almacenado en dicha dirección:

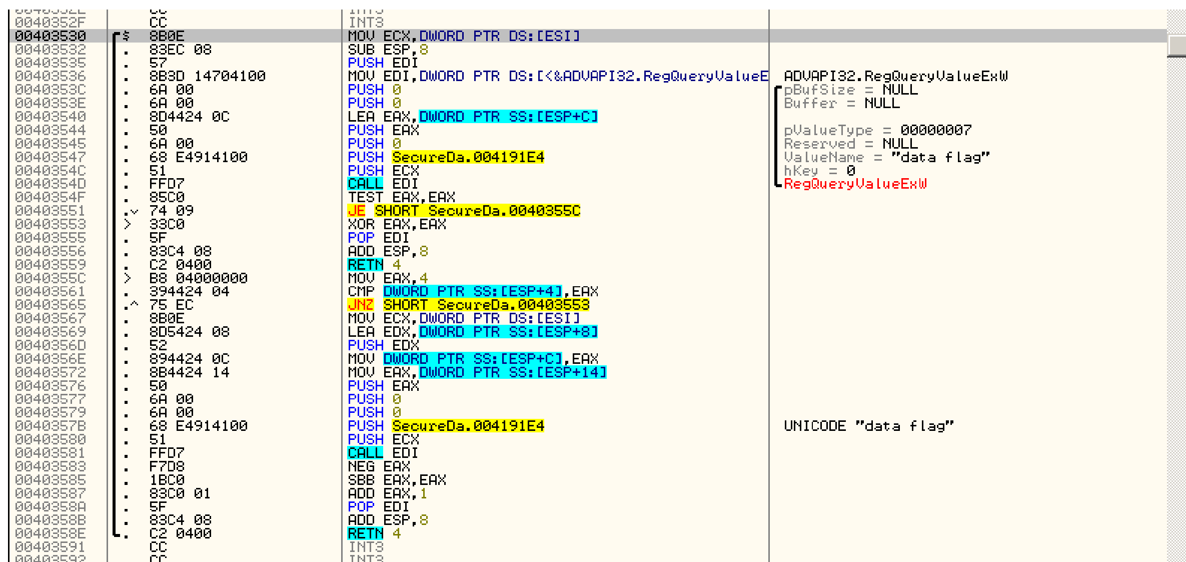
Hemos seleccionado Byte porque es la longitud de la variable. Reiniciamos la aplicaicón y Olly se detendrá en nuestro hardware breakpoint:

Hardware breakpoint 1 at SecureDa.00405AB2 - EIP points to next instruction

A continuación vemos el area que cubre el inicio de la rutina y nuestro hardware breakpoint.



En el comnienzo de la rutina vemos hacia donde se empuja el valor del registro que se encuentra en el directoio de Windows Data Count. (RegCreateKeyExW no solo se usa para crear una llave, sino también para abrirla). A continuación se le aplica algunos procesamientos alrededor de la dirección 405A7E, para comprobar si el valor de regreso es cero (lo que significaría que la aplicación no le estaría permitido acceder al registro) y se salta a un posible "bad boy", en la dirección 405B2F, indicandonos que necesitamos tener privilegios de administrador para acceder a esta llave.Si no hay error, la llamada en 405A96 cargará el valor de la llave, devolviendo dicho valor a ESP+C (que se convertirá en ESP+8 después del regreso):



Continuamos cargando el valor retornado en nuestra variable en la dirección 405AAD

Finalmente comprobamos más valores, cerramos la llava y regresamos:

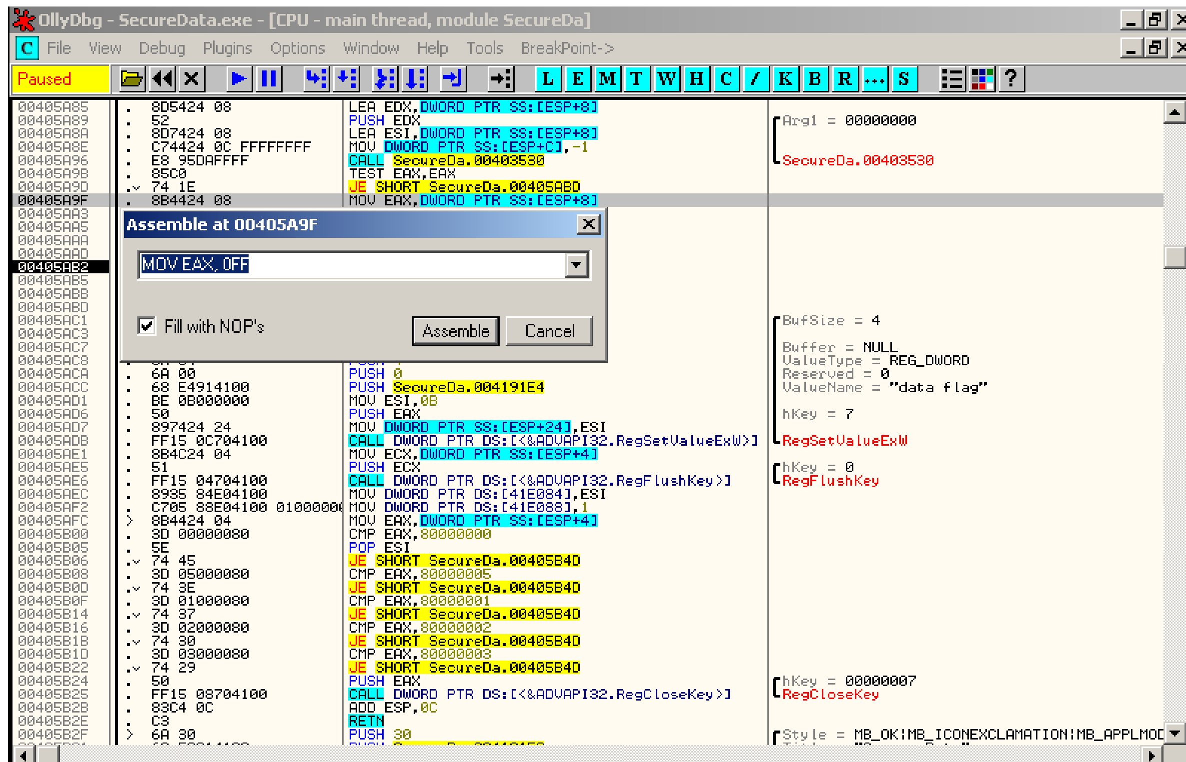
La pregunta que se plantea ahora es ¿Cual es el mejor sitio para parchear esta aplicación?

Si cojemos la dirección en 405AAD, donde el valor que se recibió del registro es almacenado en el lugar de la memoria para comprobar más tarde el número de intentos, el

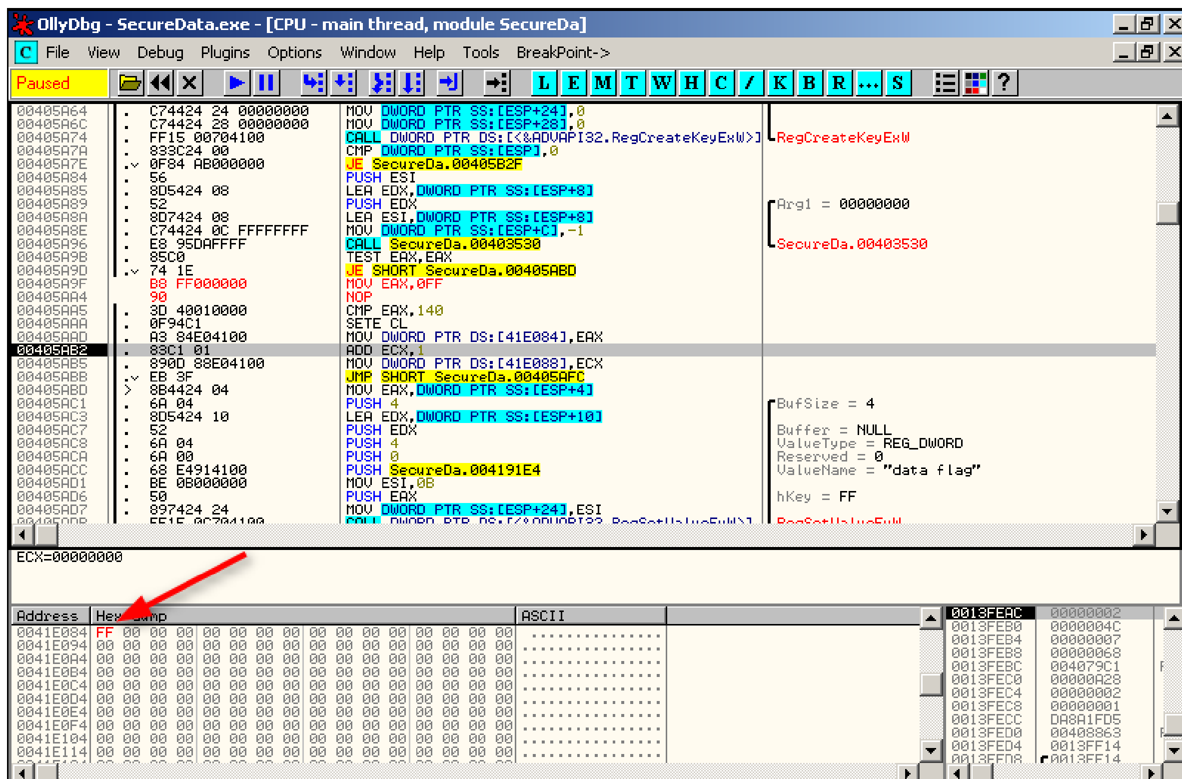
problema viene si intentamos parchear esto con algo así como MOV DWORD PTR DS:[41E084], 0FF el código resultante va eliminar las dos siguientes instrucciones:

```
00405AA3    .  33C9             XOR ECX,ECX
00405AA5    .  3D 40010000      CMP EAX,140
00405AAA    .  0F94C1           SETE CL
00405AAD       C705 84E04100 FF MOV DWORD PTR DS:[41E084],0FF
00405AB7       90               NOP
00405AB8       90               NOP
00405AB9       90               NOP
00405ABA       90               NOP
00405ABB    .- EB 3F            JMP SHORT SecureDa.00405AFC
00405ABD    >  8B4424 04        MOV EAX,DWORD PTR SS:[ESP+4]
00405AC1    .  6A 04            PUSH 4
00405AC3    .  8D5424 10        LEA EDX,DWORD PTR SS:[ESP+10]
00405AC7    .  52               PUSH EDX
00405AC8    .  6A 04            PUSH 4
```

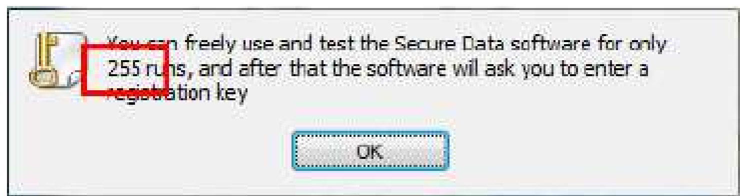
Así que parcheemos la instrucción anterior en 405A9F, donde EAX es cargado con los números de intentos devueltos:



Si pulsamos F8 podemos ver que nuestra variable ahora contiene el valor 0xFF en lugar del número real de intentos por realizar:



Una vez guardado el parche, siempre tendremos 255 posibilidades de ejecutar la aplicación independientemente del número de veces que hayamos ejecutado la aplicación anteriormente.



Conclusión: eliminar el nag puede que fuese la mejor solución para esta aplicación pero

1. Es buenos saber reconocer este tipo de sistemas de protección.

2. A veces no se puede crackear una aplicación, y este método es la siguiente mejor opción.

7.18 Caso práctico 18: Generador de parches

Un generador de parches es un programa que un ingeniero inverso puede utilizar para parchear una copia virgen de una aplicación. Se trata pues de pequeños programas que son enviados junto a un programa sin modificar (por ejemplo, el típico programa que se descarga de la web del proveedor), para poder aplicar los parches correspondientes y hacer así uso de una aplicación nueva parcheada.

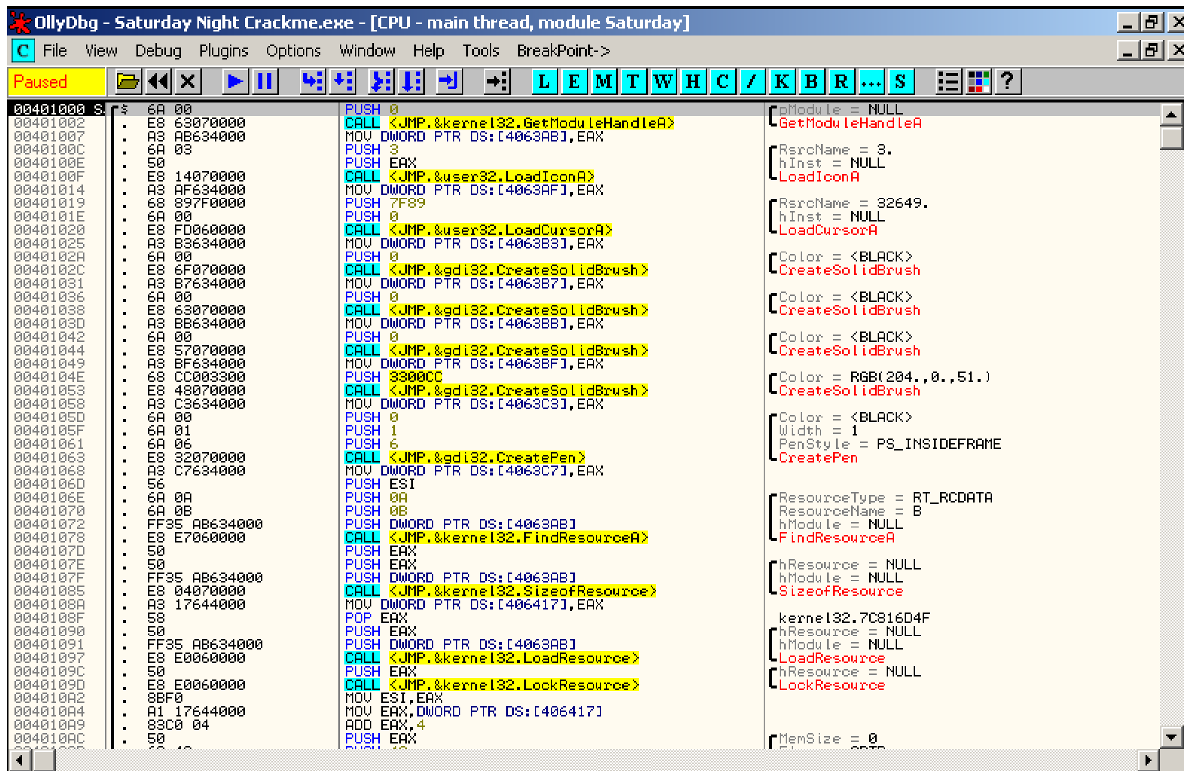
En este ejercicio vamos a parchear el crackme "Saturday Night Crackme". Utilizaremos dUP2 un generador de parches creado por Diablo2002, así como CFF Explorer.

Ejecutamos el programa haciendo doble clic sobre el ejecutable.

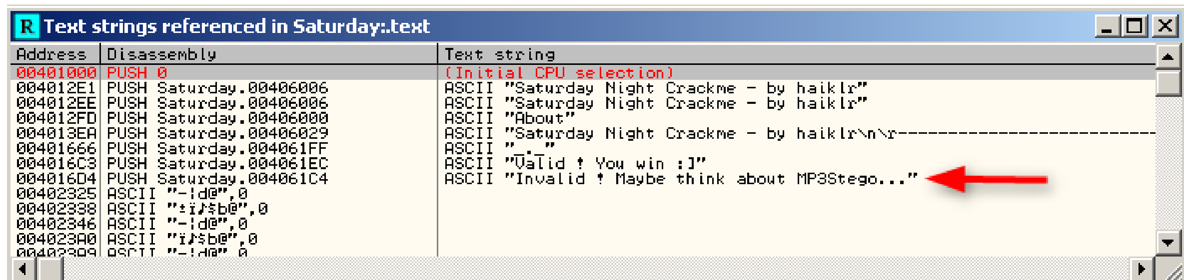
Introducimos un código:

Haciendo clic en Dance, vemos al “bad boy”.

Vamos a cargar la aplicación en Olly:



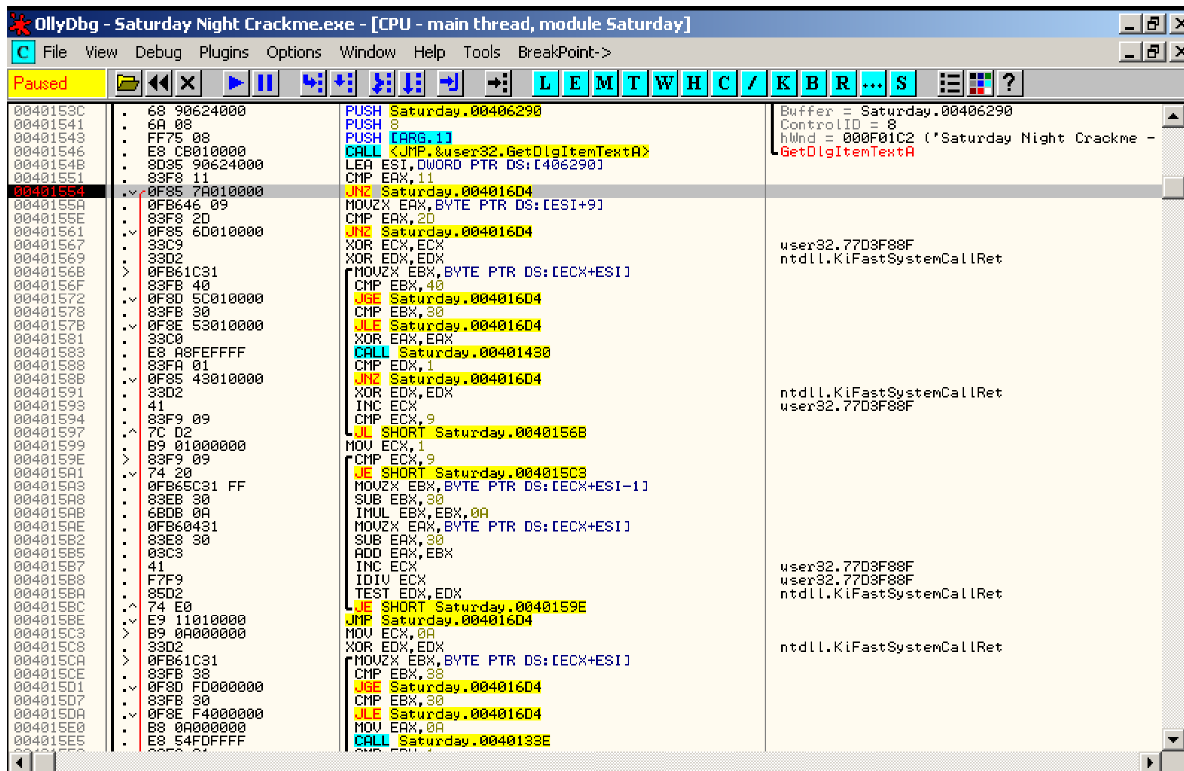
Y buscamos por cadenas de texto:



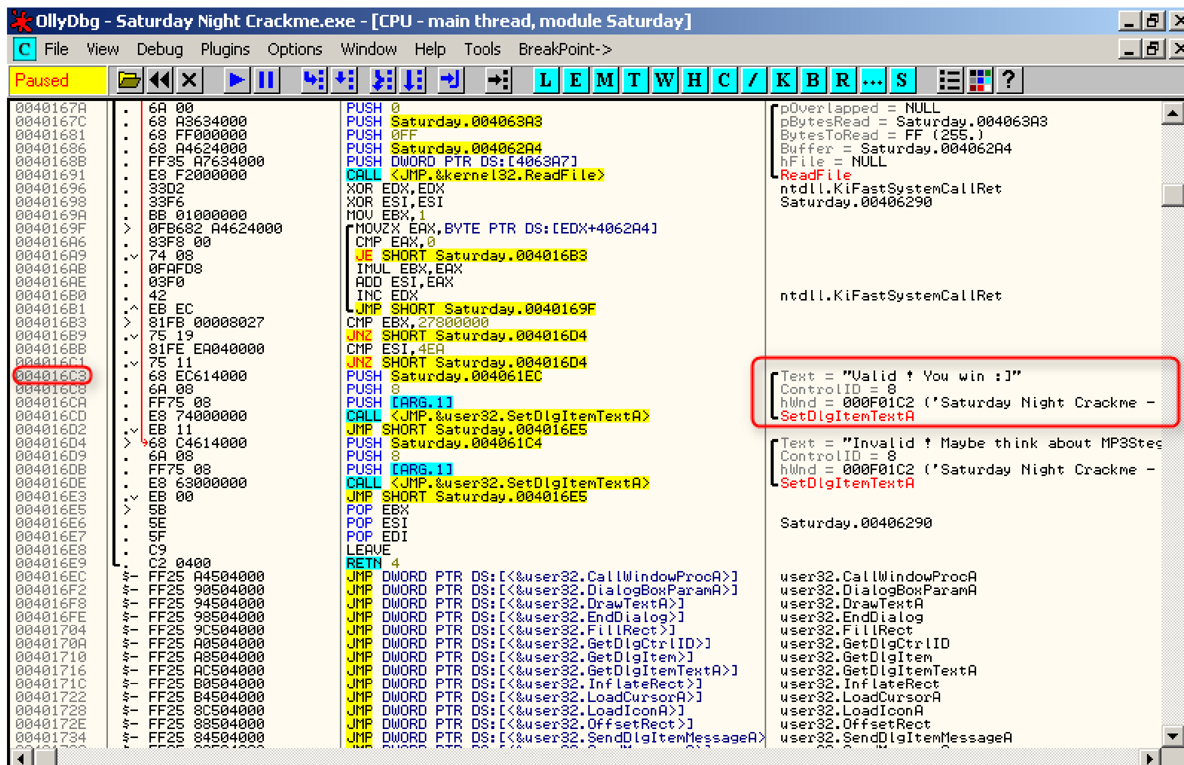
Hacemos doble clicl sobre el “bad boy” y saltamos de lleno al área que nos interesa.

Subimos un poco para averiguarl el lugar desde el cual es llamado el “bad boy”.

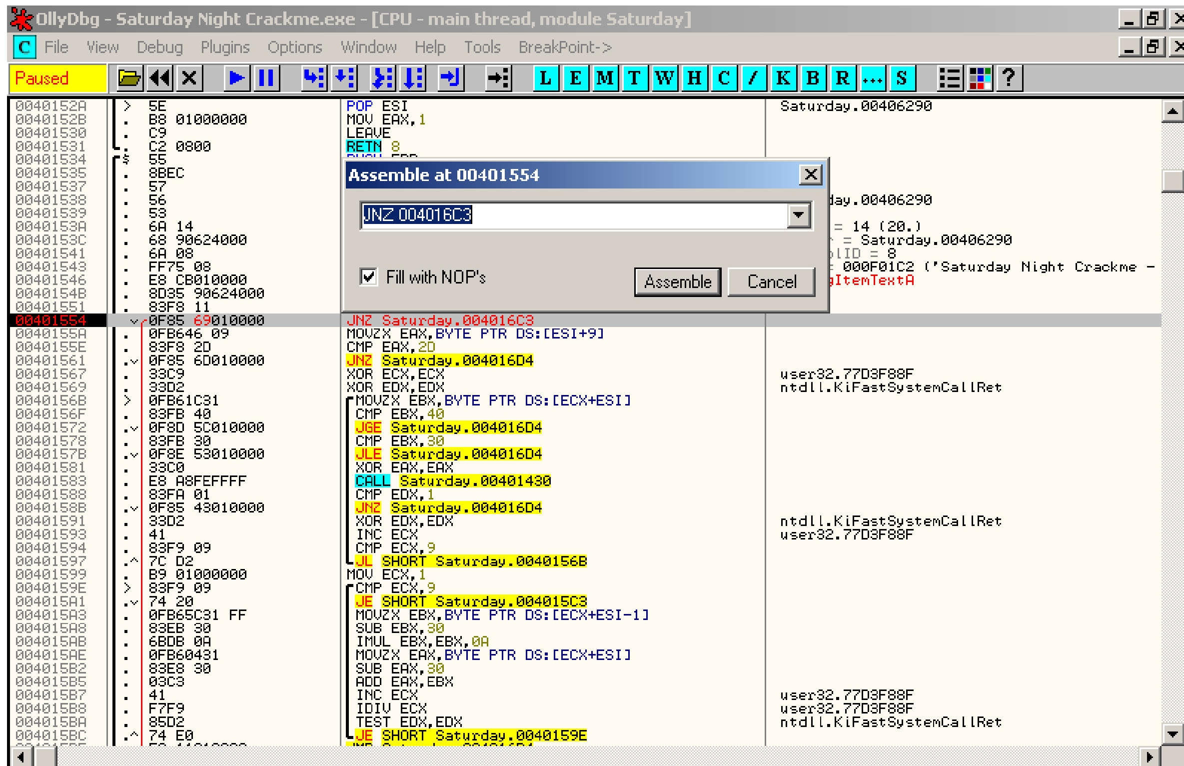
Ponemos un Breakpoint en la dirección 401554, reiniciamos la aplicación y nos detenemos en la instrucción JNZ.



Normalmente lo que haríamos a continuación es cambiar el valor de la bandera Z para evitar el salto y ejecutar línea por línea para ver si aparece el “bad boy”. Una vez hecho esto volveríamos atrás para parchear todos los saltos que van hacia el “bad boy”, cambiándoles a **<u>no</u>** saltar. Pero como esta vez nuestro objetivo es simplemente llegar al “good boy” cambiaremos la instrucción JNZ por otra que nos haga saltar siempre a nuestro “good boy”. Vemos que la dirección del “good boy” es 4016C3.



Vamos a la instrucción en la dirección 401554 y cambiamos el salto hacia el “bad boy” por un salto hacia el “good boy”.



Pulsamos F9 y vemos que independientemente del código introducido siempre llegaremos a nuestro “good boy”.

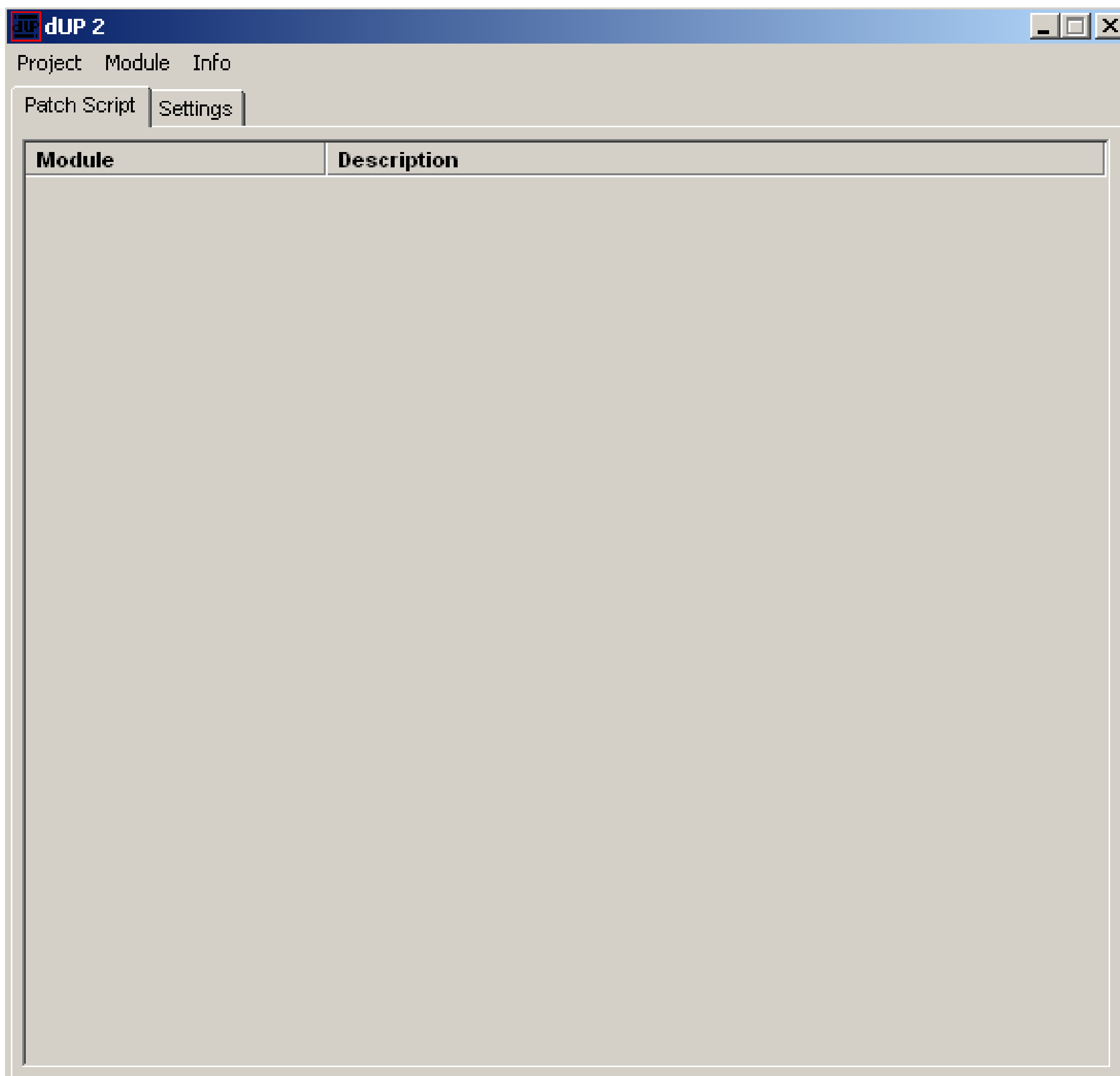
Introducción a dUP2

Hay básicamente dos maneras de crear un parche:

1. Parche de compensación: En este caso debemos saber el sitio exacto del código a parchear. Esta opción se suele usar para determinar manualmente el lugar del parche. Para realizar este tipo de parches, debemos introducir la distancia desde el principio del archivo a parchear, junto con las modificaciones y dUP2 creará una aplicación que ejecutará los parches que hemos introducidos.

2. Busca y reemplaza: se utiliza si sabemos la instrucción que queremos cambiar, pero no sabemos el lugar exacto de esa instrucción dentro del programa, o si el programa es auto-modificable, las áreas para parchear cambian cada vez que se ejecuta el programa. En este caso primero se buscan las cadenas de instrucciones, y cuando se hayan encontrado, se reemplazan con nuestras modificaciones.

Ejecutamos dUP 2 y aparece la ventana principal:



Seleccionamos “Proyect” -> “New”

Patch Info

Patcher Caption: Patch

Application

Filename (s) ...

URL visit

Author: Someone

Release Date: August 16, 2015 today

Release Info

About Box Message: created with dUP2 http://diablo2oo2.cjb.net

Scrolltext

- [x] Show this dialog when create a new project
- [ ] Run patch with administrator rights
- [ ] No Backup by default

Cancel Save

Rellenamos la ventana con algunos atributos que finalmente va a tener nuestro parche:

Patch Info

Patcher Caption: Patch

Application: Saturday Night Patcher

Filename (s): Saturday Night Crackme.exe ...

URL: visit

Author: manuel

Release Date: August 16, 2015 today

Release Info

About Box Message: For more patches contact with,
manuel.rey.vilar@gmail.com

Scrolltext

- [x] Show this dialog when create a new project
- [ ] Run patch with administrator rights
- [ ] No Backup by default

Cancel Save

Hacemos clic en “Save” y se abre otra ventana:

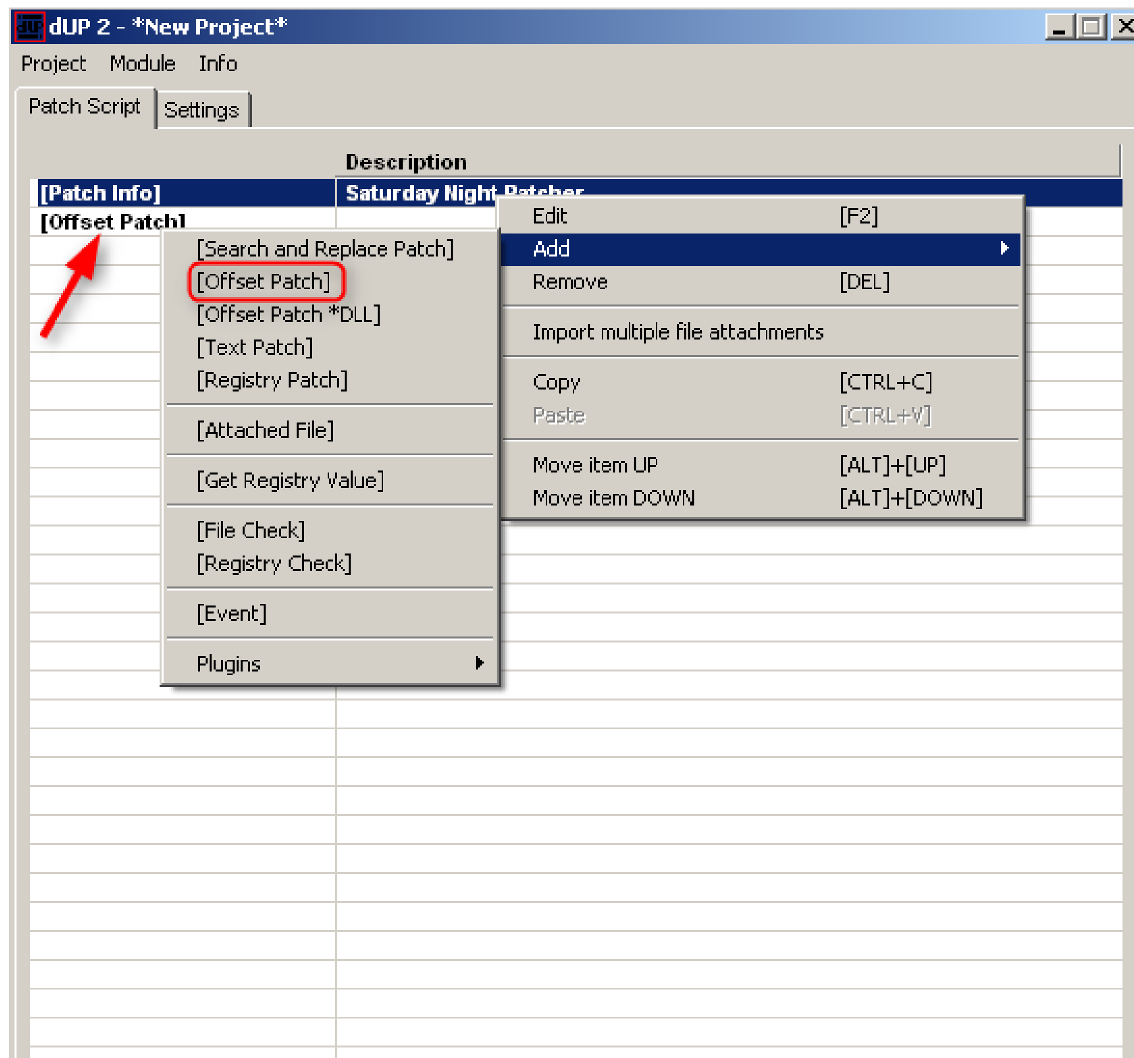
dUP 2 - *New Project*

Project Module Info

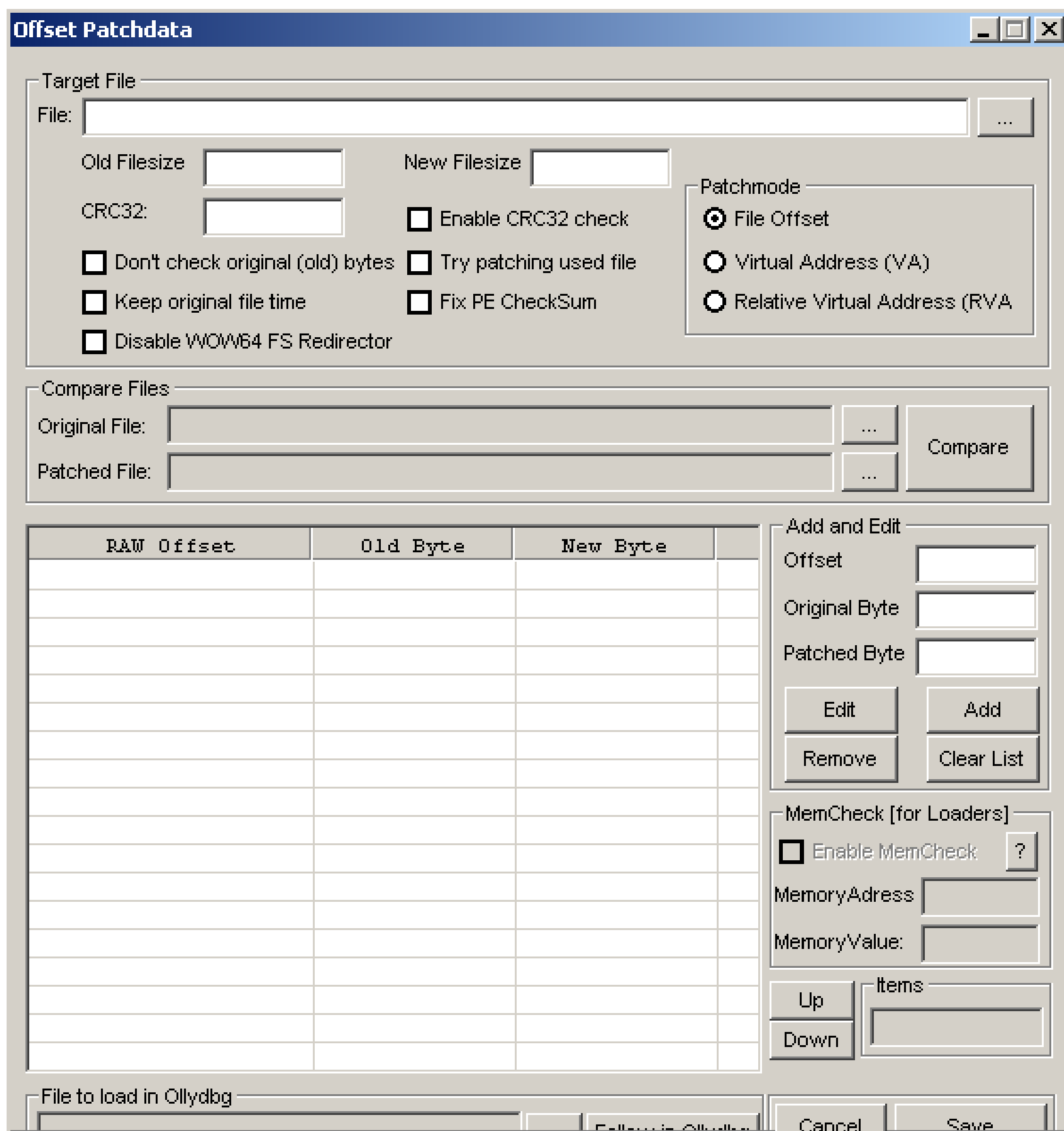
Patch Script | Settings

Module	Description
[Patch Info]	Saturday Night Patcher

A continuación vamos añadir el parche. Hacemos clic con el botón derecho sobre la línea Saturday Night Patcher, y seleccionamos “Add” -> “Offset Patch”. Esto añadirá una nueva línea a nuestro poryecto:



Haciendo doble clic sobre la nueva línea nos abrirá la ventana principal del parche offset:



Aquí es donde introduciremos toda la información de nuestro parche:

En File, nuestro ejecutable. Los cuadros de texto Old Filesize, New Filesize y CRC32 se rellenan cuando se trate de una aplicación que utilize la comprobación CRC (Cyclic Redundancy Check). Este método comprueba cada byte en el ejecutable cuando es ejecutado por primera vez y así asegurarse de que ningún byte haya sido alterado con respecto a la primera vez que aparecio la aplicación en el mercado.A continuación usará un simple algoritmo para crear una llave CRC única. Si algún byte es cambiado, la llave va ser cambiada.

El grupo ‘Patchmode’ nos permite elegir entre un offset basado en un archivo binario, un offset basado en una dirección virtualo o un RVA. En nuestro caso vamos a utilizar el que viene por defecto.

El cuadro ‘Compare Files’ permite comparar dos archivos, el original y el parcheado para crear un parche basado en la diferencia entre ambos. No vamos a utilizar esta caracteristica puesto que sabemos como va a ser nuestro parche.

En el grupo 'Add and Edit' es el area principal de los datos. Aquí introduciremos cada offset de nuestro parche, el valor del byte original y el valor del byte nuevo.

Reiniciamos la aplicación y nos situamos en el código parcheado en la dirección 401554. Si nos situamos en la columna de los opcodes podemos ver que el número original de bytes es de 0F85 7A01 0000.

```
0040154B  |.  8D35 90624000   LEA ESI,DWORD PTR DS:[406290]
00401551  |.  83F8 11         CMP EAX,11
00401554  |.v 0F85 7A010000   JNZ Saturday.004016D4
0040155A  |.  0FB646 09       MOVZX EAX,BYTE PTR DS:[ESI+9]
0040155E  |.  83F8 2D         CMP EAX,2D
00401561  |.v 0F85 6D010000   JNZ Saturday.004016D4
00401567  |.  33C9            XOR ECX,ECX
00401569  |.  33D2            XOR EDX,EDX
```

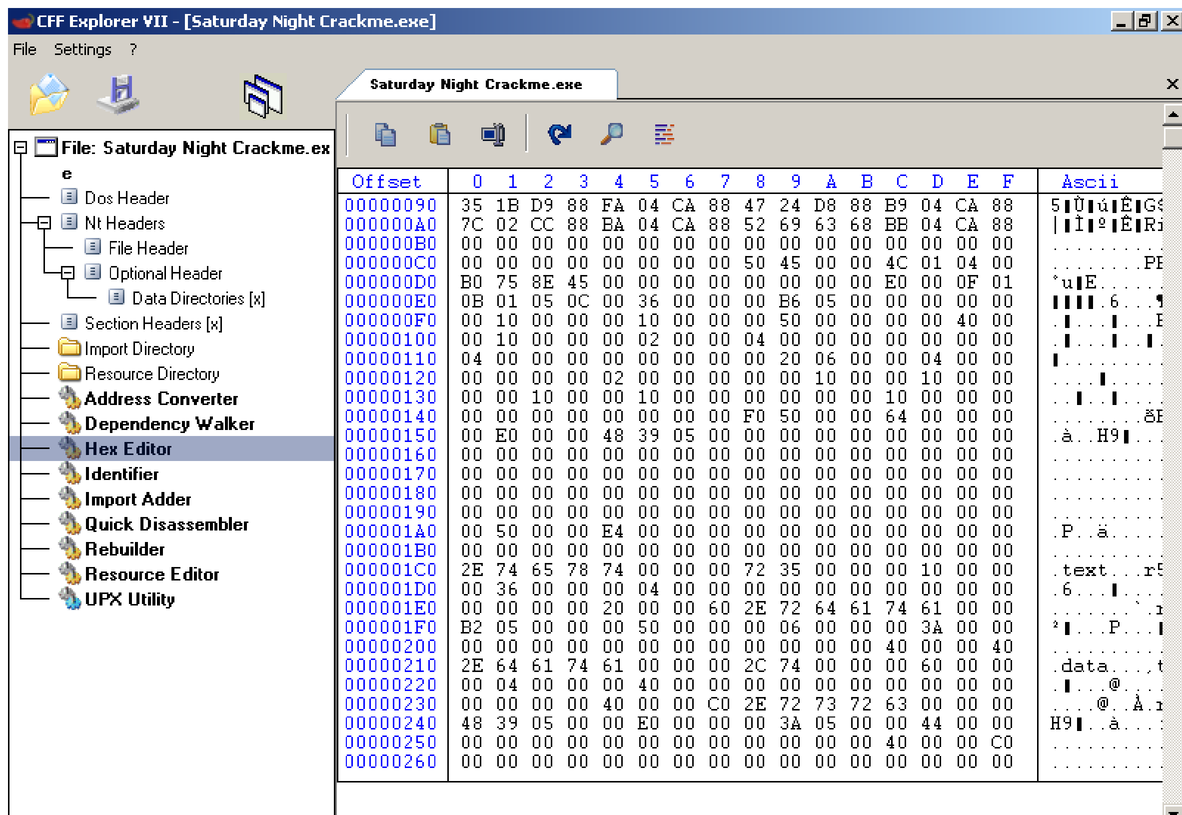
Si activamos el parche en Olly vemos que el número de bytes en esa dirección es:

```
00401554  |.v 0F85 69010000   JNZ Saturday.004016C3
```

Vemos que solo hemos cambiado un byte; de 7A a 69. Para parchear esta aplicación simplemente tenemos que cambiar el 7A en la dirección 401554 a un 69.

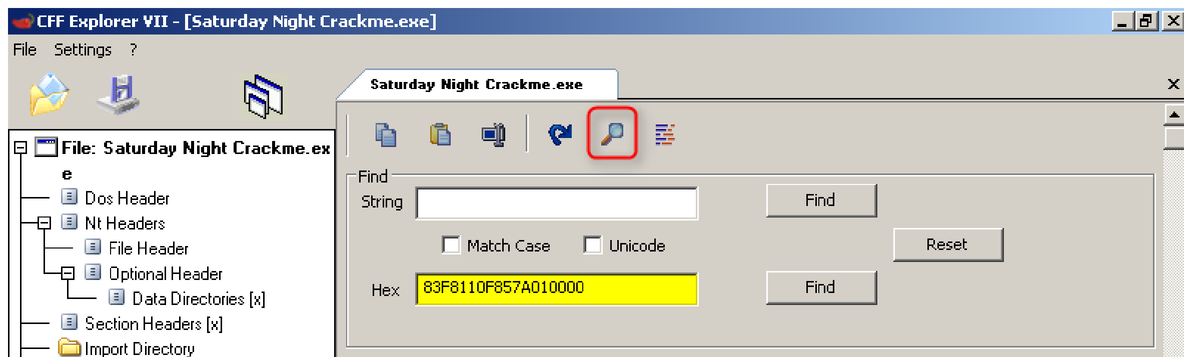
Lo siguiente que tenemos que hacer es buscar la compensación dentro del binario de nuestro parche. Dado que el lugar de nuestro parche en el binario va a ser diferente a la dirección de la memoria después de que la aplicación haya sido cargada, tendremos que buscar el lugar real de nuestro binario.

Para ello utilizaremos el editor hexadecimal CFF Explorer. Abrimos la aplicación en dicho editor y hacemos clic en la opción 'Hex Editor':

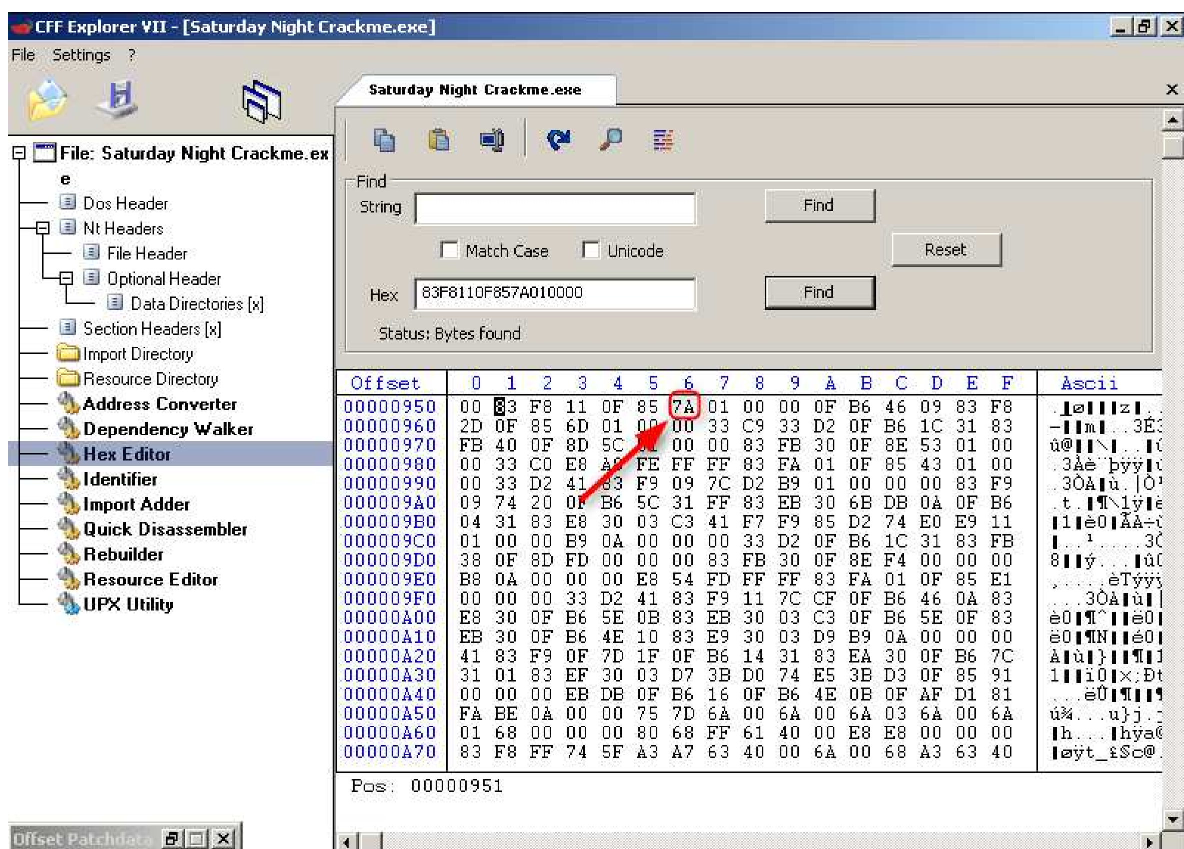


Hacemos clic en el icono de buscar e introducimos los valores hexadecimales que estamos buscando. Para evitar que nos salga código duplicado añadiremos los opcodes de la

dirección anterior a nuestro parche; 401551. De esta forma buscariamos el valor hexadecimal: 83F8110F857A010000.



Hacemos clic en 'Find'. CFF nos mostrará donde se encuentran estos valores:



Vemos que el offset para el byte que necesitamos cambiar es el 956 (6 bytes después del offset 950).

Volvemos a dUP2 e introducimos los valores. Para Offset: 956, para Original Byte: 7A y para Patched Blyte: 69.

Add and Edit

Offset 956

Original Byte 7A

Patched Byte 69

Edit | Add

Remove | Clear List

Despues de hacer clic en ‘Add’ podemos ver nuestro parche en la lista de los parches:

RAW Offset	Old Byte	New Byte
00000956	7A	69

Hacemos clic en ‘Save’ y añadimos nuestra información a al ventana principal:

Offset Patchdata
Target File
File: Saturday Night Crackme.exe
Old Filesize 00057E00
New Filesize
CRC32: 57BF47F2
Enable CRC32 check
Patchmode
File Offset
Virtual Address (VA)
Relative Virtual Address (RVA
Don't check original (old) bytes
Try patching used file
Keep original file time
Fix PE CheckSum
Disable WOW64 FS Redirector
Compare Files
Original File:
Patched File:
Compare
RAW Offset
Old Byte
New Byte
00000956
7A
69
Add and Edit
Offset
Original Byte
Patched Byte
Edit
Add
Remove
Clear List
MemCheck [for Loaders]
Enable MemCheck
?
MemoryAdress
MemoryValue:
Up
Down
Items
1
File to load in Ollydbg
Follow in Ollydbg
Cancel
Save

dUP 2 - *New Project*

Project Module Info

Patch Script | Settings

Module	Description
[Patch Info]	Saturday Night Patcher
[Offset Patch]	Saturday Night Crackme.exe

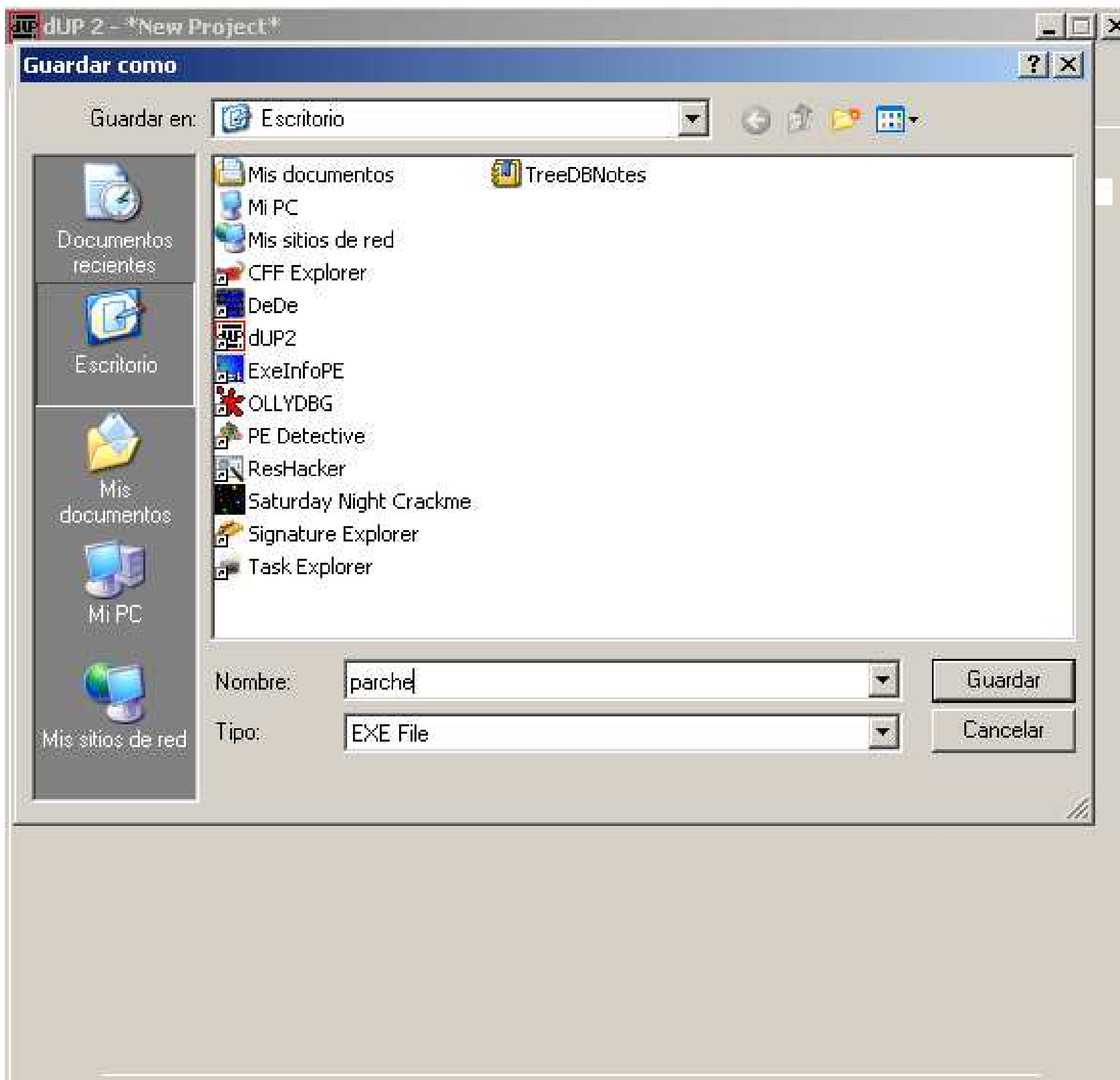
A continuación seleccionamos ‘Proyect’ -> ‘Create Patch’.

dUP 2 - *New Pro

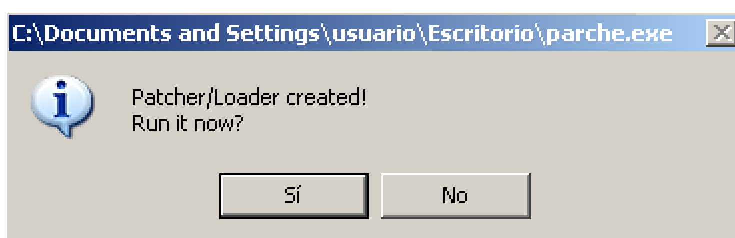
Project Module Inf

New
Open
Save
Save As
Close
Create Patch
Create Loader
Recent
Exit

Introducimos el nombre de nuestro parche y guardamos:

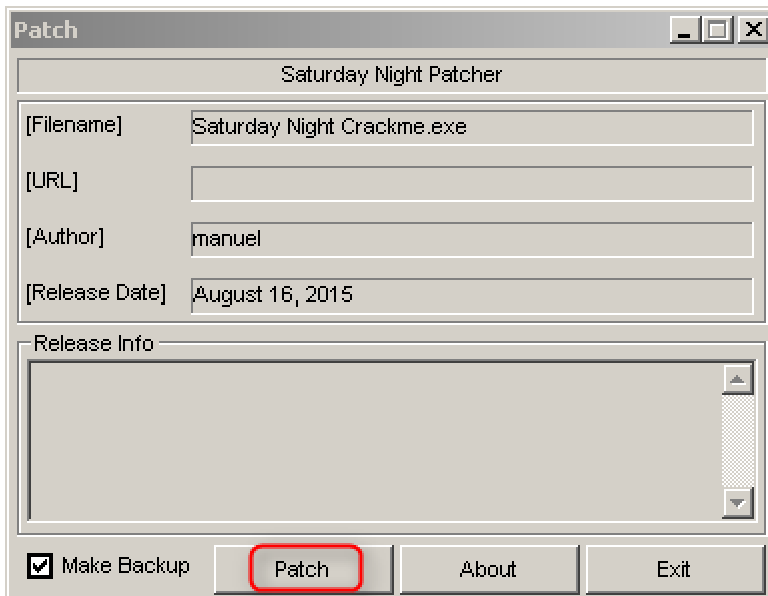


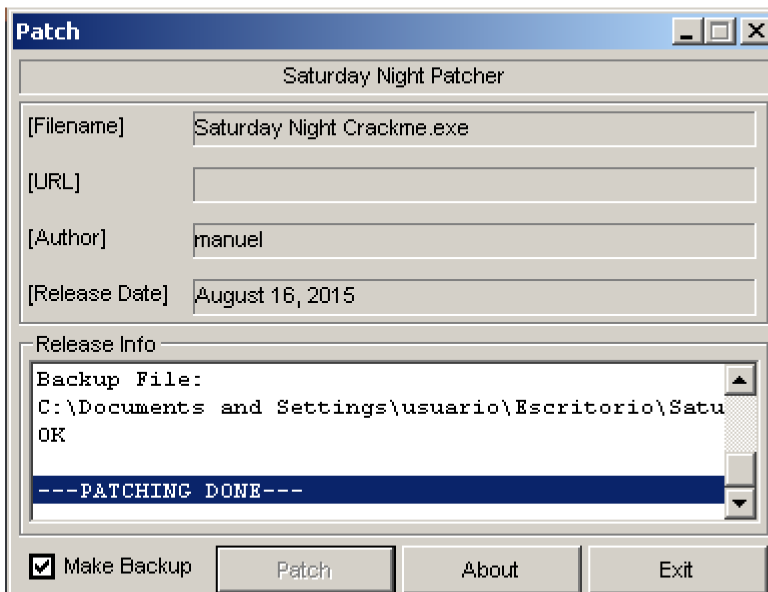
Nos pregunta si queremos ejecutarlo ahora, y le decimos que **NO**.



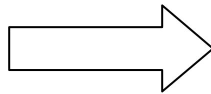
Vemos que tenemos ahora un parche para nuestra aplicación. Hacemos doble clic en el parche:

Se habre la ventana del parche. Hacemos clic en 'Patch'.



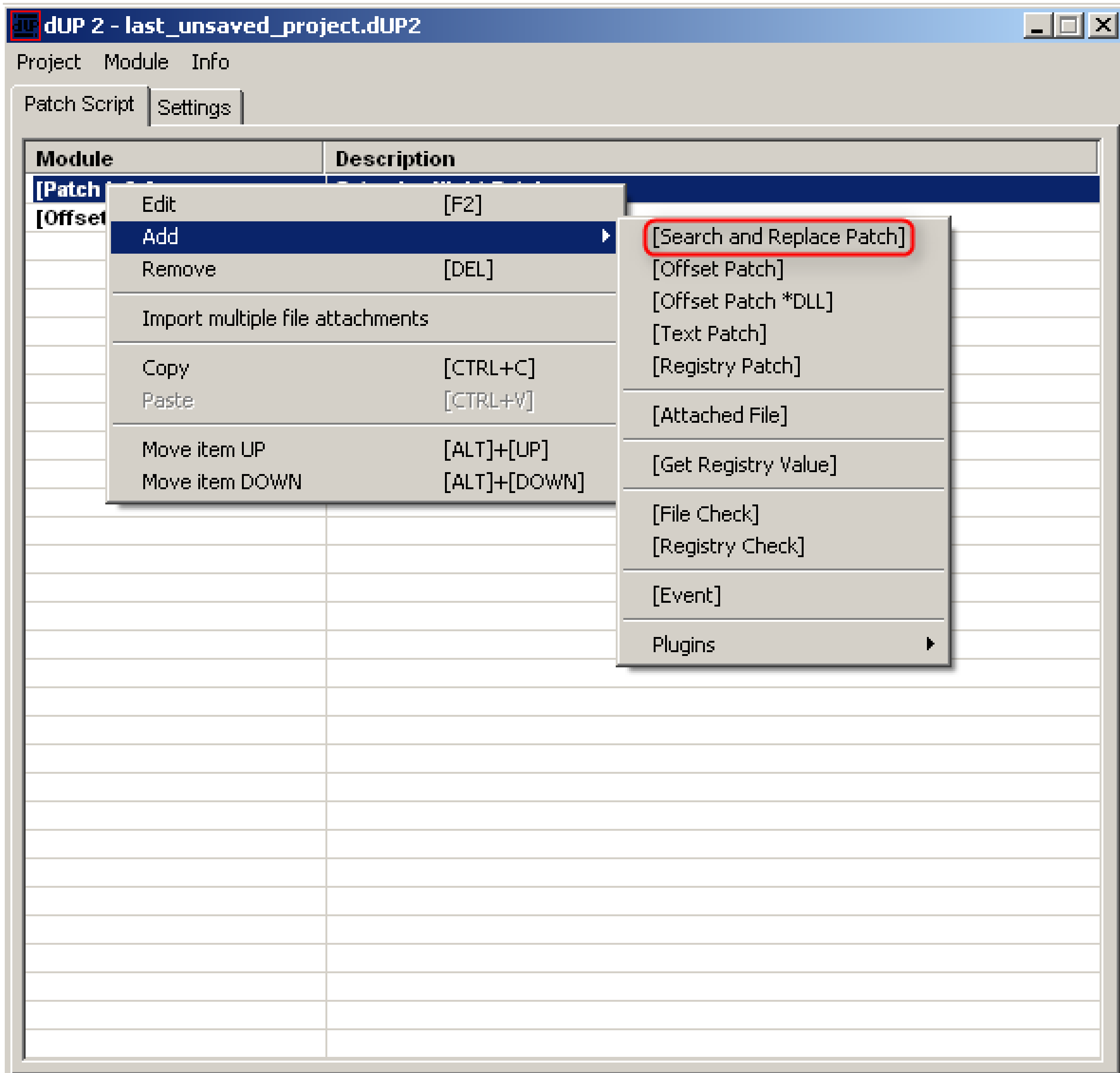


Vemos que el parche ha sido creado con éxito. También vemos que ha sido creado una copia de seguridad de la aplicación. Si ahora ejecutamos nuestra aplicación vemos que la hemos parcheada de forma satisfactoria.



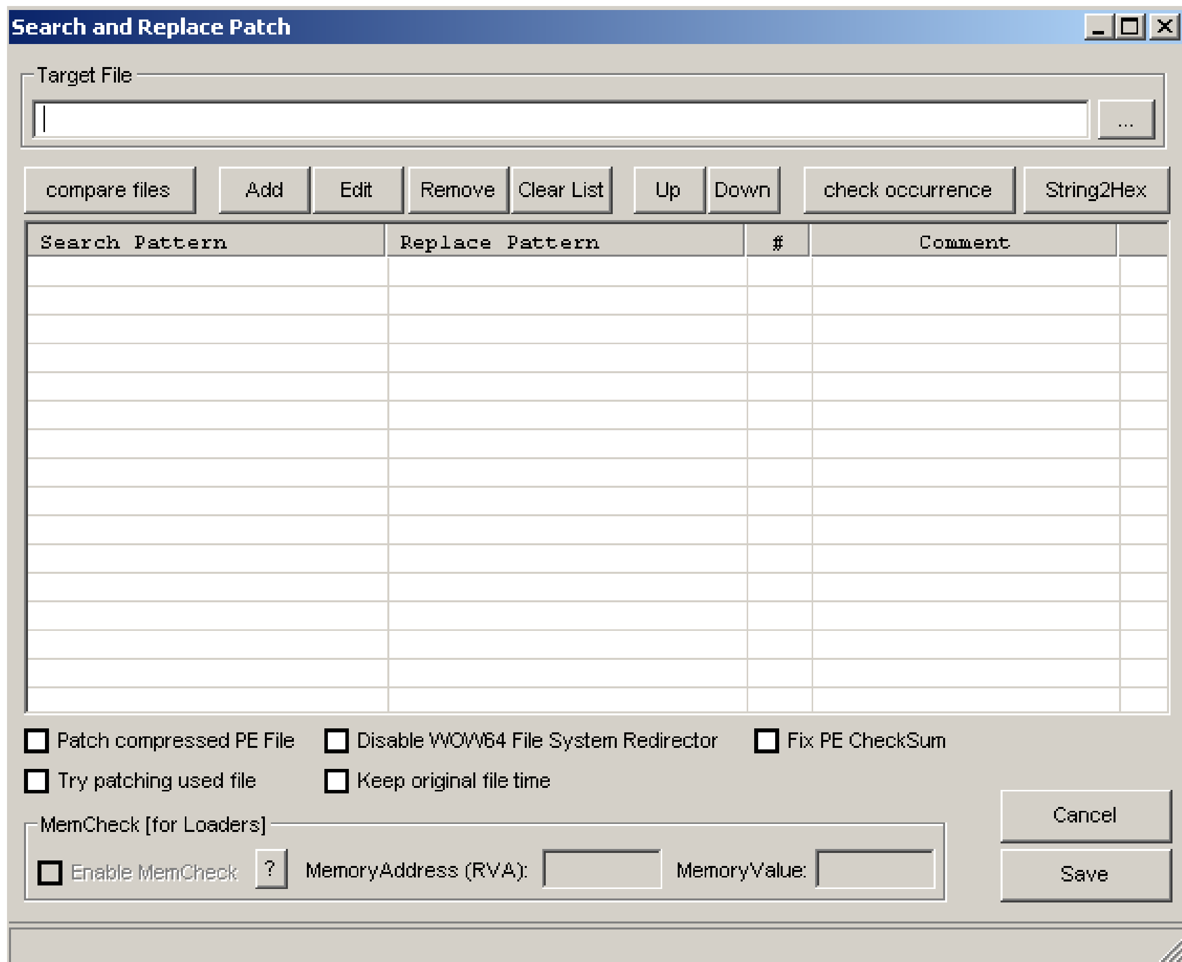
Nota: *Para que esto funcione, el generador de parches debe estar en la misma carpeta que la aplicación a la que deseamos aplicar el parche.*

Si en lugar de añadir un 'Offset Patch' quisieramos utilizar la opción de 'Search and Replace Patch' lo seleccionariamos en la ventana de dUP2:

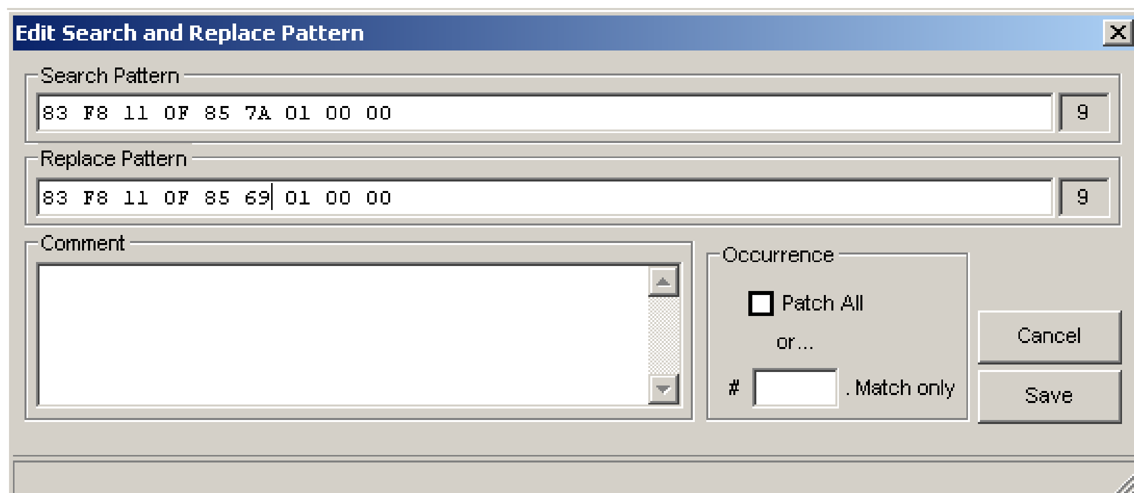


Haciendo doble clic en la nueva línea nos llevaría a la ventana de ‘Search y Replace Patch’.

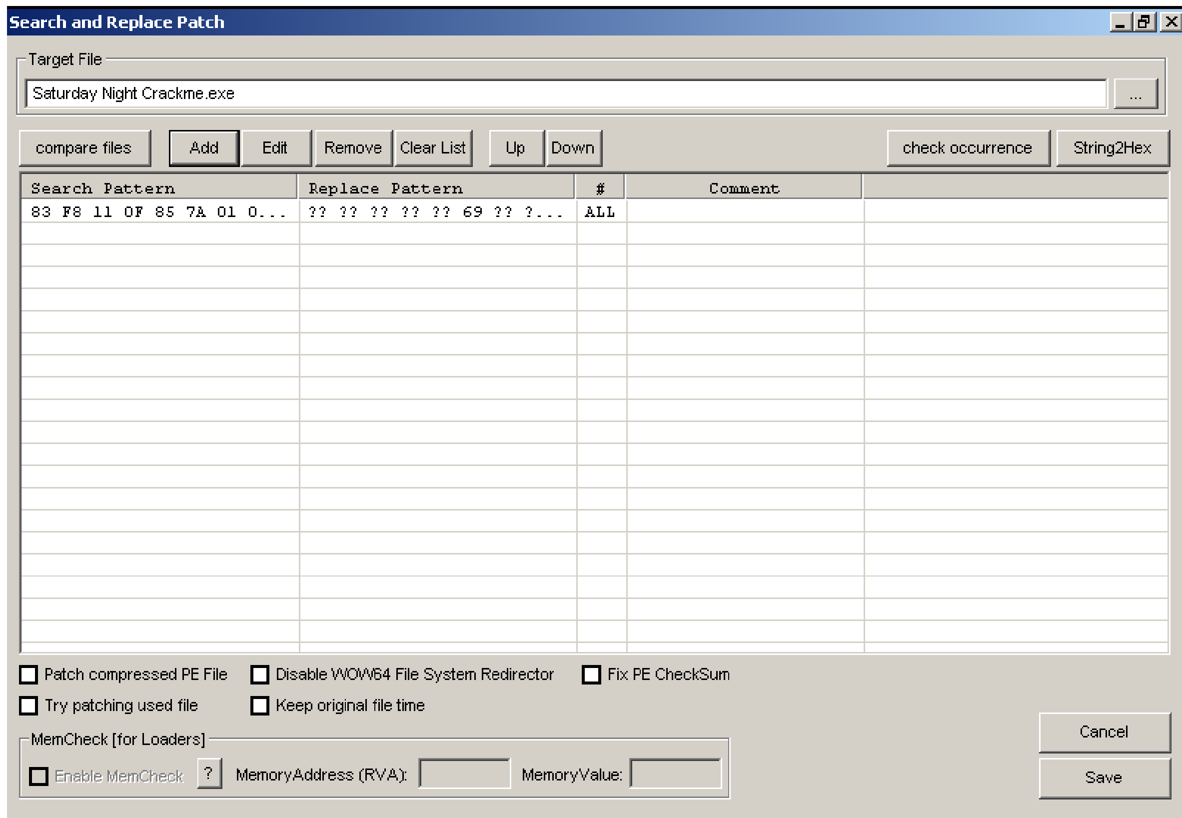
Seleccionamos la aplicación y pulsamos sobre el botón ‘Add’:



Introduciremos la misma información que introducimos en CFF para encontrar los bytes hexadecimales, cambiando el 7A por el 69:



Guardamos:



Hacemos clic en 'Save'y creamos el parche como anteriormente. Si ejecutamos la aplicación crackeada vemos que obtenemos el mismo resultado.

Nota: *En la pestaña 'Settings' podemos configurar la ventana del generador de parches.*

7.19 Caso práctico 19: Trabajando con binarios de Visual Basic (I)

En este ejercicio vamos a trabajar con aplicaciones escritas en Visual Basic. Utilizaremos dos crackme's así como un decompilador para Visual Basic (edición Lite).

Introducción a Visual Basic

Visual Basic es un lenguaje de programación dirigido por eventos, desarrollado por Alan Cooper para Microsoft. Este lenguaje de programación es un dialecto de BASIC, con importantes agregados. Su primera versión fue presentada en 1991, con la intención de simplificar la programación utilizando un ambiente de desarrollo que facilitó en cierta medida la programación misma.

La última versión fue la 6, liberada en 1998, para la que Microsoft extendió el soporte hasta marzo de 2008.

En 2001 Microsoft propuso abandonar el desarrollo basado en la API Win32 y pasar a un framework o marco común de librerías, independiente de la versión del sistema operativo, .NET Framework, a través de Visual Basic .NET (y otros lenguajes como C Sharp (C#) de fácil transición de código entre ellos); fue el sucesor de Visual Basic 6.

Aunque Visual Basic es de propósito general, también provee facilidades para el desarrollo de aplicaciones de bases de datos usando Data Access Objects, Remote Data Objects o ActiveX Data Objects.

Visual Basic contiene un entorno de desarrollo integrado o IDE que integra editor de textos para edición del código fuente, un depurador, un compilador (y enlazador) y un editor de interfaces gráficas o GUI.

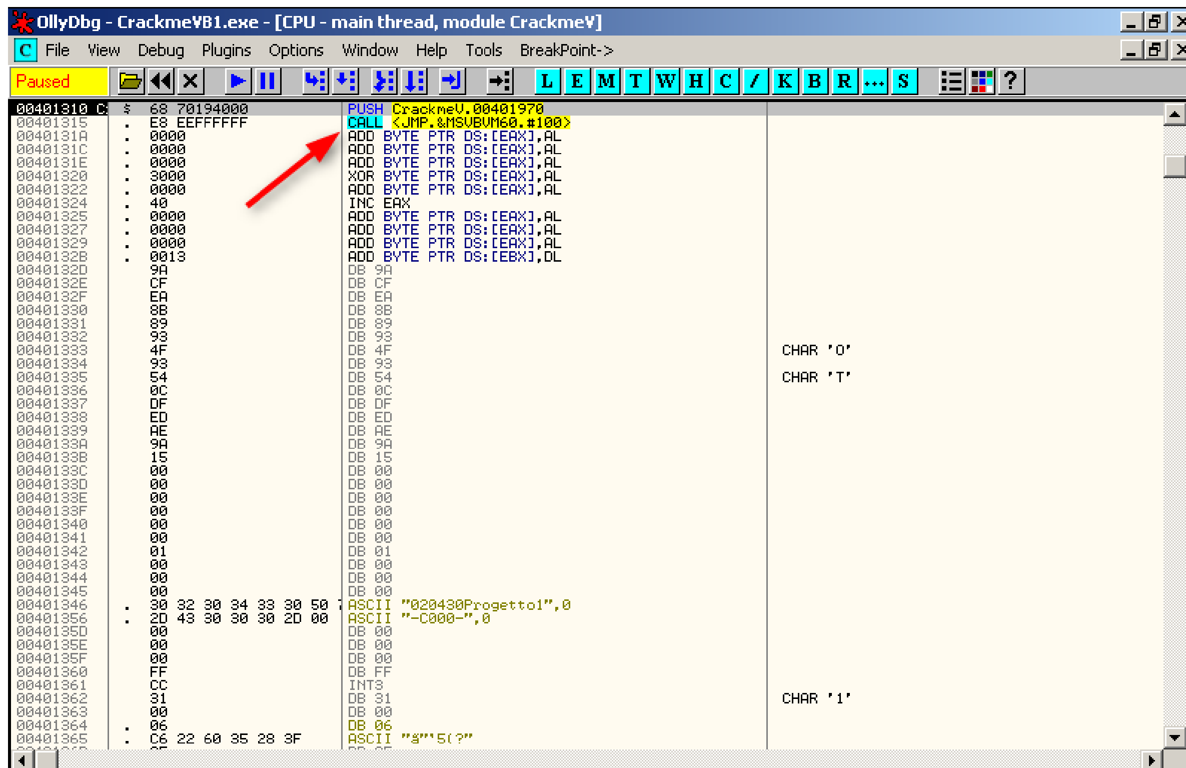
Los compiladores de Visual Basic generan código que requiere una o más librerías de enlace dinámico para que funcione, conocidas comúnmente como DLL (sigla en inglés de dynamic-link library); en algunos casos reside en el archivo llamado MSVBVMxy.DLL (siglas de "MicroSoft Visual Basic Virtual Machine x.y", donde x.y es la versión) y en otros como VBRUNXXX.DLL ("Visual Basic Runtime X.XX"). Estas bibliotecas DLL proveen las funciones básicas implementadas en el lenguaje, incluyendo las rutinas de código ejecutable que son cargadas bajo demanda en tiempo de ejecución. Además de las esenciales, existe un gran número de bibliotecas del tipo DLL con variedad de funciones, tales como las que facilitan el acceso a la mayoría de las funciones del sistema operativo o las que proveen medios para la integración con otras aplicaciones.

Dentro del mismo entorno de desarrollo integrado (IDE) de Visual Basic se puede ejecutar el programa a desarrollar, es decir en modo intérprete (en realidad pseudo-compila el programa muy rápidamente y luego lo ejecuta, simulando la función de un intérprete puro). Desde ese entorno también se puede generar el archivo en código ejecutable (exe); ese programa así generado en disco puede luego ser ejecutado sin requerir del ambiente de programación (incluso en modo stand alone), aunque sí será necesario que las librerías DLL requeridas por la aplicación desarrollada se encuentren también instaladas en el sistema para posibilitar su ejecución.

El propio Visual Basic provee soporte para empaquetado y distribución; es decir, permite generar un módulo instalador que contiene al programa ejecutable y las bibliotecas DLL necesarias para su ejecución. Con ese módulo la aplicación desarrollada se distribuye y puede ser instalada en cualquier equipo (que tenga un sistema operativo compatible).

Existen numerosas aplicaciones desarrolladas por terceros que permiten disponer de variadas y múltiples funciones, incluso mejoras para el propio Visual Basic; las hay también para el empaquetado y distribución, y hasta para otorgar mayor funcionalidad al entorno de programación (IDE).

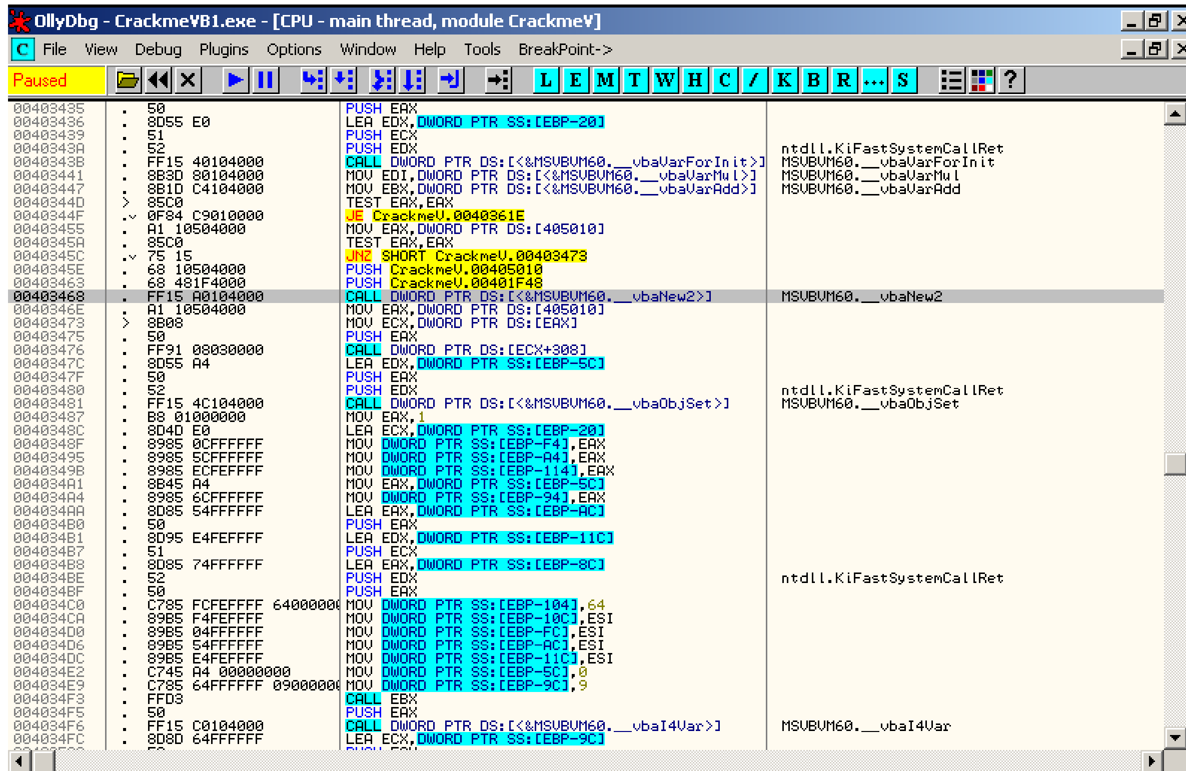
Cuando cargamos un programa de Visual Basic por primera vez en un depurador como Ollly, veremos que se ejecuta inmediatamente una instrucción CALL dentro de la DLL de Visual Basic, donde permanecerá hasta que ocurra un evento. Este es el motivo por el cual los programas de Visual Basic se les aplica una ingeniería inversa algo distinta. Lo primero que veremos es que el call stack no tiene cabida; esto es así porque la mayoría de los tiempos de ejecución de un programa se realizan dentro de un archivo DLL; el tiempo de ejecución DLL de Visual Basic. No le daremos mucha importancia a este DLL, pero sí a los métodos de devolución de llamadas que son responsables de manejar los eventos.



Otra diferencia está en la manera de manejar las cadenas de texto. La mayoría de los cuadros de mensajes, así como todos los demás controles de las ventanas son almacenadas en las secciones de recursos. De ahí que buscar cadenas de textos para encontrar secciones relevantes no es una opción válida.

Otro obstáculo lo encontramos en los métodos CALL, que son completamente diferentes en un ejecutable de Visual Basic. En lugar de hacer llamadas a RegisterWindowEx y MessageBoxA, Visual Basic utiliza su propia API para hacer las llamadas, estando embebidas en el tiempo de ejecución de los DLL:

Si hacemos doble clic en cualquier de estos métodos veremos que no encontraremos cadenas que nos puedan ayudar, ni API's a llamadas conocidas.



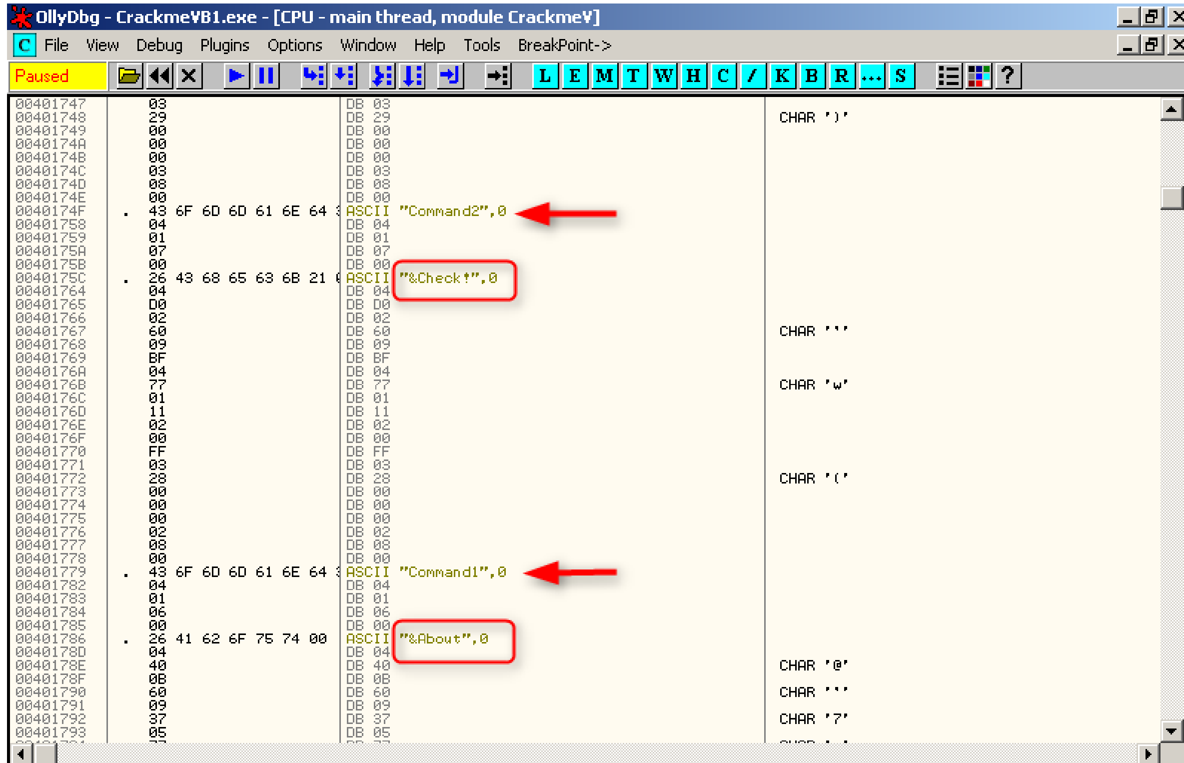
Antes de listar las herramientas que tenemos a nuestra disposición, veremos la estructura básica de un ejecutable de Visual Basic. Una vez cargado CrackmeVB1.exe, que está compilado en código P, vemos la lista de funciones en el binario (para ello debemos subir hasta la primer línea en Olly).

Bajando un poco nos encontramos con la tabla de saltos. Esta es similar a la mayoría de los binarios de Windows, y está aquí para ayudar a reubicar el código:

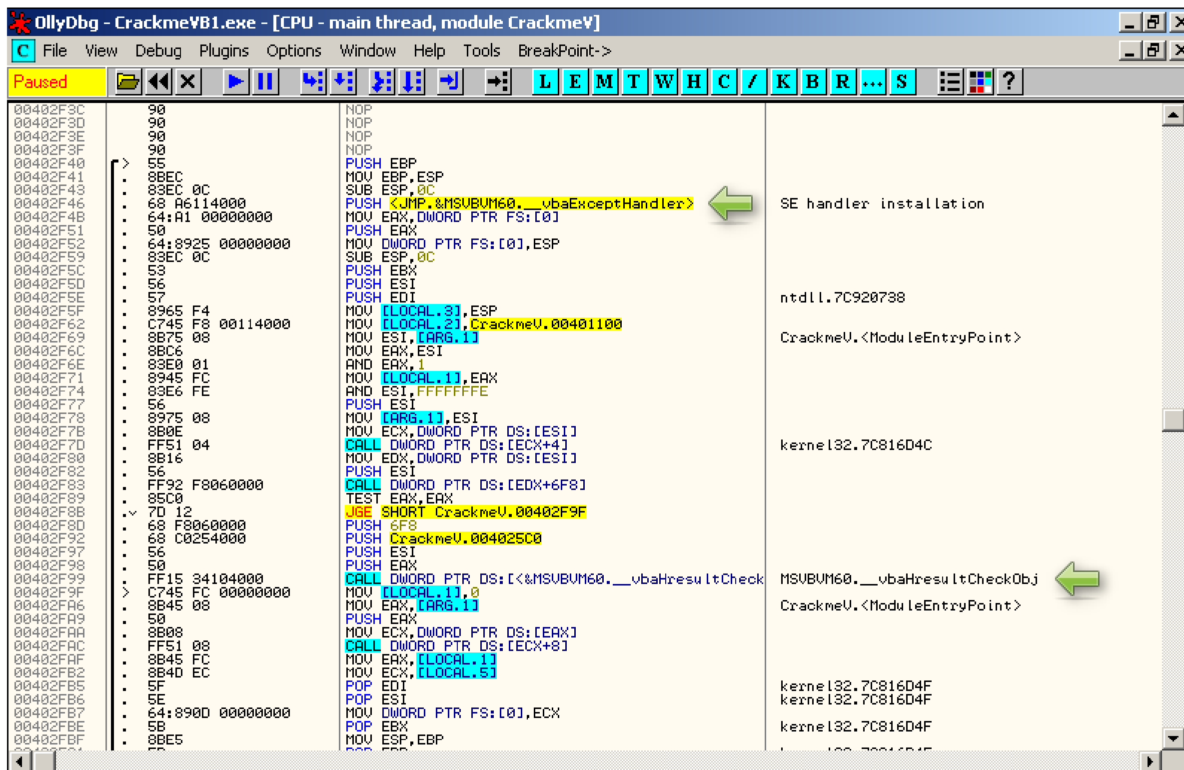
A continuación llegamos a la sección donde los binarios de Visual Basic almacenan sus recursos. Una cosa a tener en cuenta es que Visual Basic utiliza el nombre real de la devolución de llamada; si queremos que “MyButtonCallback” maneje el evento botón, se utilizará esa

cadena de texto para hacer referencia a ello. Este es el motivo por el cual veremos algunos nombres de devolución de llamadas embebidos en la sección de los recursos.

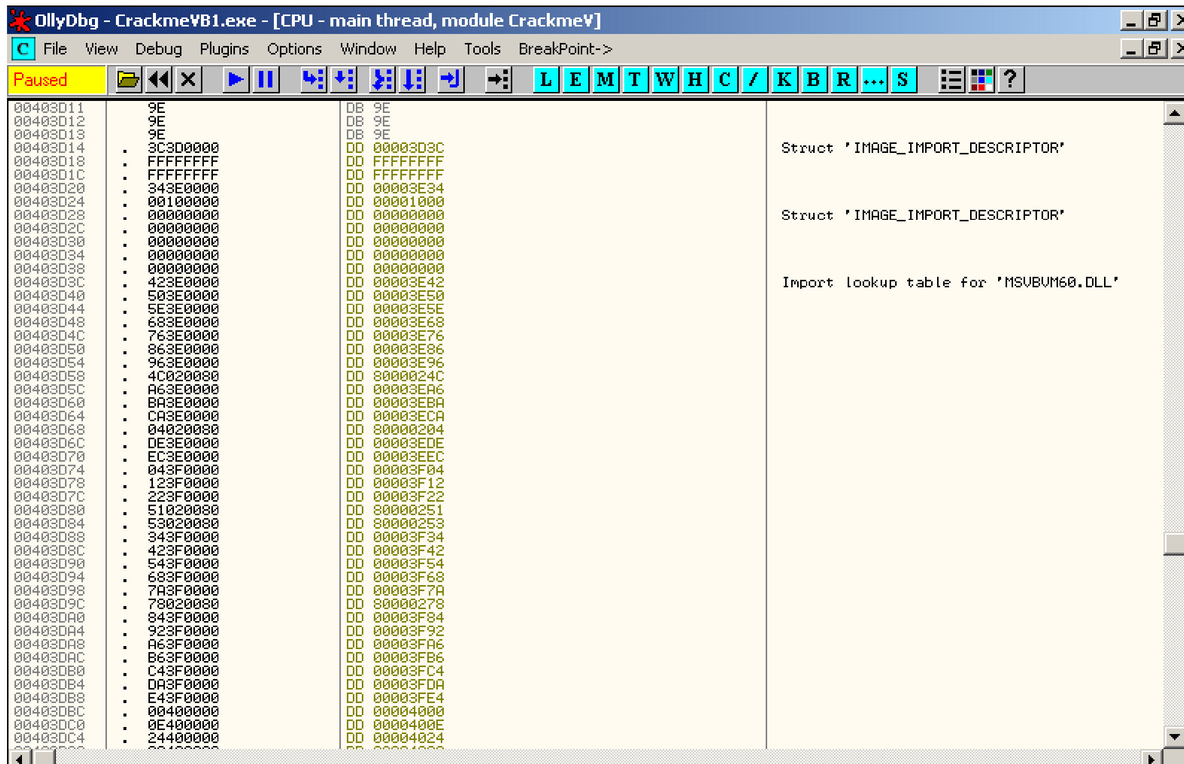
A continuación vemos un ejemplo de los botones 'Check' y 'About' con sus respectivo devoluciones de llamadas.



Bajando un poco más llegaremos a la devolución de llamadas de los eventos. Los métodos de devolución de llamadas son generados por los usuarios para manejar varios eventos.

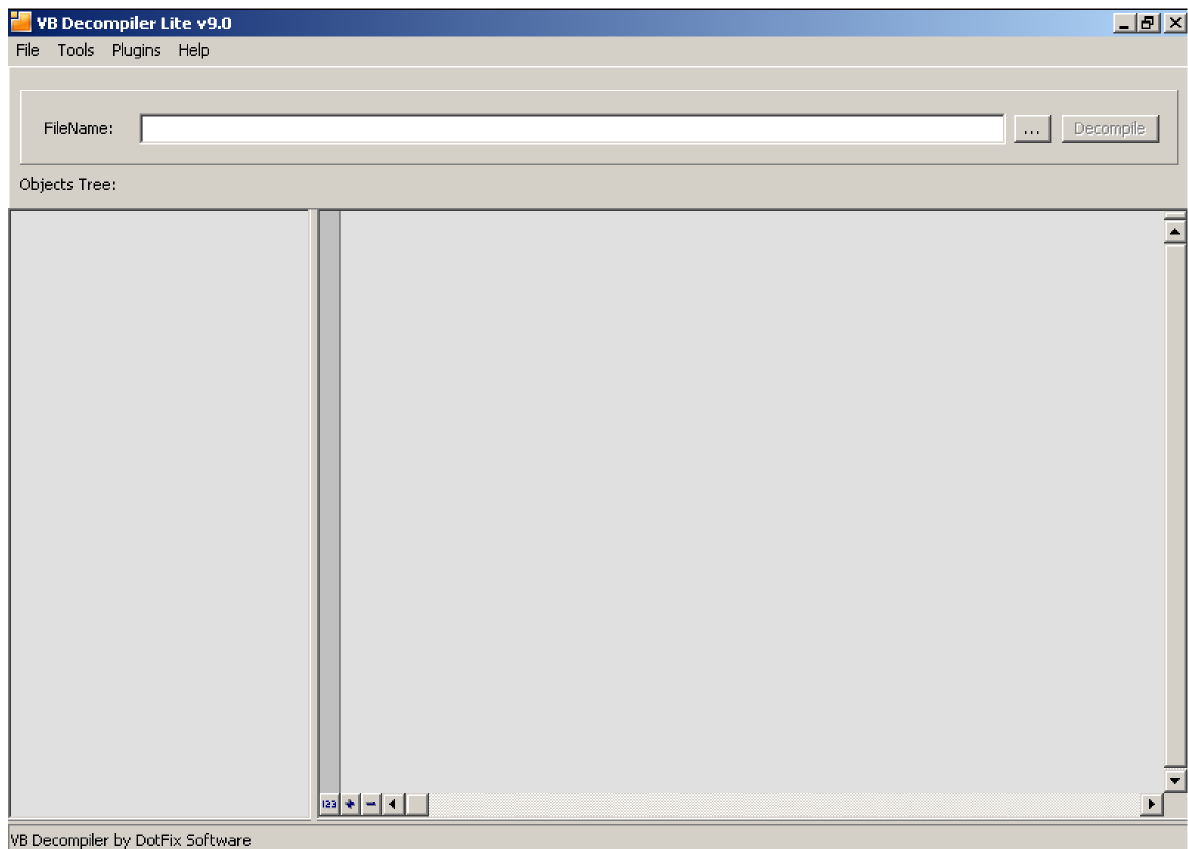


Finalmente llegaremos a las tablas de importación de direcciones o IAT (Import Address Table).

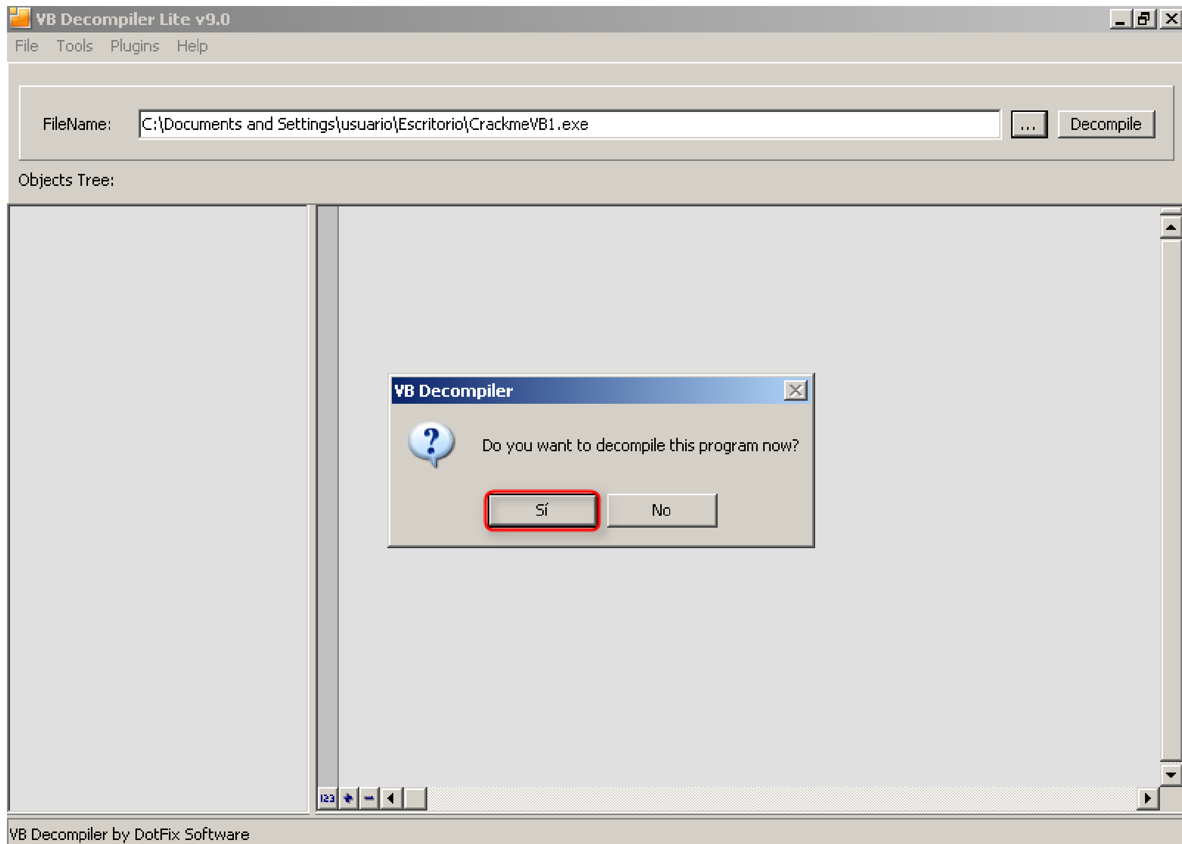


A la vista de la escasa información obtenida por Olly, nos ayudaremos de la herramienta VB Decompiler Lite. Dicha herramiento nos ayudará a decompilar el código de Visual Basic, es decir nos ayudará a convertir el código P de vuelta a su estado original, el código fuente de Visual Basic. Asimismo nos ayudará a ver los recursos embebidos en los ejecutables de una

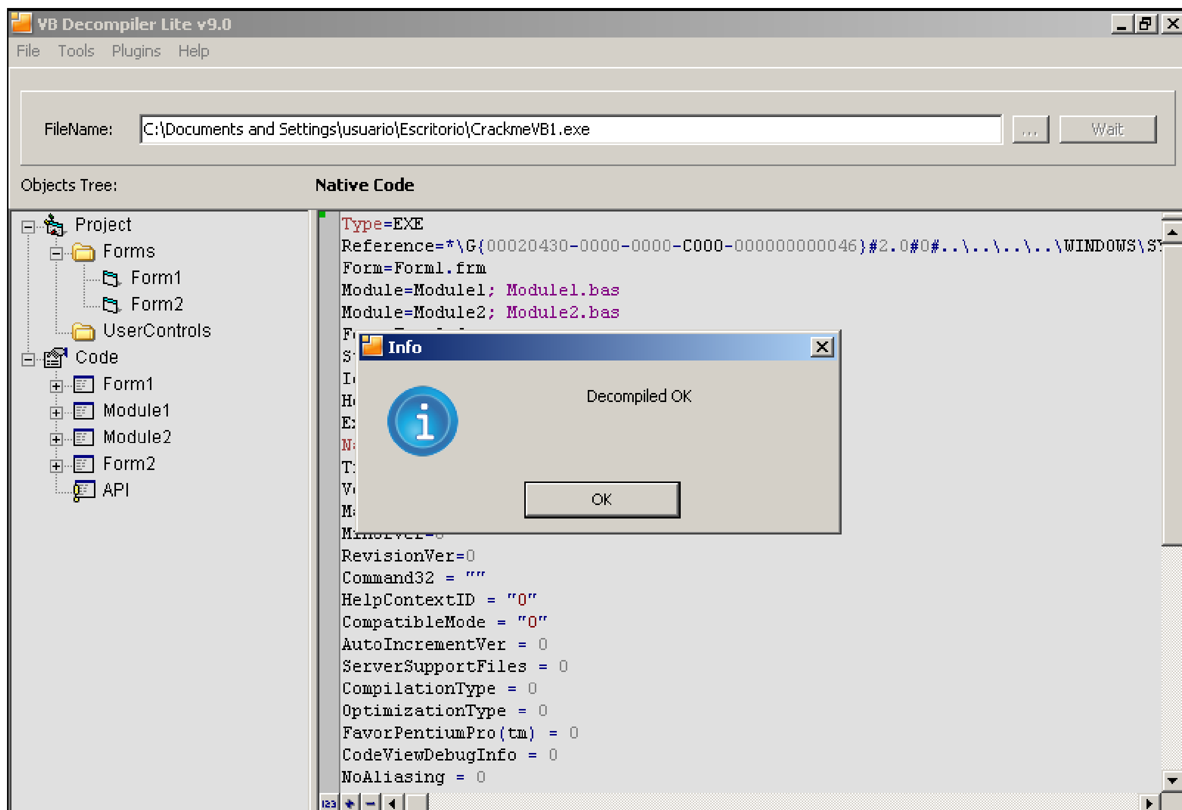
manera más amigable. Ejecutamos pues VB Decompiler Lite y lo primero que aparece es la siguiente ventana principal:



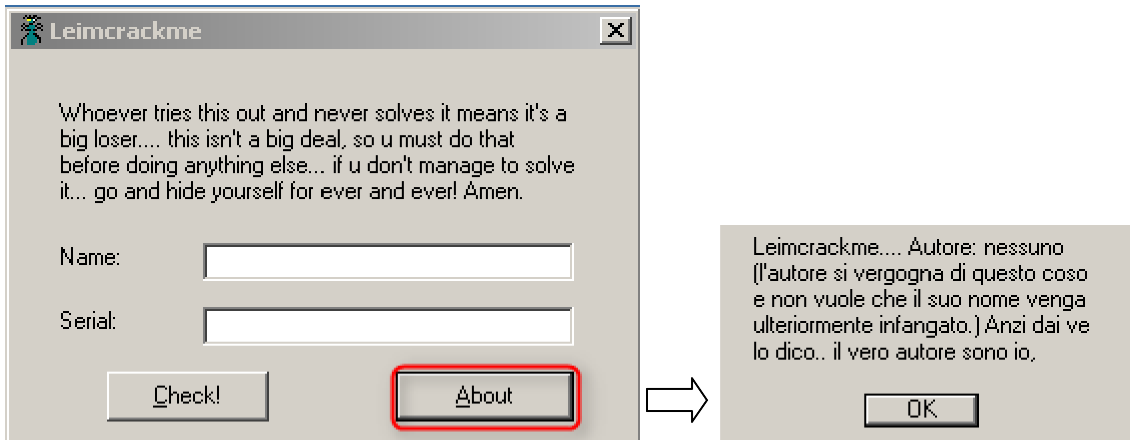
Seleccionamos nuestro crackme y le decimos que queremos decompilarlo ahora:



Si todo ha ido bien el decompilador nos muestra el siguiente mensaje:

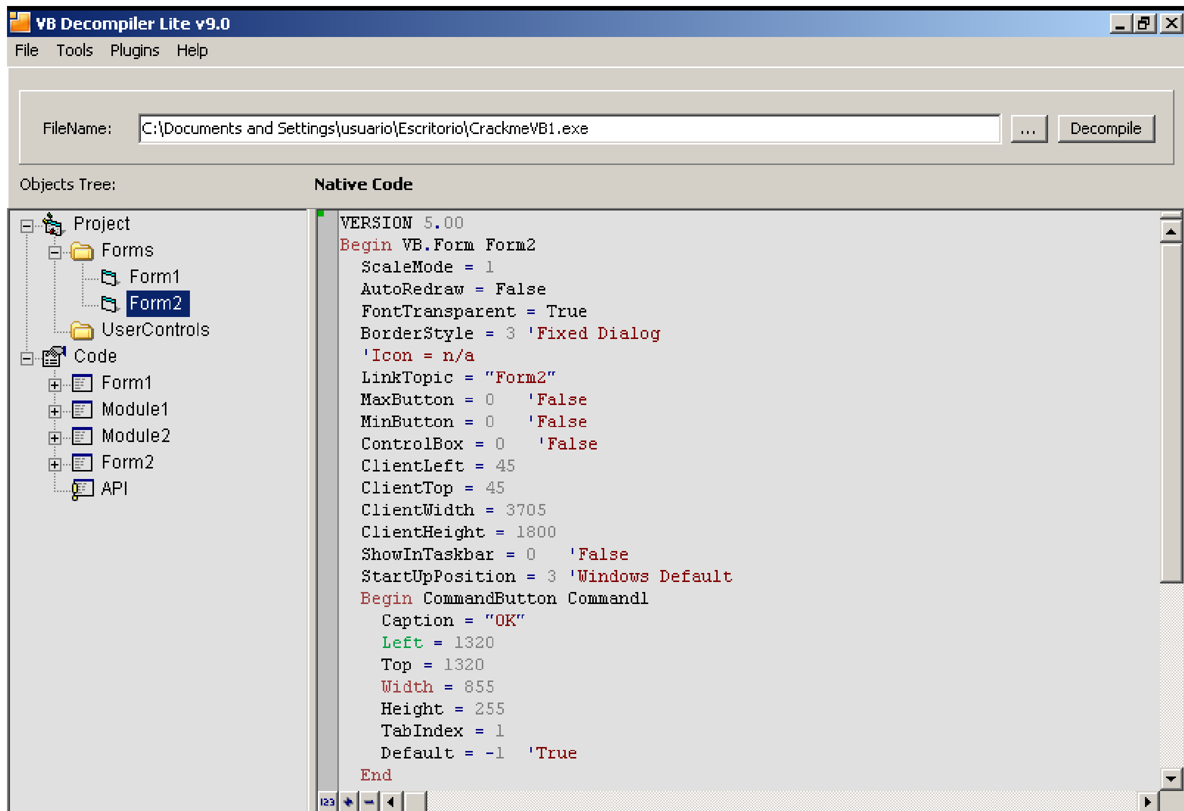


La mayoría de la información de este crackme es irrelevante. Lo importante a tener en cuenta es que podemos observar que existen dos formularios dentro de la carpeta de Forms; Form1 y Form2. Se trata de los recursos asociados a cada formulario. Podemos suponer que la aplicación tiene dos ventanas; una será la ventana principal y la otra corresponderá a la ventana 'About'. Verifiquémoslo haciendo doble clic sobre la aplicación:



Cabe mencionar dos cosas: 1) vemos que la ventana 'About' está en un lenguaje distinto y 2) no podemos hacer clic en el botón de 'OK' dentro de la ventana 'About'.

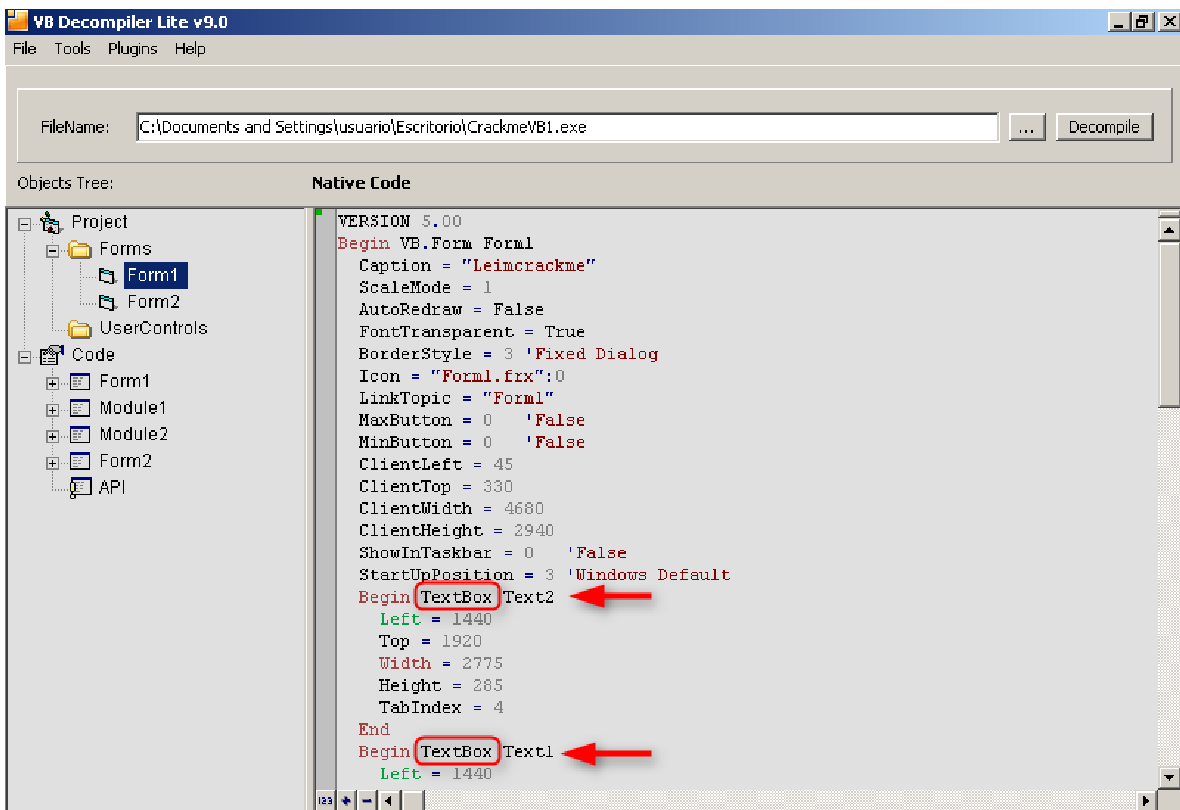
Si hacemos doble clic en "Form2" dentro de la carpeta Forms veremos los distintos recursos con sus atributos de Form2:



```
API
  Begin Label Label1
    Caption = "Form2.frx":0
    Left = 480
    Top = 120
    Width = 2535
    Height = 975
    TabIndex = 0
  End
End

Attribute VB_Name = "Form2"
```

Podemos ver un botón con el texto ‘OK’, una etiqueta con el texto en otra lengua y un método de devolución de llamada para el evento del botón ‘OK’; ‘Command1’.

Haciendo doble clic en Form1 veremos los atributos de la ventana principal:



```
Project
  Forms
    Form1
    Form2
  UserControls
Code
  Form1
  Module1
  Module2
  Form2
  API
```

```
   Width = 2775
   Height = 285
   TabIndex = 3
End
Begin CommandButton Command2
   Caption = "&Check!"
   Left = 720
   Top = 2400
   Width = 1215
   Height = 375
   TabIndex = 2
End
Begin CommandButton Command1
   Caption = "&About"
   Left = 2880
   Top = 2400
   Width = 1335
   Height = 375
   TabIndex = 1
End
Begin Label Label2
   Caption = "Serial:"
   Index = 1
   Left = 360
   Top = 1920
   Width = 855
```

```
Project
  Forms
    Form1
    Form2
  UserControls
Code
  Form1
  Module1
  Module2
  Form2
  API
```

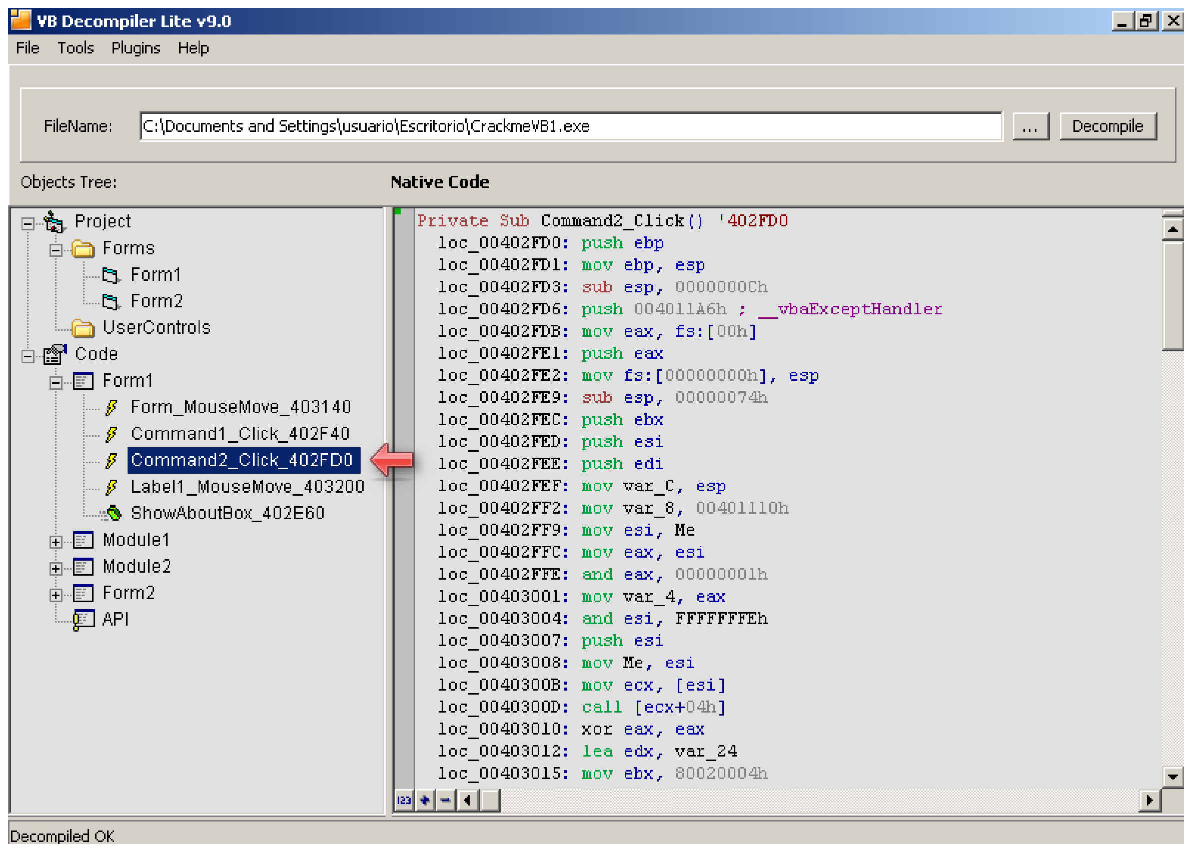
```
Begin Label Label2
   Caption = "Serial:"
   Index = 1
   Left = 360
   Top = 1920
   Width = 855
   Height = 255
   TabIndex = 6
   BackStyle = 0 'Transparent
End
Begin Label Label2
   Caption = "Name:"
   Index = 0
   Left = 360
   Top = 1440
   Width = 855
   Height = 255
   TabIndex = 5
   BackStyle = 0 'Transparent
End
Begin Label Label1
   Caption = "Form1.frx":30A
   Left = 360
   Top = 360
   Width = 3855
   Height = 975
```

Ahora ya podemos recopilar algo más de información acerca de nuestra aplicación:

- Vemos que existen dos cuadros de texto.
- Los botones 'Check' y 'About' con el nombre de sus respectivas devoluciones de llamadas.
- Finalmente aparecen más cuadros de texto.

Si nos fijamos en la parte izquierda bajo la pestaña 'Code' podemos ver el código correspondiente a los diferentes formularios. Si desplegamos la pestaña Form1 podemos ver que existen cinco devoluciones de llamadas. Uno para el botón 'Check' (Command2_Click_402FD0) y otros para los restantes botones y el movimiento del ratón.(El color del texto de la ventana principal cambia cuando se pasa el ratón).

Lo que a nosotros nos interesa es el Command2, que es nuestra devolución de llamada:

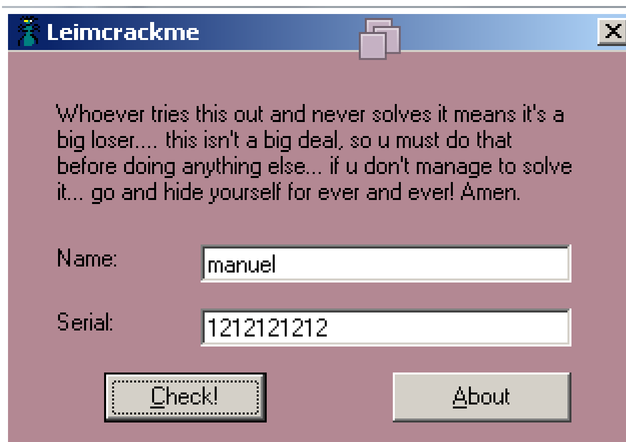


Vemos que haciendo doble clic nos muestra en la parte derecha el código ensamblado…

Lo importante de esta pantalla es la dirección de la devolución de llamada. Lo unico que necesitamos del VB Decompiler (en este caso en particular) es buscar la dirección de la devolución de llamada para el botón "Check", que como podemos observar es 402FD0. Localizando esta dirección en Olly podemos ver que nos muestra el principio de la función devolución de llamada. Si ponemos un Breakpoint, reiniciamos la aplicación e introducimos un nombre y un serial vemos que después de hacer clic en el botón "Check" nos detenemos en nuestro Breakpoint:

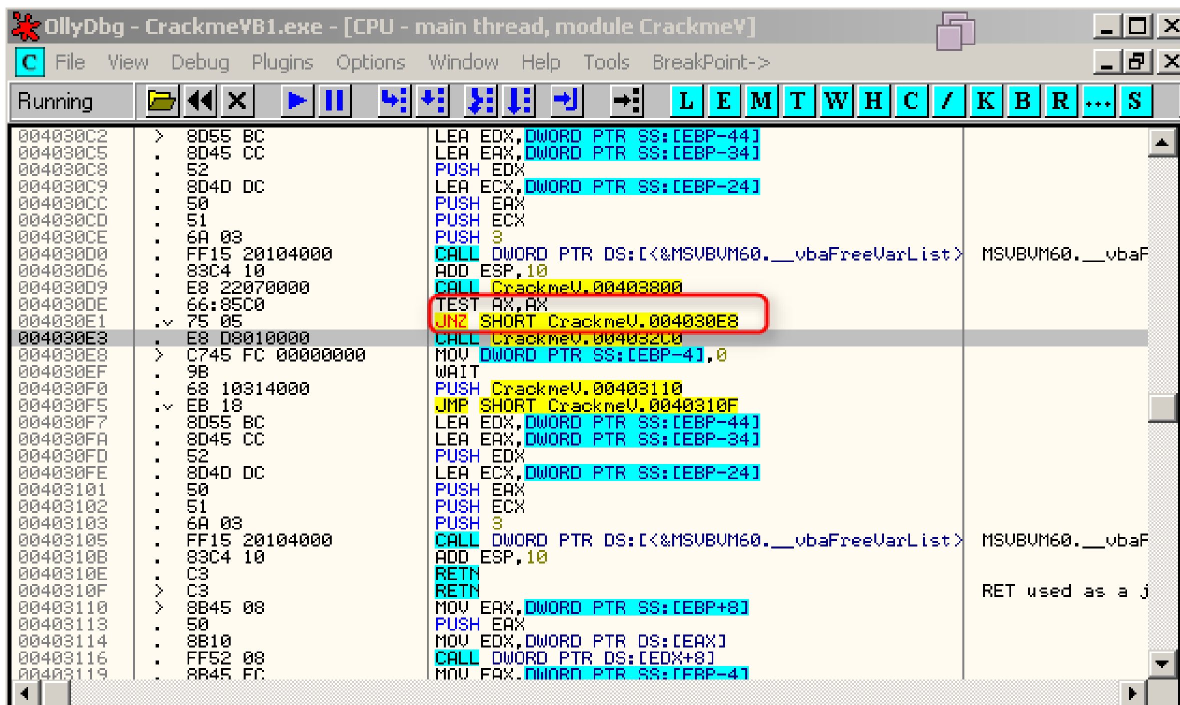
¡Hemos encontrado nuestro código de devolución de llamada del registro principal!

Pulsamos F8 hasta llegar a la dirección 4030AD. Vemos como cambia el color de fondo:

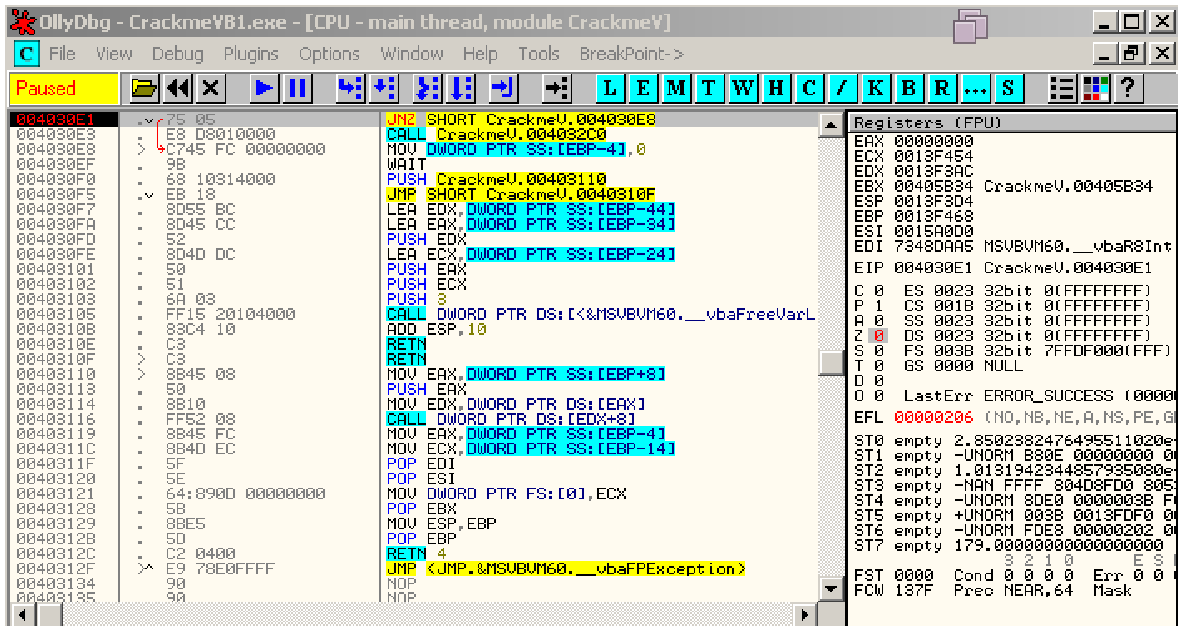


En la dirección 4030E3 vemos el "bad boy" en nuestra pantalla:

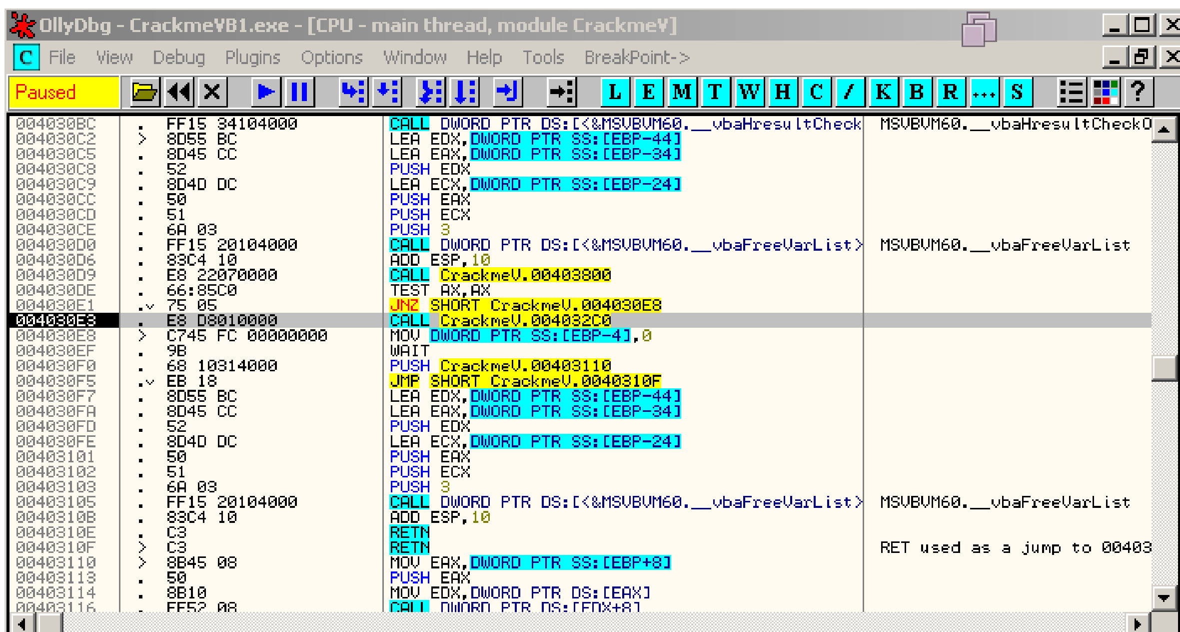
Echando un vistazo en esta área, vemos que justo antes del "bad boy" hay una combinación de comparar/saltar:



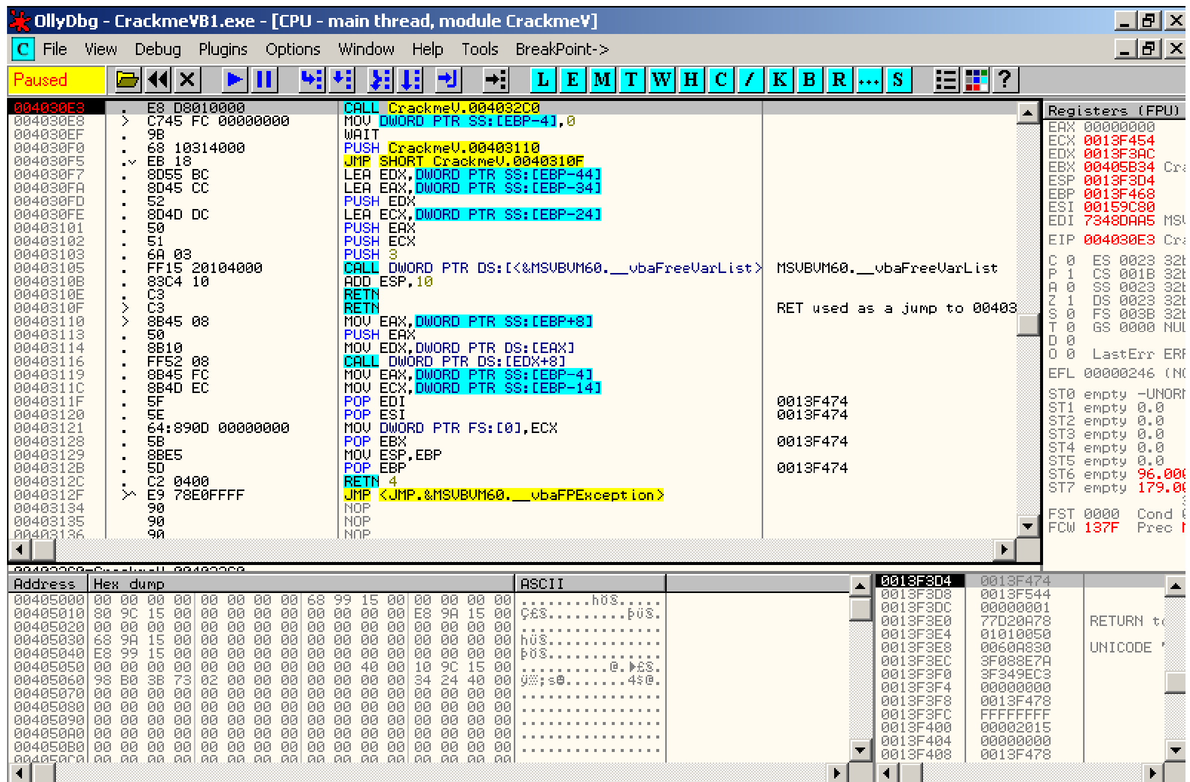
Ponemos un Breakpoint en la dirección 4030E1, reiniciamos la aplicación y miramos si esta es la comprobación que estamos buscando. Cuando Olly se detiene forzamos el salto cambiando el valor de la bandera Z:

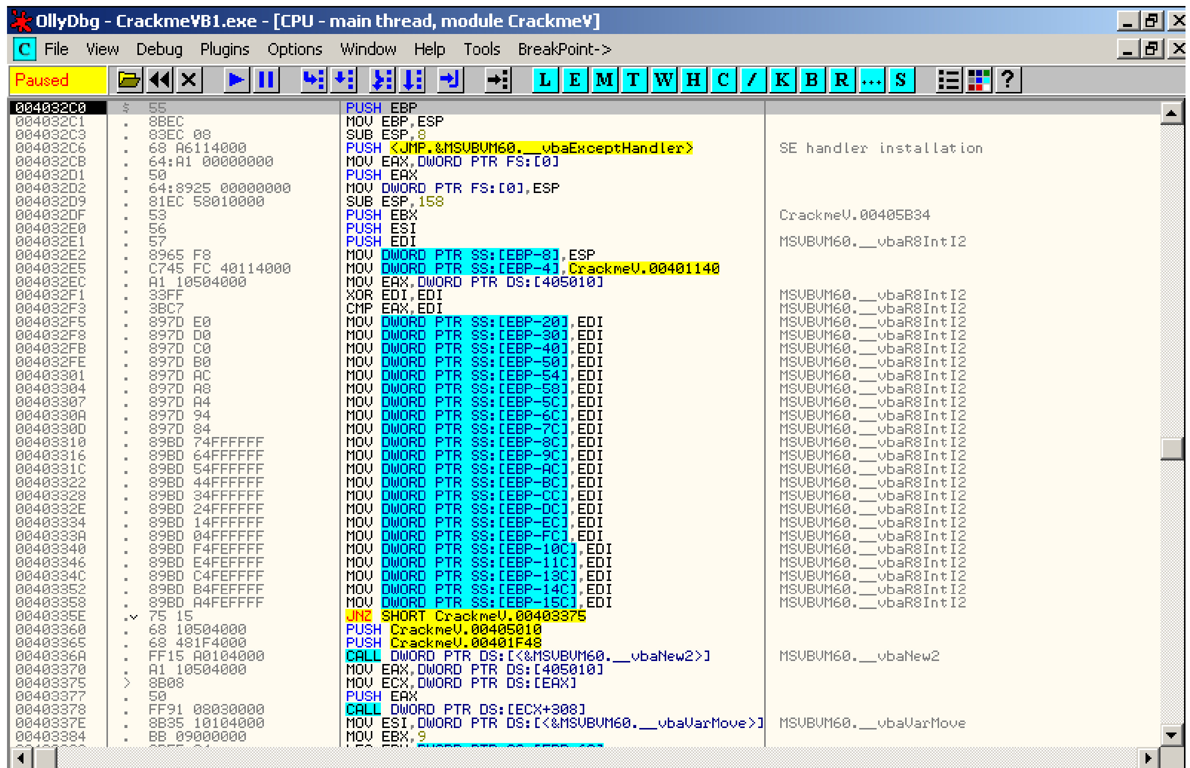


Desafortunadamente, la aplicación no nos muestra nada nuevo. Entramos pues en la instrucción CALL en la dirección 4030E3.



Ponemos un Breakpoint y pulsamos F7, lo que nos lleva a la dirección 4032C0, donde podemos ver la rutina principal de descifrado.

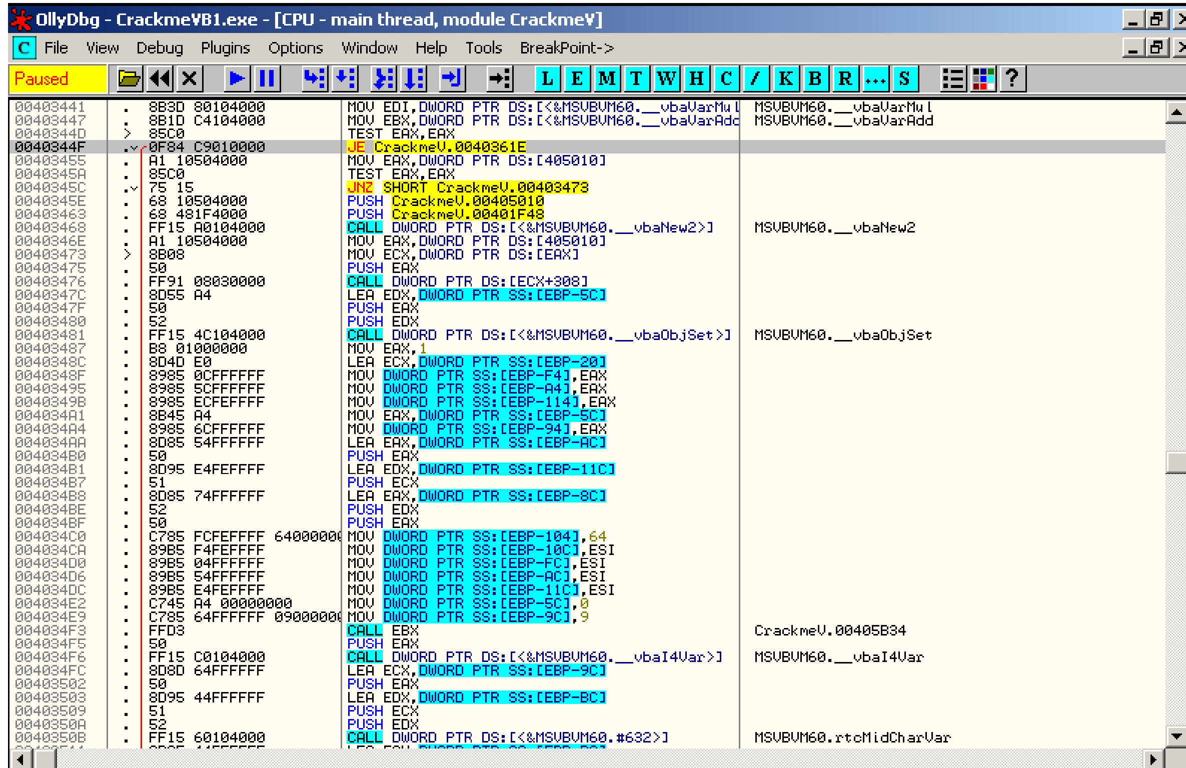


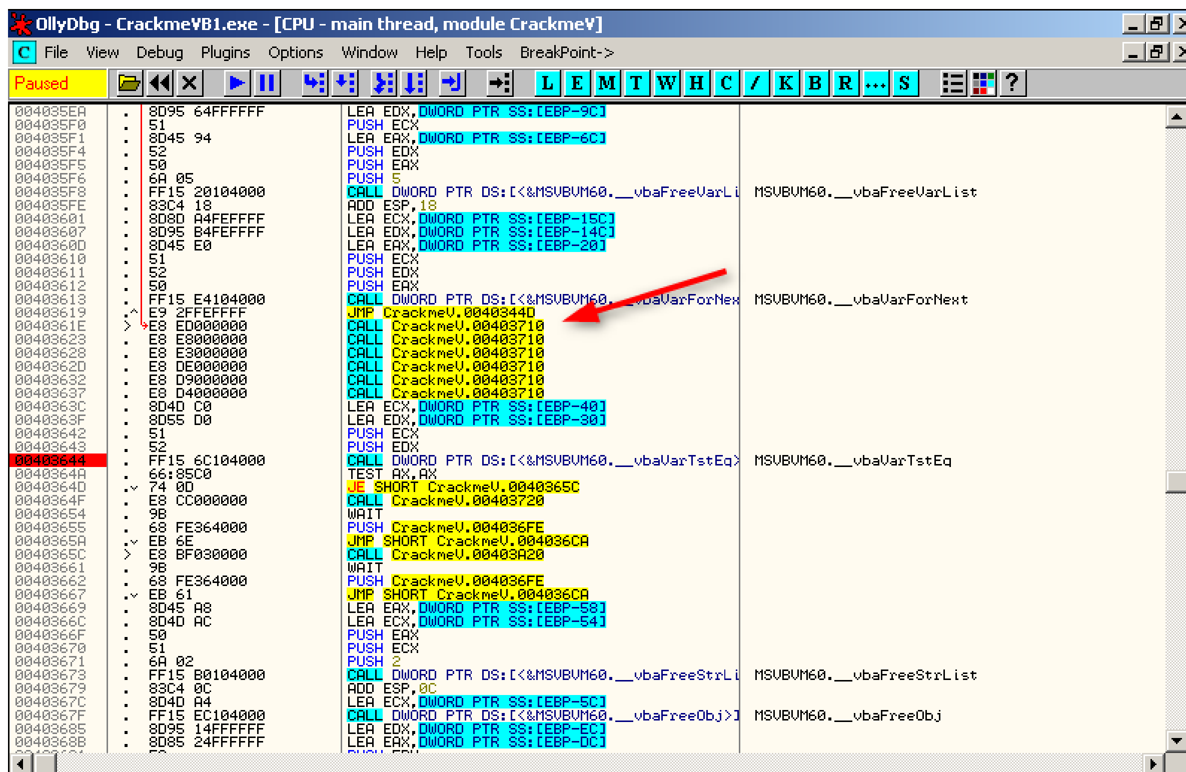


Como veremos a continuación, hay algunas llamadas a métodos muy estandarizados en Visual Basic que deberíamos memorizar. Desplazandonos un poco hacia abajo en el código podemos apreciar uno de ellos en la dirección 403644:

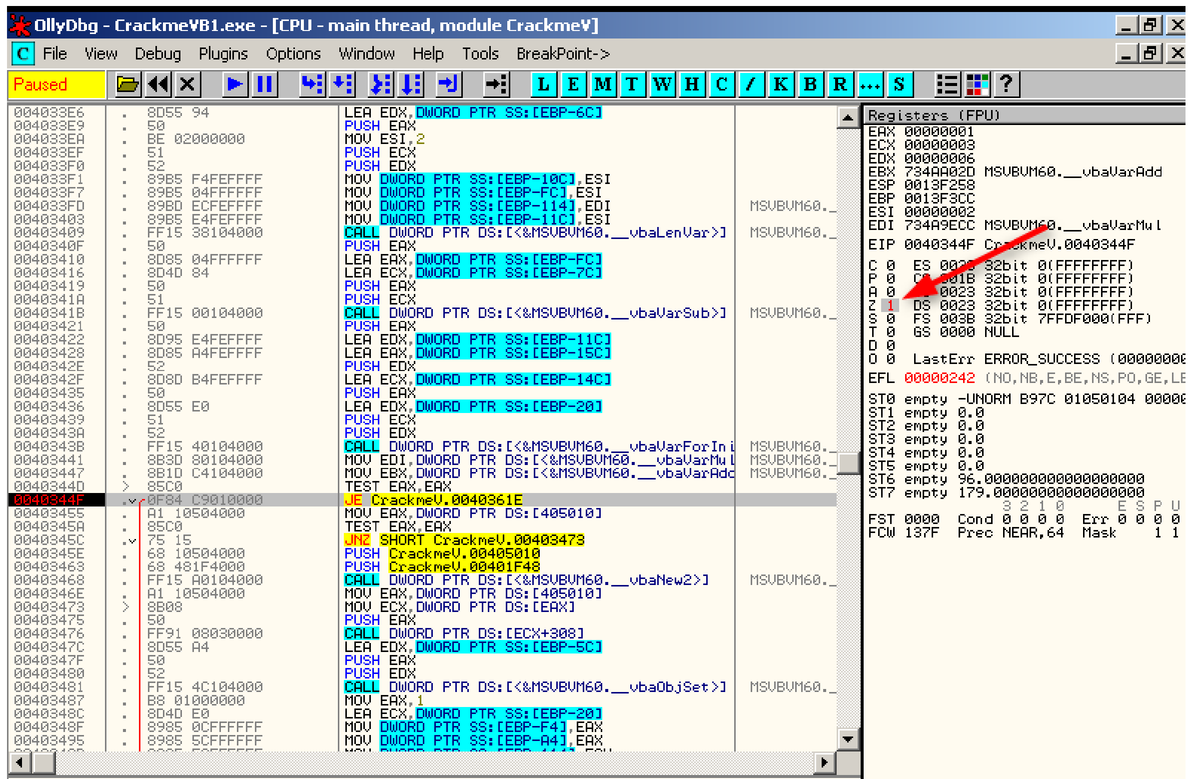
_vbaVarTstEq es como StrCmp en código nativo; comprueba dos entidades para mirar si coinciden. Seleccionamos el CALL de la dirección 40364F y pulsamos “Intro”. Vemos como Olly nos guía por el buen camino:

Sabemos que tenemos que ejecutar el código hasta la dirección 403644. Si miramos más ariba en el código aparecen algunos saltos. Seleccionamos el salto JE de la dirección 40344F y vemos que saltamos al area que nos insteresa:

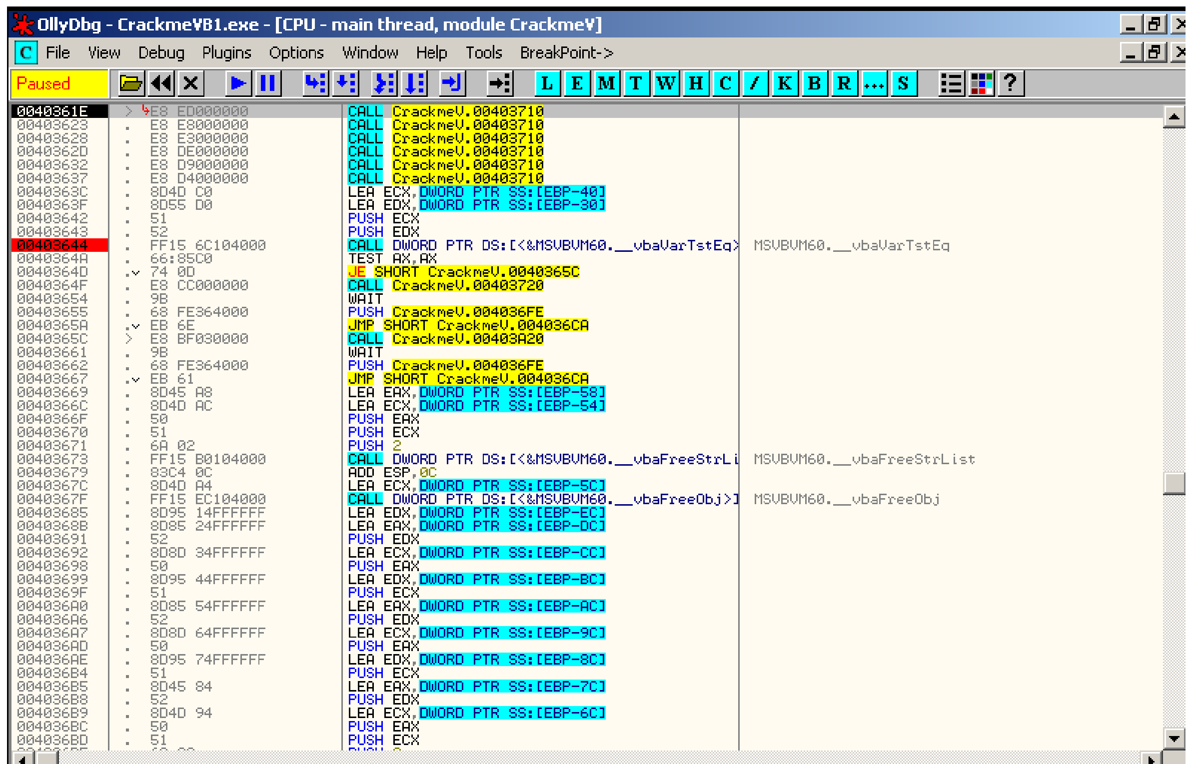




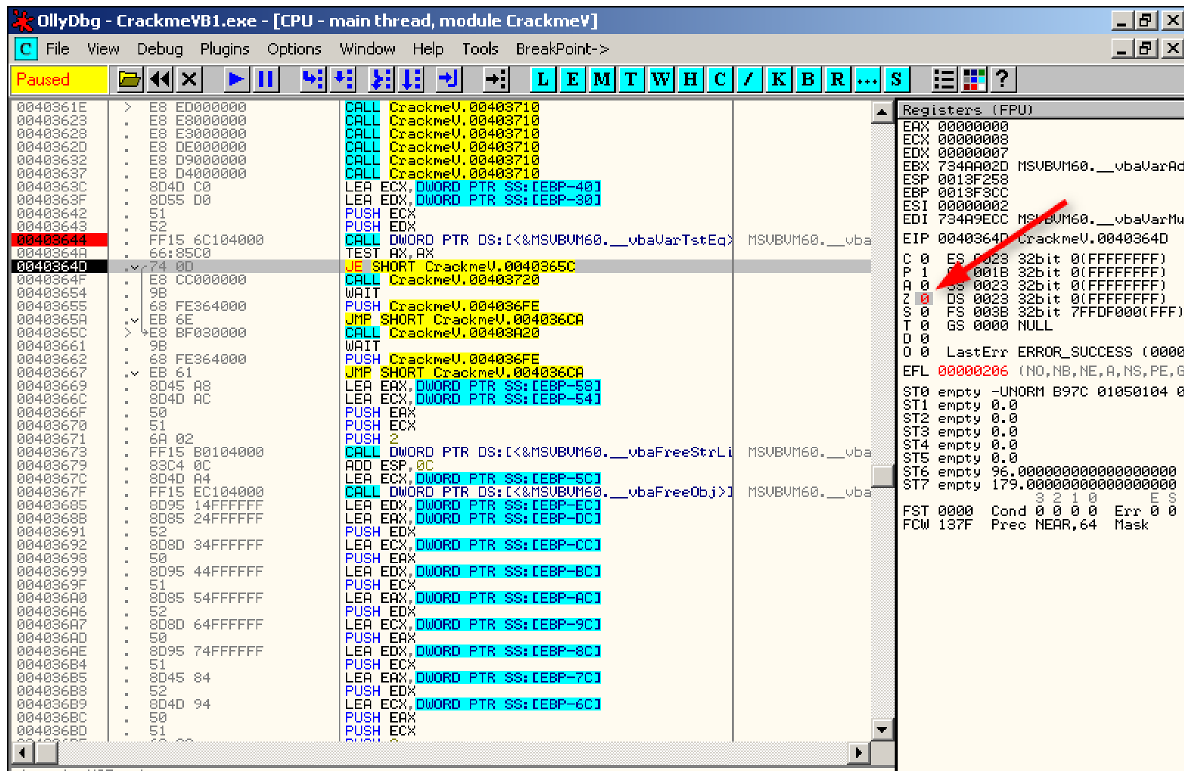
Pongamos pues un Breakpoint en la dirección 40344F, pulsamos F9 y cambiamos el valor de la bandera Z para forzar el salto:



Tomamos el salto para situarnos en la dirección 40361E.



Pulsamos F8 hasta llegar a la instrucción JE en la dirección 40364D y cambiamos el valor de la bandera Z. Esta vez para no saltar:



Pulsamos F8 una vez más para llegar al CALL de la dirección 40364F.

Como se comentó anteriormente, existen algunos métodos que son llamados con cierta frecuencia a la hora de buscar esquemas de protección:

_vbaVarTstEq

_vbaVarTstNe

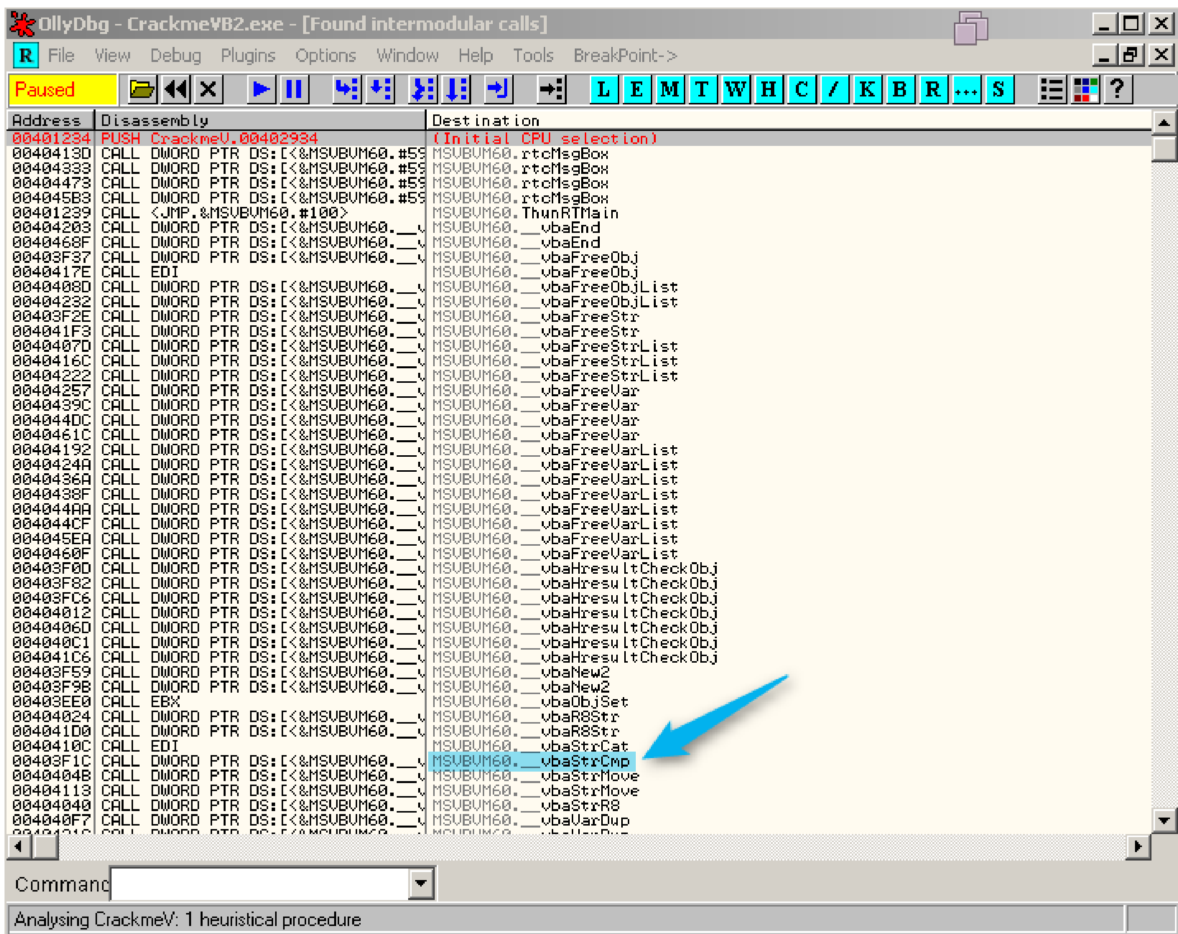
_vbaVarCmpEq

_vbaStrCmp

_vbaStrCom

_vbaStrCompVar

Nueve de cada diez veces, una de estas rutinas se utilizará para comparar un serial. Vamos a cargar CrackmeVB2.exe en Olly para realizar una busqueda de los "Intermodular calls":



Podemos apreciar un CALL a _vbaStrCmp. Lo que hace es coger dos cadenas como argumentos y devuelve un integro. El valor de retorno puede ser -1,0 y 1, dependiendo si el primer argumento es mayor, menor o igual que el segundo. En VB este CALL se vería de la siguiente forma:

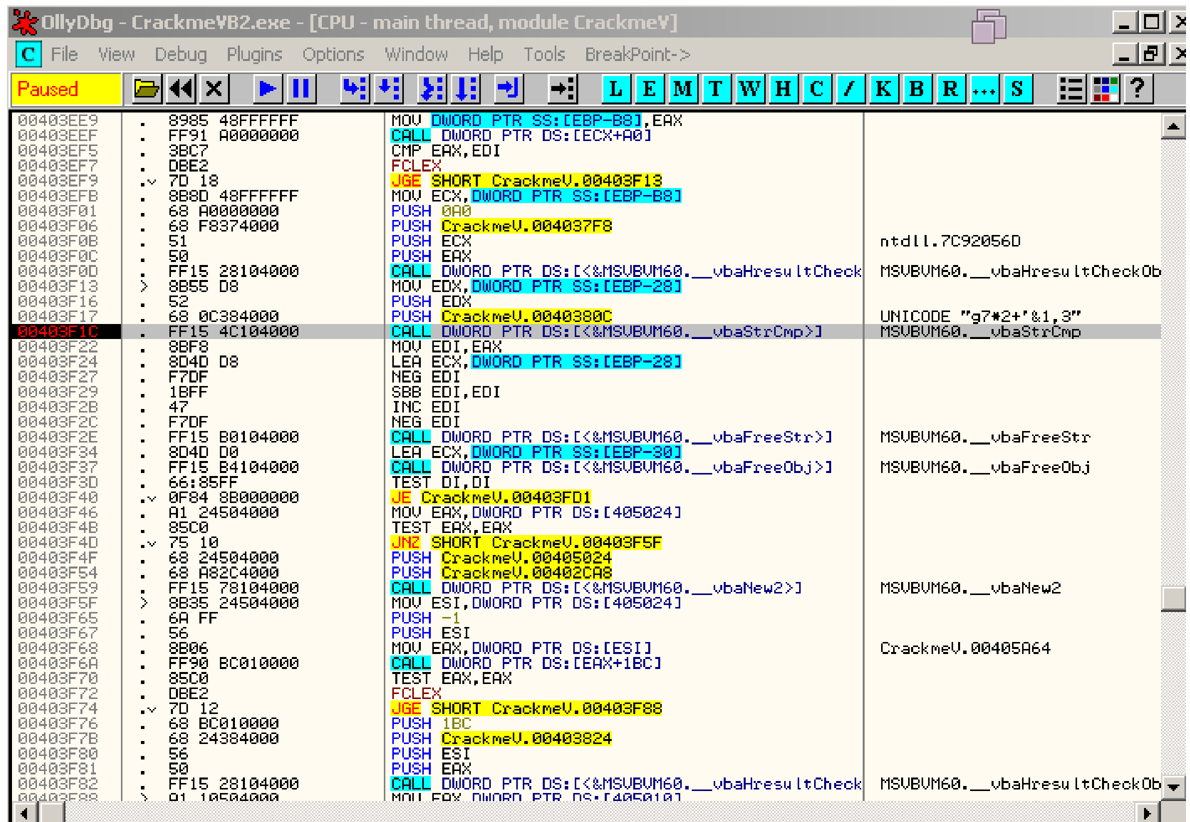
```
'Declaration
Public Shared Function Compare ( _
        strA As String, _
        strB As String _
) As Integer
'Usage
Dim strA As String
Dim strB As String
Dim returnValue As Integer

returnValue = String.Compare(strA, strB)
```

Haciendo doble clic sobre el CALL en Olly, saltaremos al lugar donde se realiza la llamada.

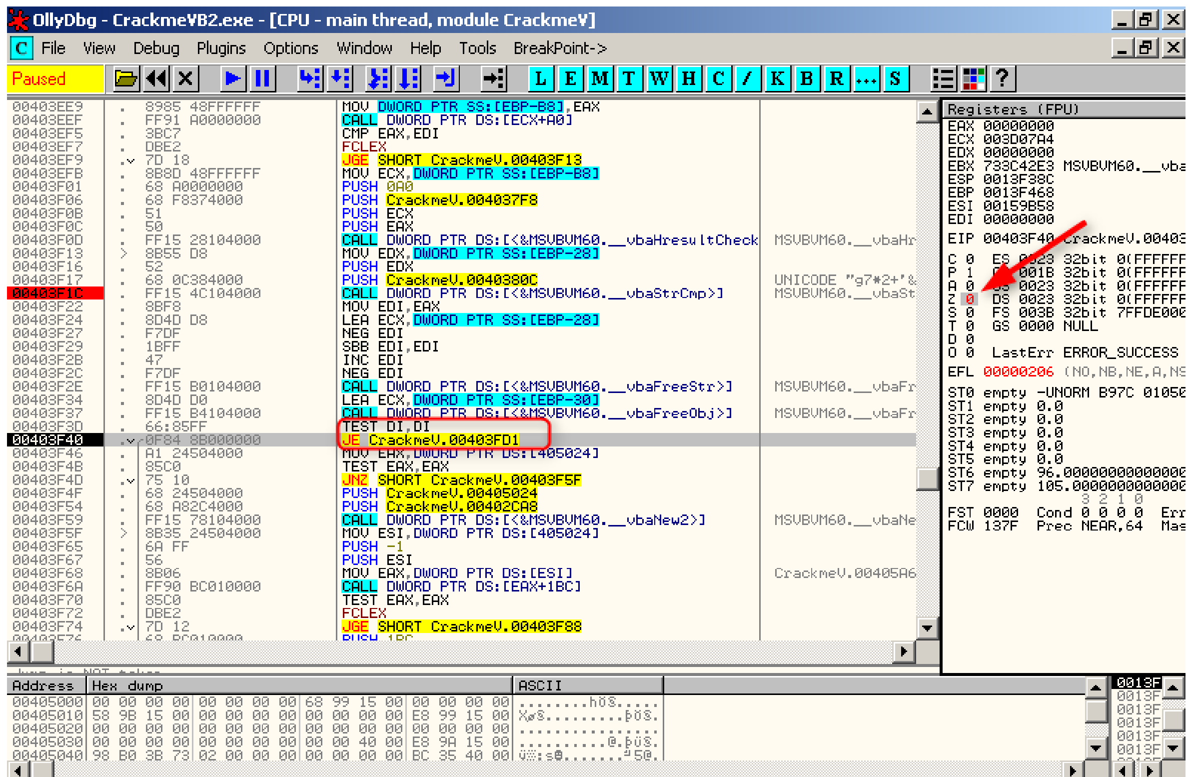
```
OllyDbg - CrackmeVB2.exe - [CPU - main thread, module CrackmeV]
C File View Debug Plugins Options Window Help Tools BreakPoint->
Paused
00403EE9  .  8985 48FFFFFF   MOV DWORD PTR SS:[EBP-B8],EAX
00403EEF  .  FF91 A0000000   CALL DWORD PTR DS:[ECX+A0]
00403EF5  .  3BC7            CMP EAX,EDI                                 ntdll.7C920738
00403EF7  .  DBE2            FCLEX
00403EF9  .v 7D 18           JGE SHORT CrackmeV.00403F13
00403EFB  .  8B8D 48FFFFFF   MOV ECX,DWORD PTR SS:[EBP-B8]
00403F01  .  68 A0000000     PUSH 0A0
00403F06  .  68 F8374000     PUSH CrackmeV.004037F8
00403F0B  .  51              PUSH ECX
00403F0C  .  50              PUSH EAX
00403F0D  .  FF15 28104000   CALL DWORD PTR DS:[<&MSVBVM60.__vbaHresultCheck  MSVBVM60.__vbaHresultCheckOt
00403F13  >  8B55 D8         MOV EDX,DWORD PTR SS:[EBP-28]               ntdll.7C920738
00403F16  .  52              PUSH EDX                                    ntdll.KiFastSystemCallRet
00403F17  .  68 0C384000     PUSH CrackmeV.0040380C                      UNICODE "g7#2+'&1,3"
00403F1C  .  FF15 4C104000   CALL DWORD PTR DS:[<&MSVBVM60.__vbaStrCmp>]  MSVBVM60.__vbaStrCmp
00403F22  .  8BF8            MOV EDI,EAX
00403F24  .  8D4D D8         LEA ECX,DWORD PTR SS:[EBP-28]
00403F27  .  F7DF            NEG EDI                                     ntdll.7C920738
00403F29  .  1BFF            SBB EDI,EDI                                 ntdll.7C920738
00403F2B  .  47              INC EDI                                     ntdll.7C920738
00403F2C  .  F7DF            NEG EDI                                     ntdll.7C920738
00403F2E  .  FF15 B0104000   CALL DWORD PTR DS:[<&MSVBVM60.__vbaFreeStr>]  MSVBVM60.__vbaFreeStr
00403F34  .  8D4D D0         LEA ECX,DWORD PTR SS:[EBP-30]
00403F37  .  FF15 B4104000   CALL DWORD PTR DS:[<&MSVBVM60.__vbaFreeObj>]  MSVBVM60.__vbaFreeObj
00403F3D  .  66:85FF         TEST DI,DI
00403F40  .v 0F84 8B000000   JE CrackmeV.00403FD1
00403F46  .  A1 24504000     MOV EAX,DWORD PTR DS:[405024]
00403F4B  .  85C0            TEST EAX,EAX
00403F4D  .v 75 10           JNZ SHORT CrackmeV.00403F5F
00403F4F  .  68 24504000     PUSH CrackmeV.00405024
00403F54  .  68 A82C4000     PUSH CrackmeV.00402CA8
00403F59  .  FF15 78104000   CALL DWORD PTR DS:[<&MSVBVM60.__vbaNew2>]  MSVBVM60.__vbaNew2
00403F5F  >  8B35 24504000   MOV ESI,DWORD PTR DS:[405024]
00403F65  .  6A FF           PUSH -1
00403F67  .  56              PUSH ESI
00403F68  .  8B06            MOV EAX,DWORD PTR DS:[ESI]
00403F6A  .  FF90 BC010000   CALL DWORD PTR DS:[EAX+1BC]
00403F70  .  85C0            TEST EAX,EAX
00403F72  .  DBE2            FCLEX
00403F74  .v 7D 12           JGE SHORT CrackmeV.00403F88
```

Vamos a poner un Breakpoint en esta lína y pulsamos F9.

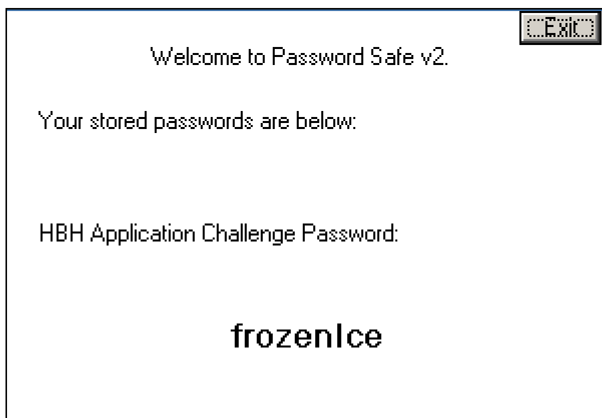
Una vez introducido una contraseña, Olly se detendrá en nuestro Breakpoint.



Un poco más abajo,en la dirección 403F40, podemos apreciar la combinación comparar/saltar. Pulsamos F8 hasta llegar a ella, cambiamos el valor de la bandera Z y ejecutamos la aplicación:



¡Este ha sido de lejos nuestro Crack más facil!



7.20 Caso práctico 20: Trabajando con binarios de Visual Basic (II)

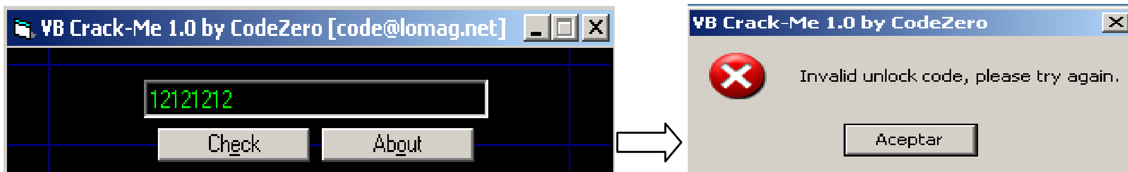
Empezamos por cargar CrackmeVB3.exe en Olly.

Vemos la tipica llamada al tiempo de ejecución de Visual Basic.

Como no sabemos mucho acerca de esta aplicación vamos a ejecutarla haciendo doble clic sobre el ejecutable:

Vemos que aparece un nag con un contador de 5 segundos.

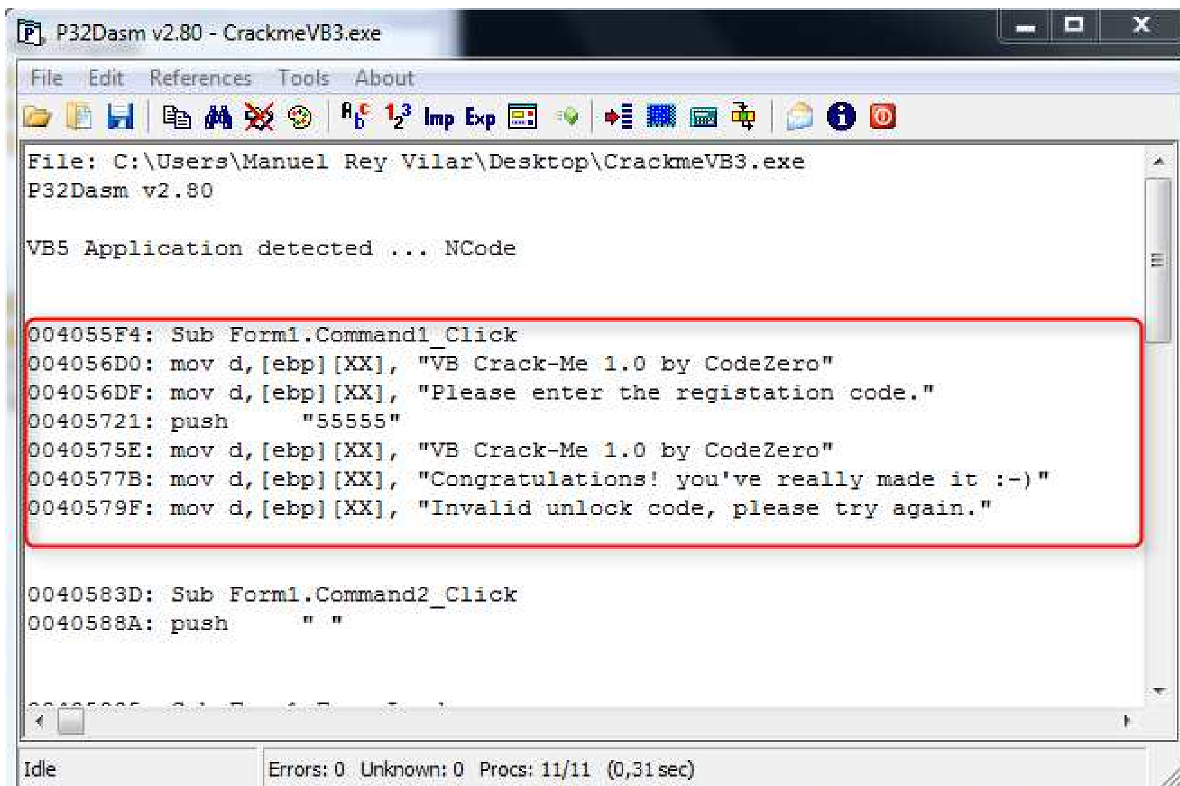
Al cabo de los cinco segundos aparece la pantalla principal del serial. Introducimos uno cualquiera y hacemos clic en “Check”.



En este ejercicio vamos a introducir otro decompilador. En lugar de utilizar VB Decompiler vamos a utilizar P32Dasm, un compilador nativo de P –code.

Cargamos CrackmeVB3.exe en P32Dasm y aparece la ventana principal con los siguientes datos:

1. Cadenas de texto:



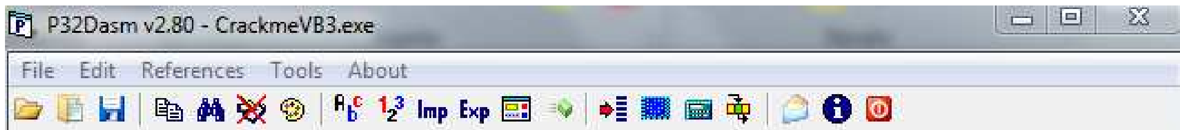
2. Botones:

```
0040579F: mov d,[ebp][XX], "Invalid unlock code, please try again."

0040583D: Sub Form1.Command2_Click
0040588A: push     " "

00405905: Sub Form1.Form_Load
00405957: push     "??????????????????????????"
00405971: push     "   ?@"

004059C2: Sub Form2.Command1_Click
00405A87: push     "??????????????????????????"

00405AC5: Sub Form2.Timer1_Timer
```

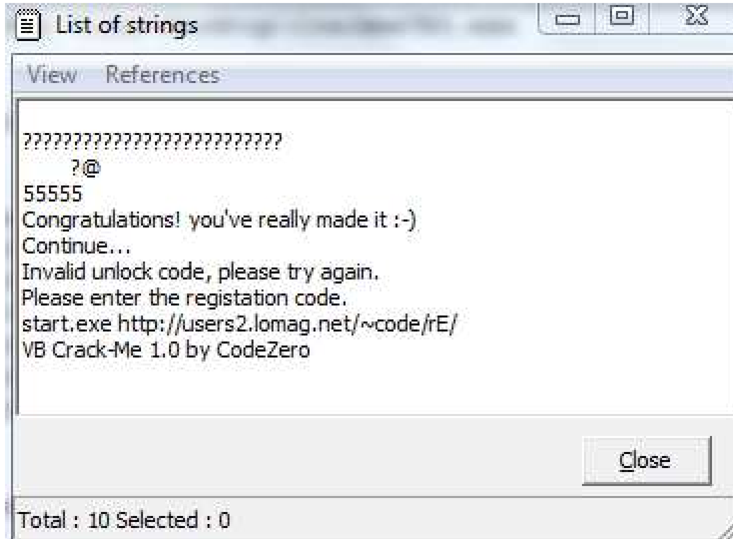
3. Contadores:

```
00405AC5: Sub Form2.Timer1_Timer

00405BCE: Sub Form2.Timer2_Timer

00405CD7: Sub Form2.Timer3_Timer

00405DE0: Sub Form2.Timer4_Timer

00405EE9: Sub Form2.Timer5_Timer

00405FF2: Sub Form2.Timer6_Timer
0040604B: push     "Continue..."
```

Podemos apreciar algunas similitudes con VB Decompiler, por ejemplo el Form1.Command1_Click, que sería la devolución de llamada después de hacer clic sobre un botón. En la barra de herramientas aparecen los botones más significativos de la aplicaicón:

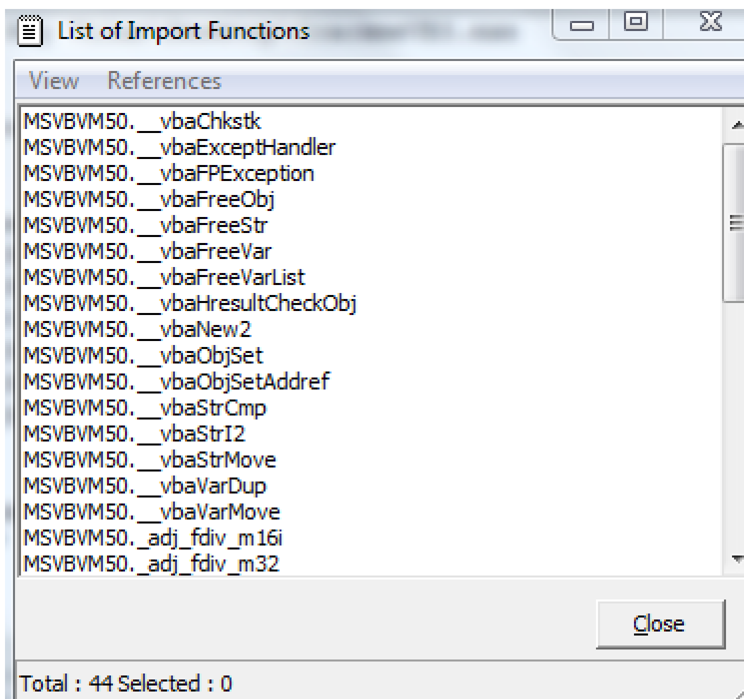


Si hacemos clic en “Strings”, veremos algo similar al “Seachr for” -> “Strings” en Olly:



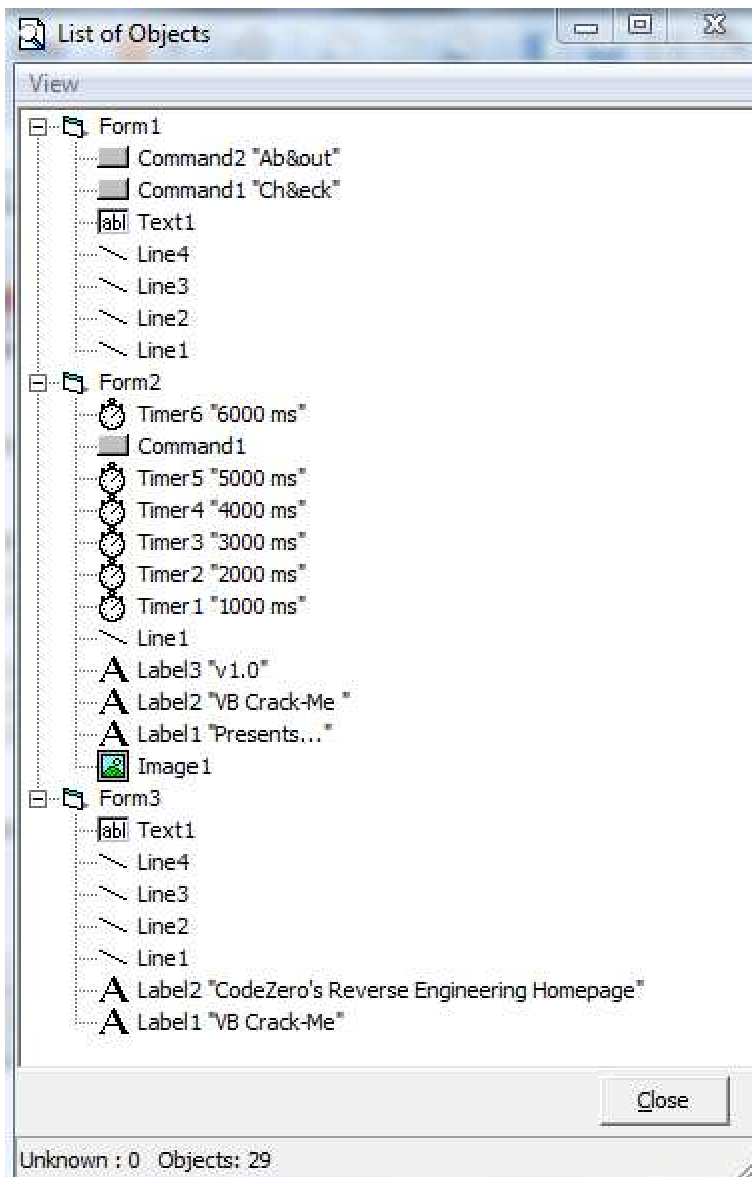
A diferencia de Olly, si hacemos doble clic sobre cualquier de estas cadenas de texto no nos lleva a la sección de código dentro del desensamblador.

El botón de “Numbers” tampoco nos revela ningún tipo de información. Luego tenemos el botón de “Imports”, que es similar al de “All intermodular calls” en Olly:



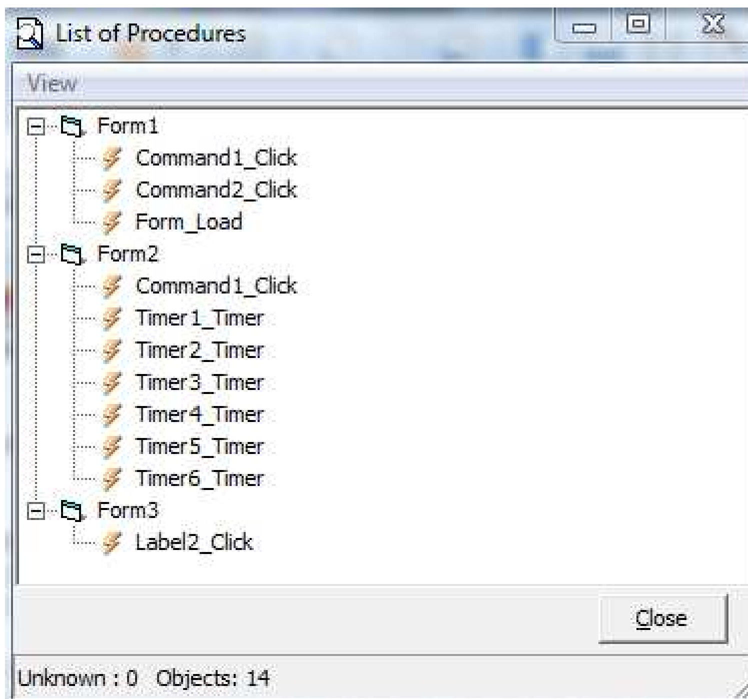
El siguiente botón es el de “Exports”. En nuestro caso está vacío.

El botón de “Objects” debería recordarnos a la pantalla del VB Decompiler:



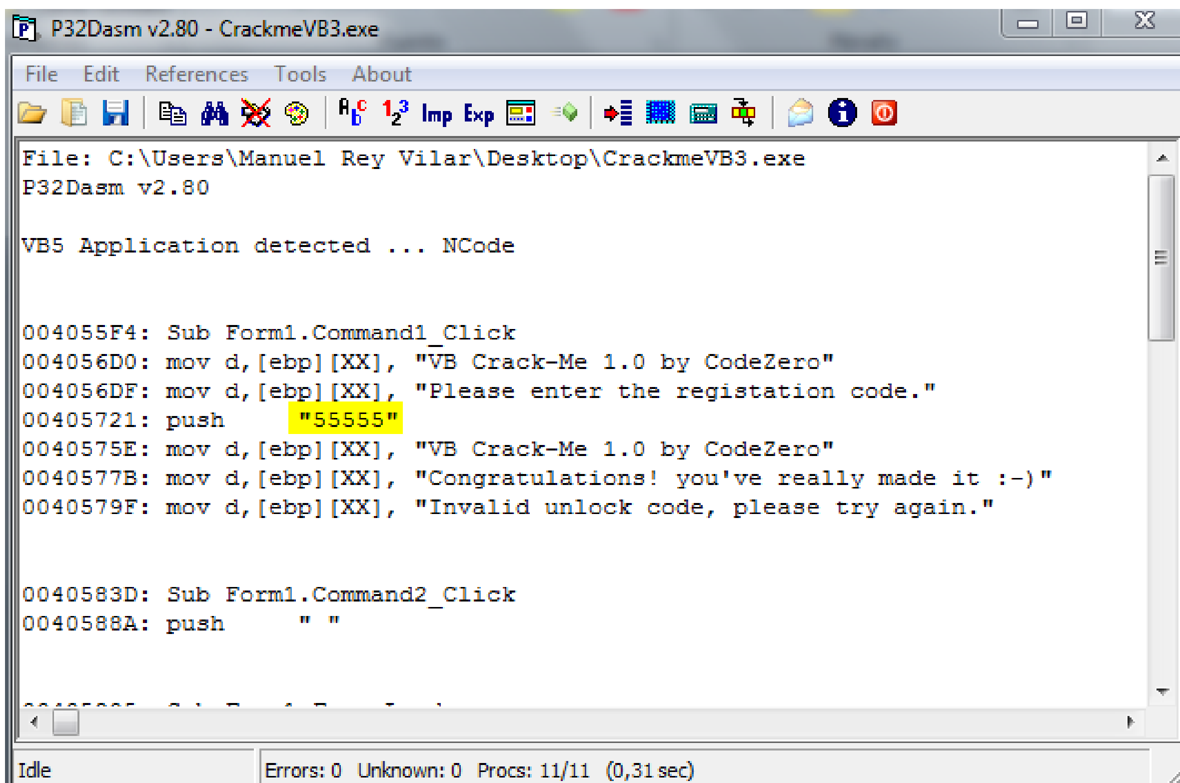
Podemos apreciar todos los objetos de la aplicación, como son los botones, las etiquetas y los contadores. Podemos ver claramente que el botón "Check" se denomina "Command1". Y en este caso será el botón de devolución de llamada.

Por último tenemos el botón de "Procedures":

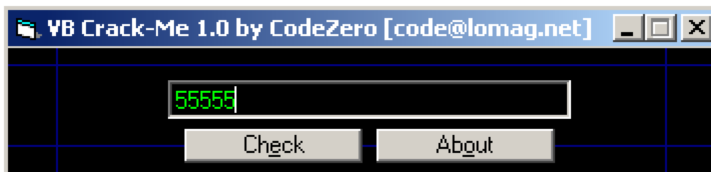


Esta ventana nos muestra todas las devoluciones de llamadas de nuestra aplicación. Podemos ver que la devolución de llamada de nuestro botón "Check" se llama "Command1_Click", donde Command1 es la devolución de llamada de nuestro botón "Check".

Una cosa a destacar en la sección de la cadena de textos son los 5 cincos que aparecen ahí.



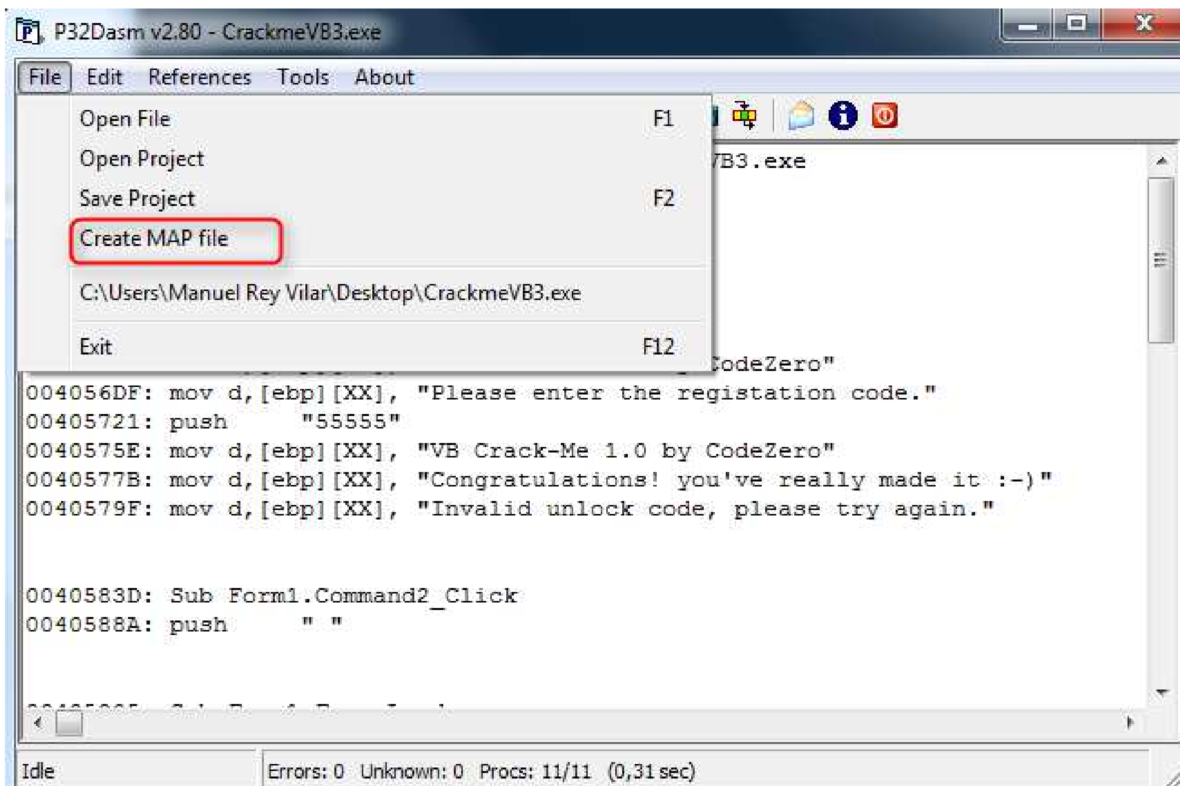
Será que… ¡No, no puede ser tan facil! Probemos a ver que pasa…



Bueno, seguiremos analizando este Crackme para ver que más opciones hay en P32Dasm. Una de las herramientas que tenemos a nuestra disposición son los ficheros MAP.

Un archivo MAP es una collección de nombres para procedimientos de CALL's que han sido compilados en el código de Visual Basic. Recordemos que Visual Basic utiliza cadenas de nombres para hacer referencia a las devoluciones de llamadas, de forma que podemos extrapolarlos e importarlos a Olly. Podemos hacerlo con VB Decompliler Pro, pero como la versión Pro no es gratis, utilizaremos para ello P32Dasm.

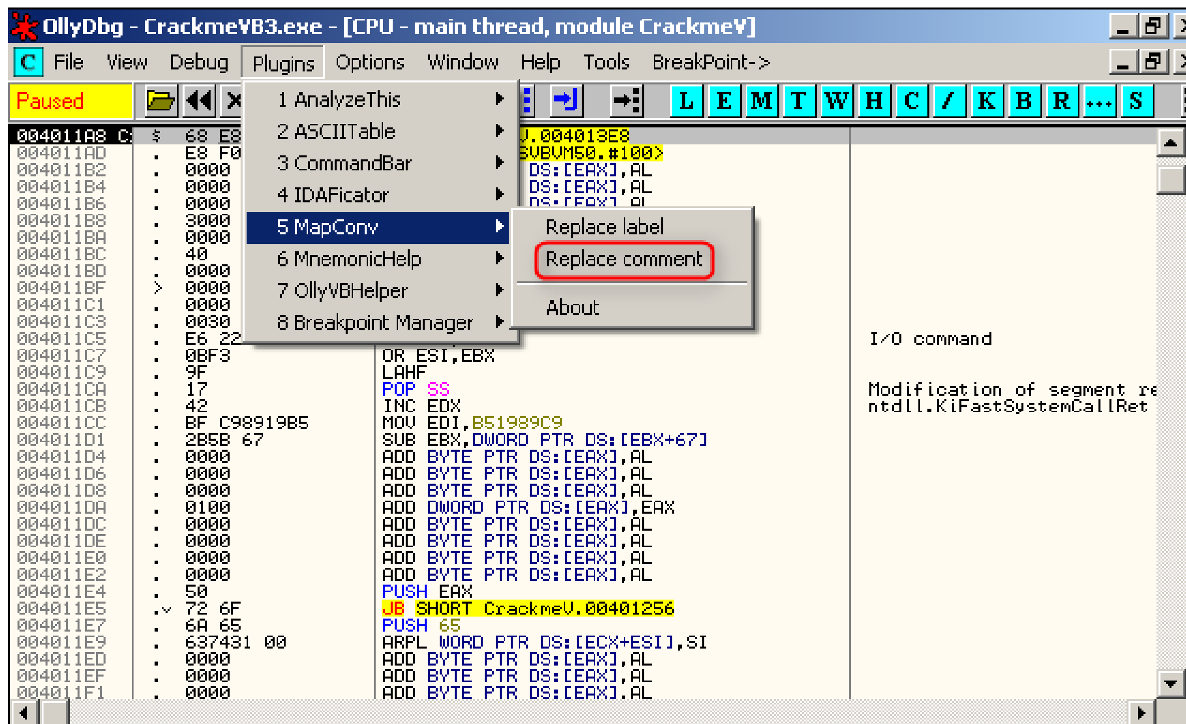
Seleccionamos "File" -> "Create MAP file". Guardamos el archivo.



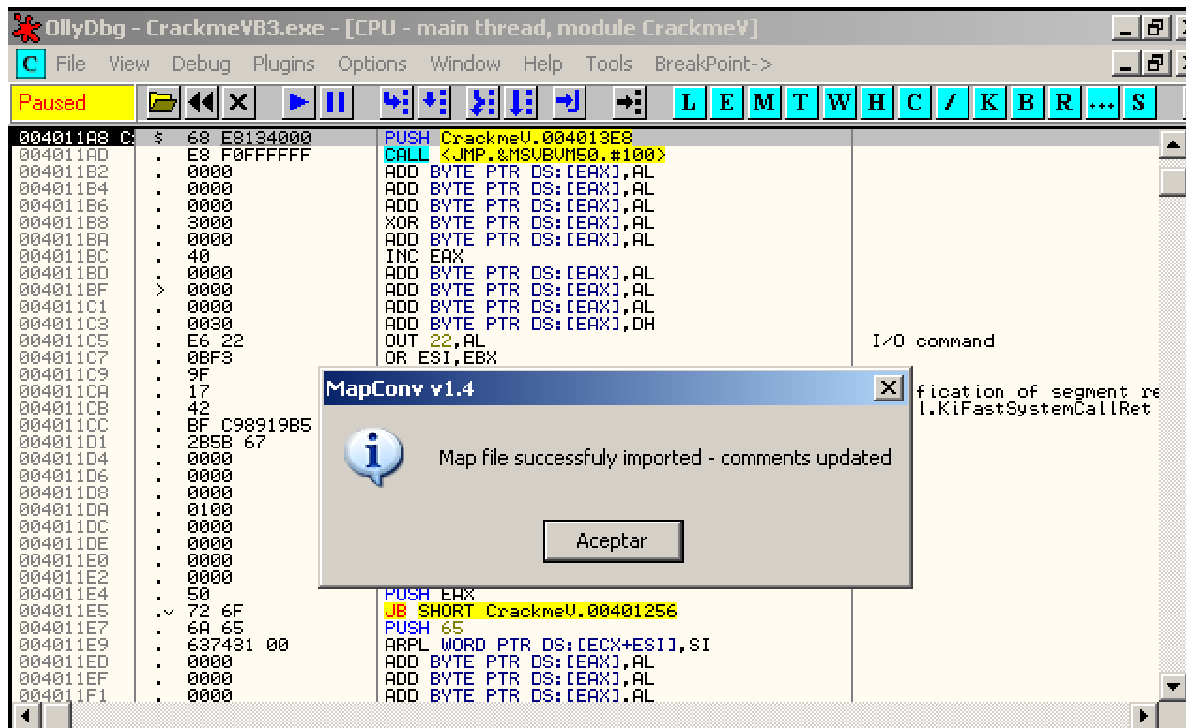
Cargamos CrackmeVB3.exe en Olly y vamos a la dirección 4055F4, que es la dirección de la devolución de llamada de Command1_Click.

Vemos por lo tanto donde empieza la devolución de llamada para el botón "Check". Para importar el archivo MAP en Olly, utilizaremos el plugin MAP converter. (Copiamos el archivo DLL y lo guardamos en la carpeta de plugins de Olly).

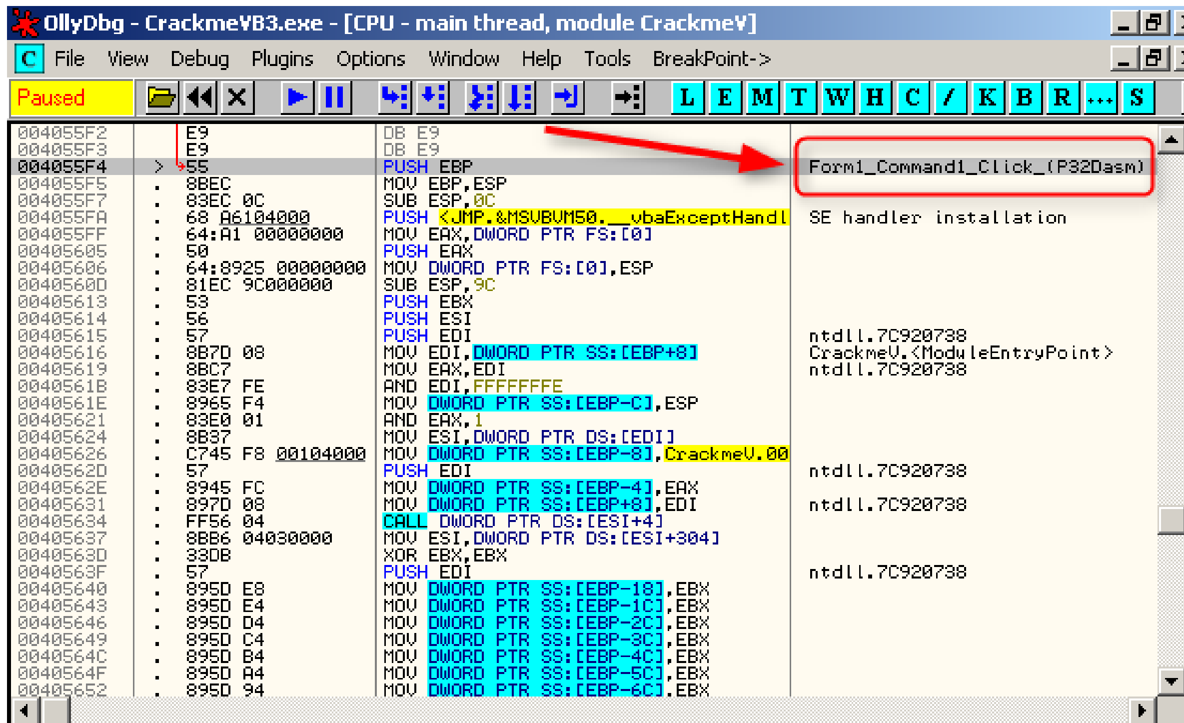
Reiniciamos Olly y volvemos a cargar la aplicación. Seleccionamos "Plugins" -> "MapConv" -> "Replace Comment".



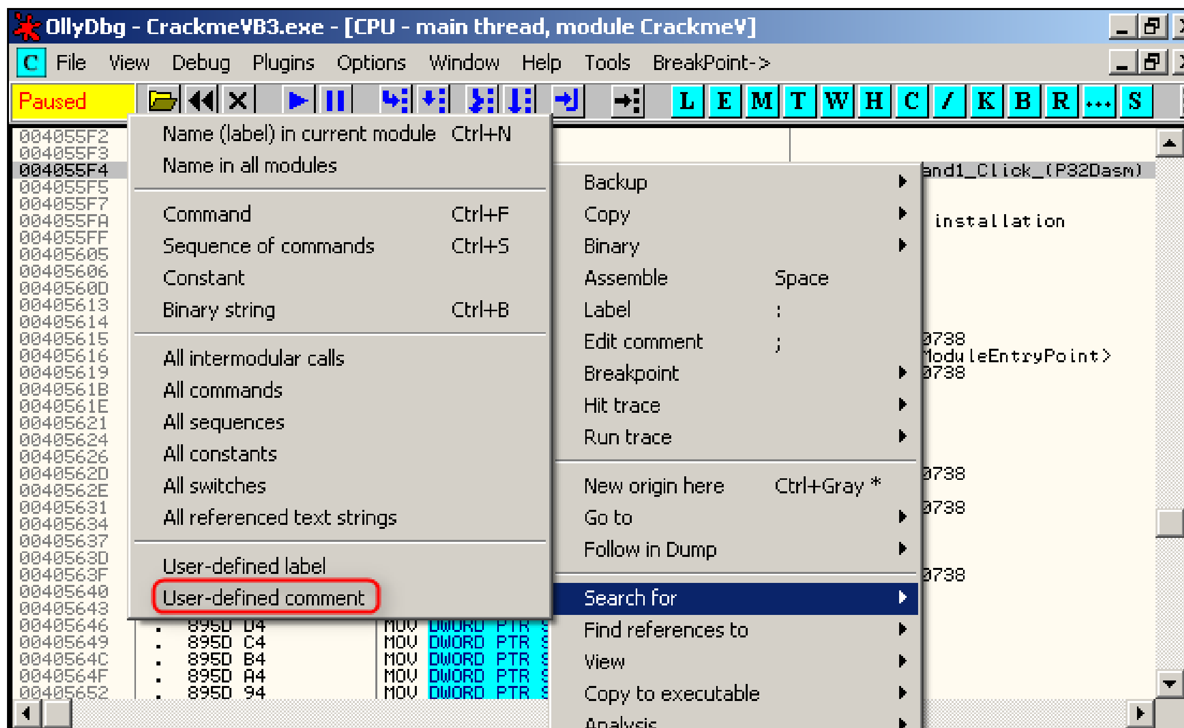
Seleccionamos el archivo MAP que hemos creado con P32Dasm. Esto nos permite poner la información del archivo MAP en la columna de comentarios de Olly.



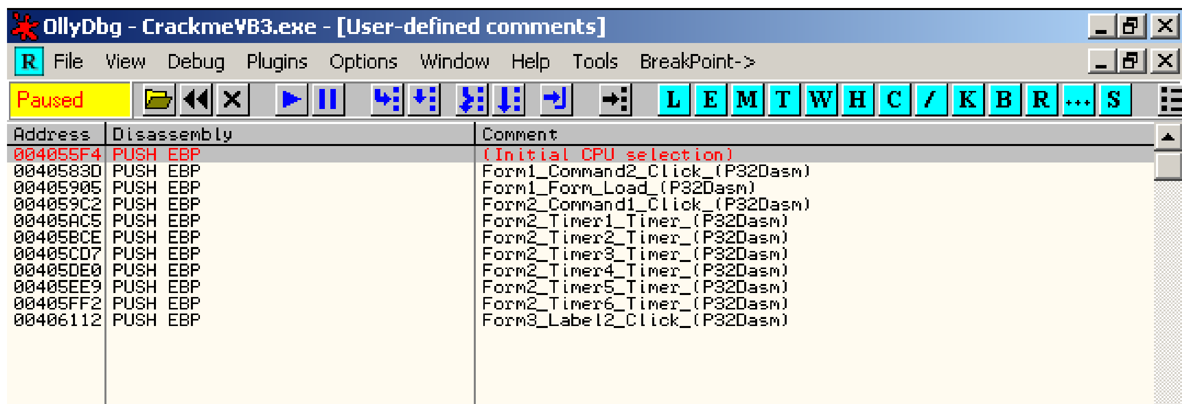
Si nos fijamos ahora en nuestra devolución de llamada, podemos ver el comentario que nos permite seguir investigando la aplicación:



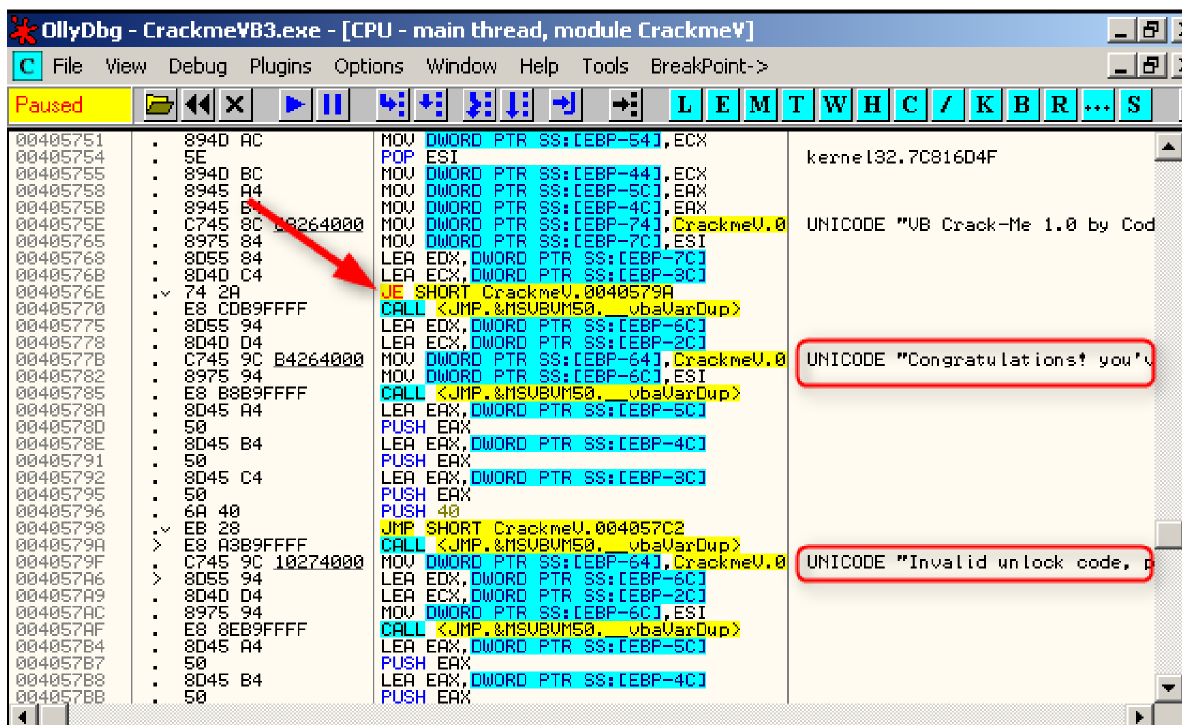
A continuacións hacemos clic con el botón derecho y seleccionamos “Search for” -> “User-defined comment”:



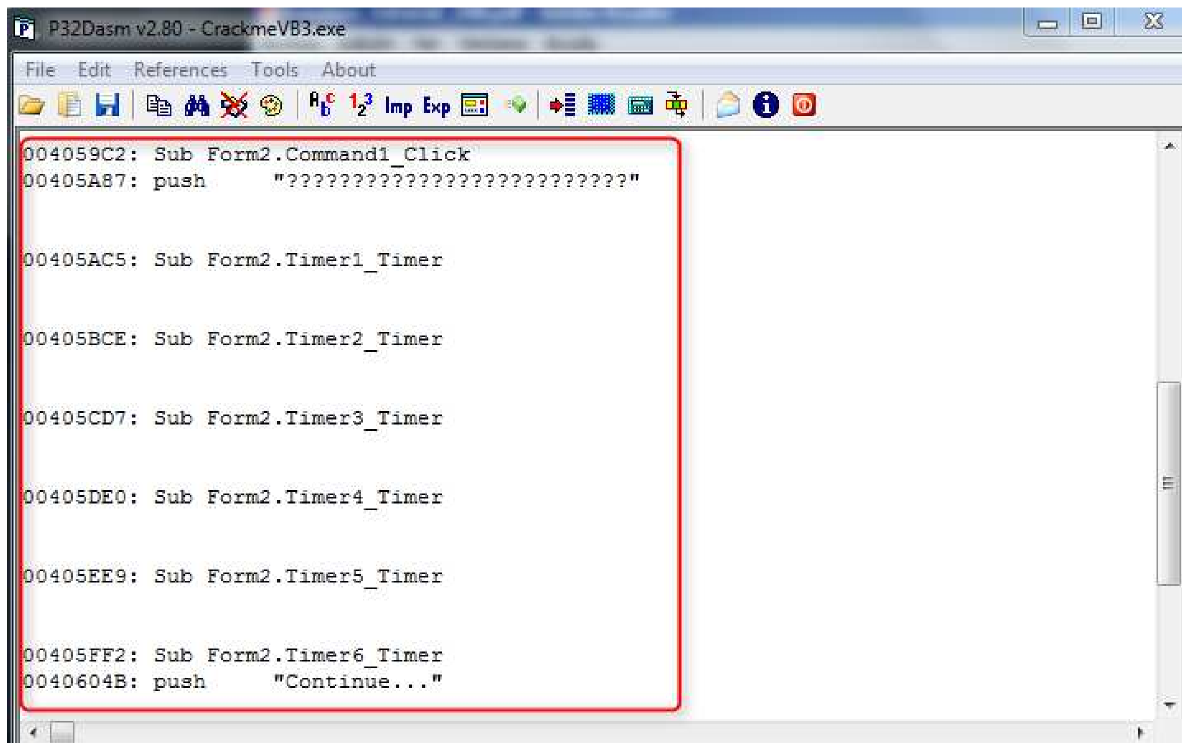
Y podremos ver todos los nombres de nuestras devoluciones de llamadas:



Bajamos un poco desde la devolución de llamada Command1_Click y podemos ver el “good boy”, el “bad boy” y el parche que tenemos que hacer:



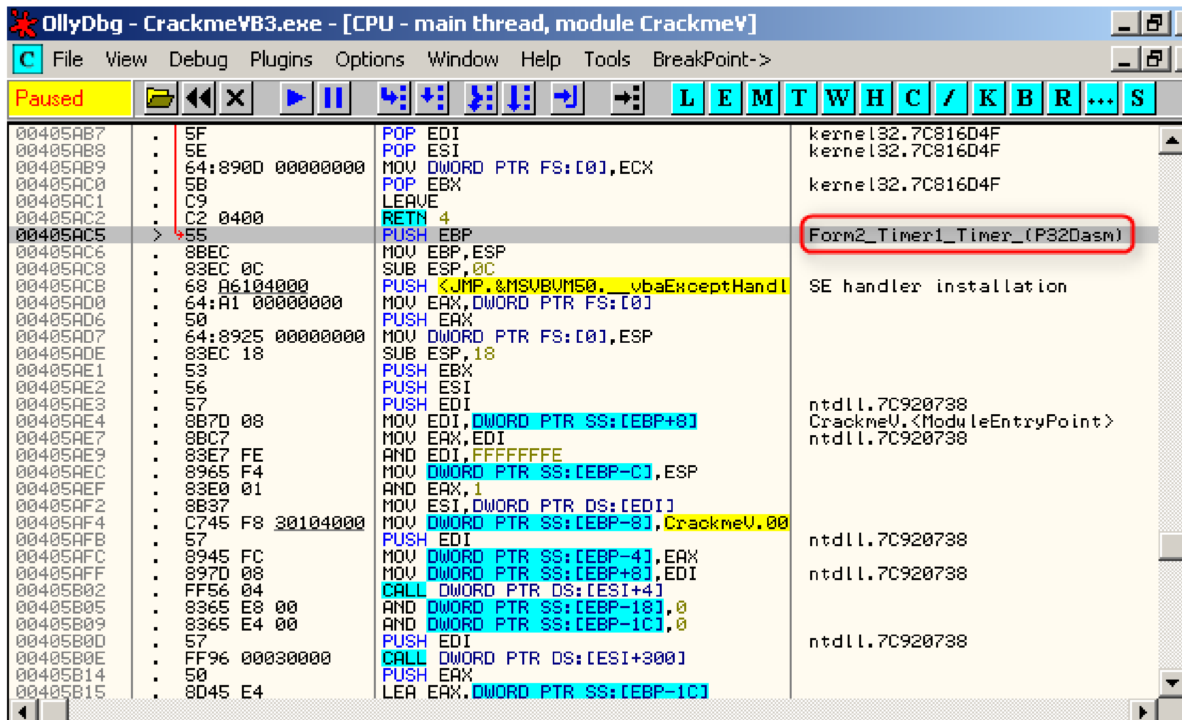
Ahora lo único que nos queda por hacer es eliminar el Nag. Volviendo a P32Dasm echemos un vistazo a los CALL’s de los contadores:



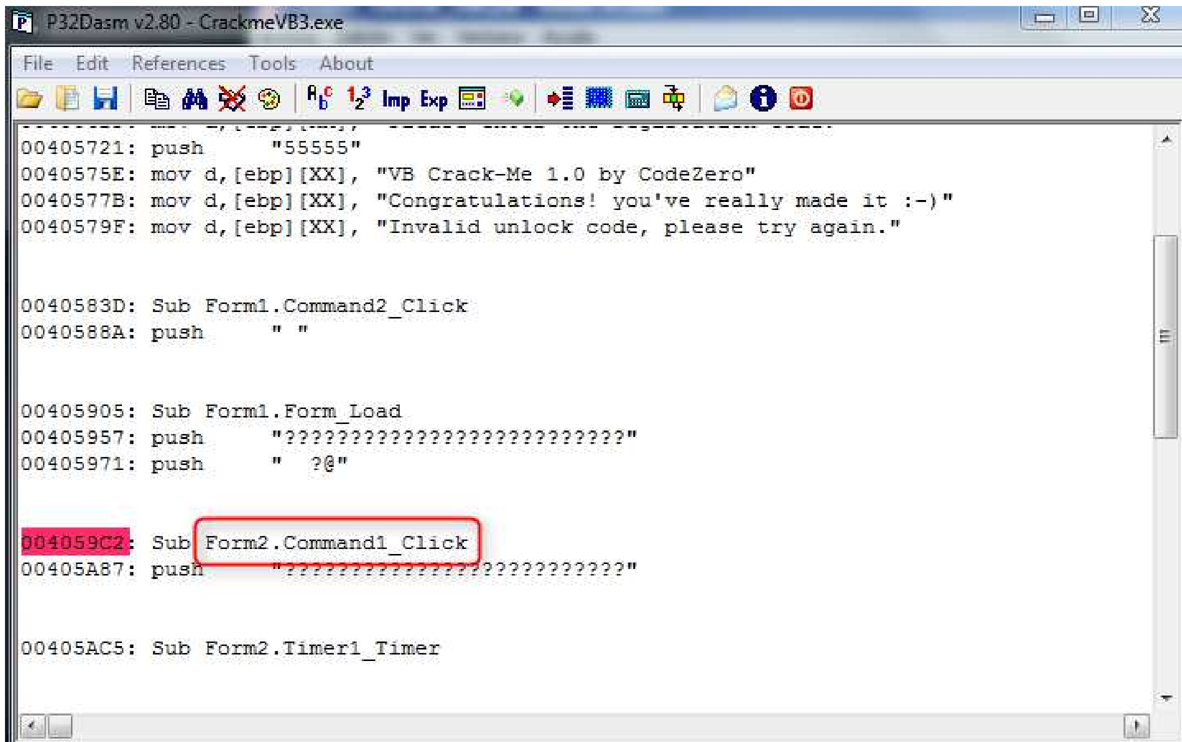
Podemos ver que Form2 es la pantalla del nag, ya que es el que llama a los contadores. También podemos ver que Form2.Command1_Click es la devolución de llamada para hacer clic en el botón “OK” después de que el contador haya finalizado.

Una solución sencilla sería sobreescribir el primer call al contador y hacerlo saltar a la devolución de la llamada para el clic del botón “OK”.

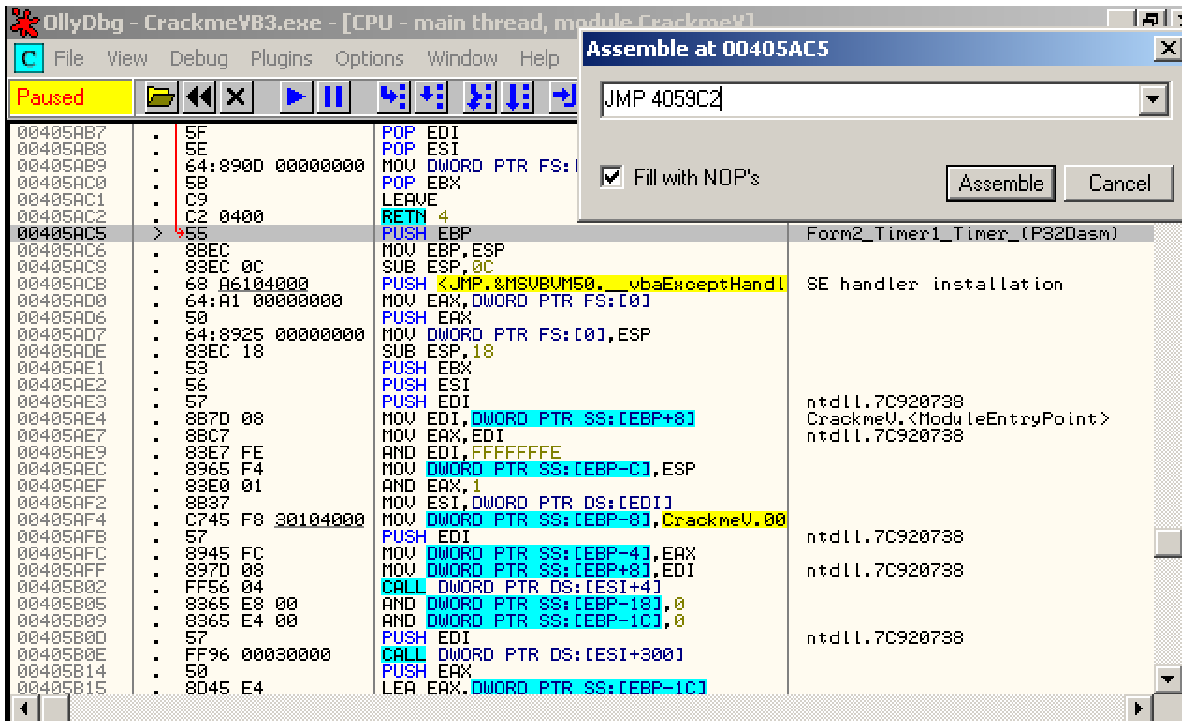
Si vamos a la dirección 405AC5, vemos el comienzo del primer contador:



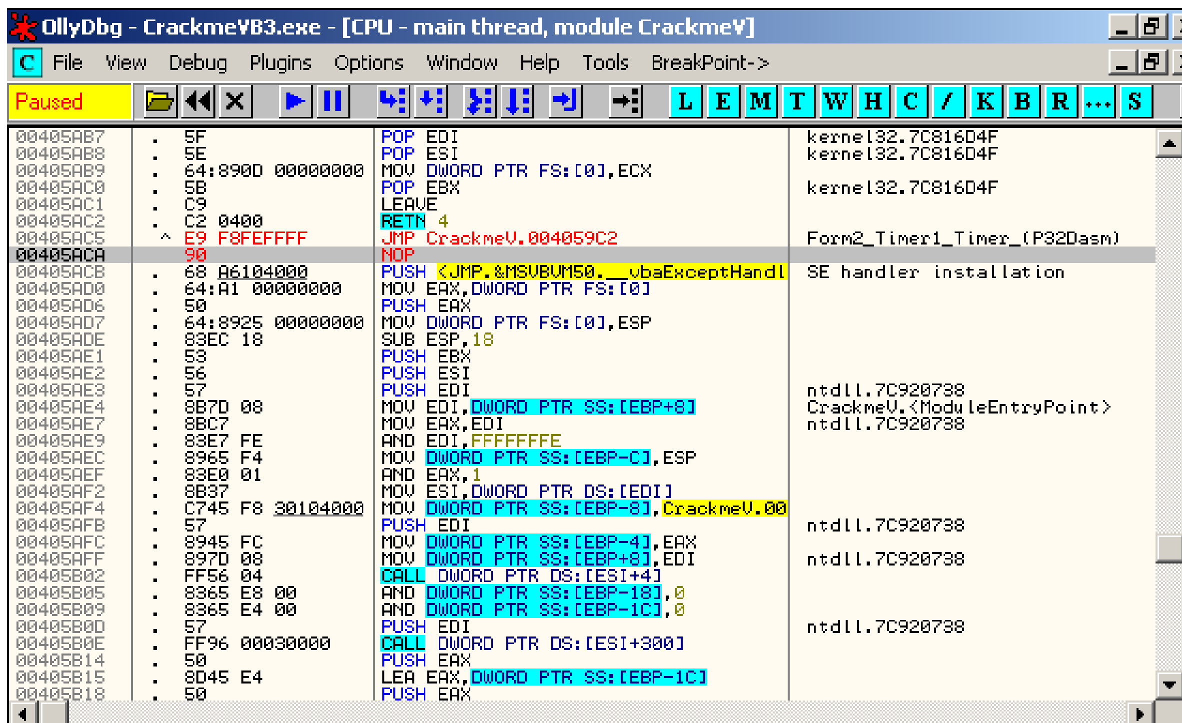
Cambiemos el comienzo para que apunte directamente al código responsable de cerrar la pantalla nag. En P32Dasm podemos ver que esa devolución de llamada está en la dirección 4059C2:



Procedamos a parchearlo:

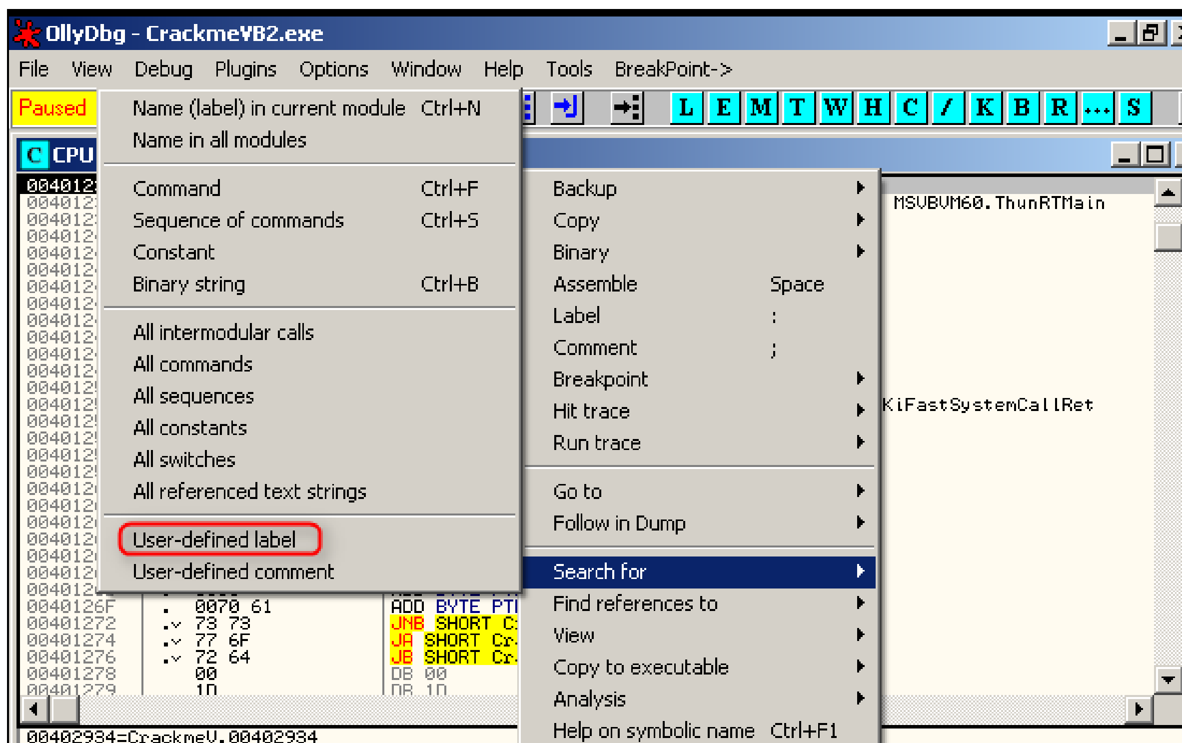


Saltamos a la dirección que cierra nuestro nag.

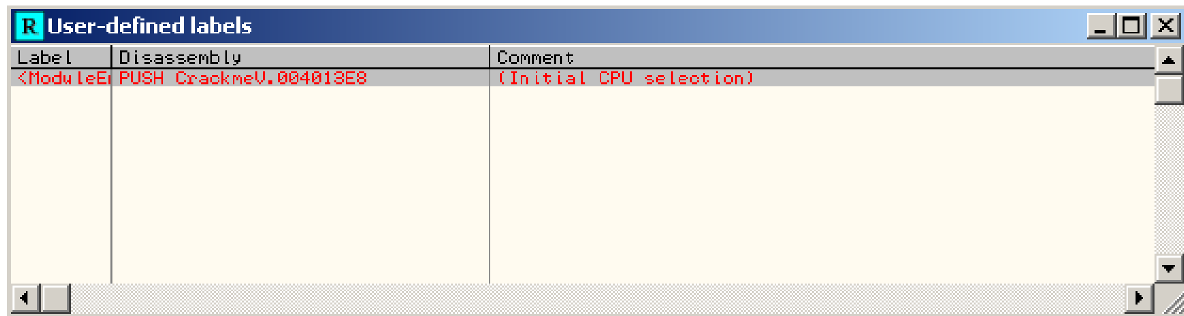


Si reiniciamos la aplicación podemos ver como aparece el nag durante unos pocos segundos para luego desaparecer.

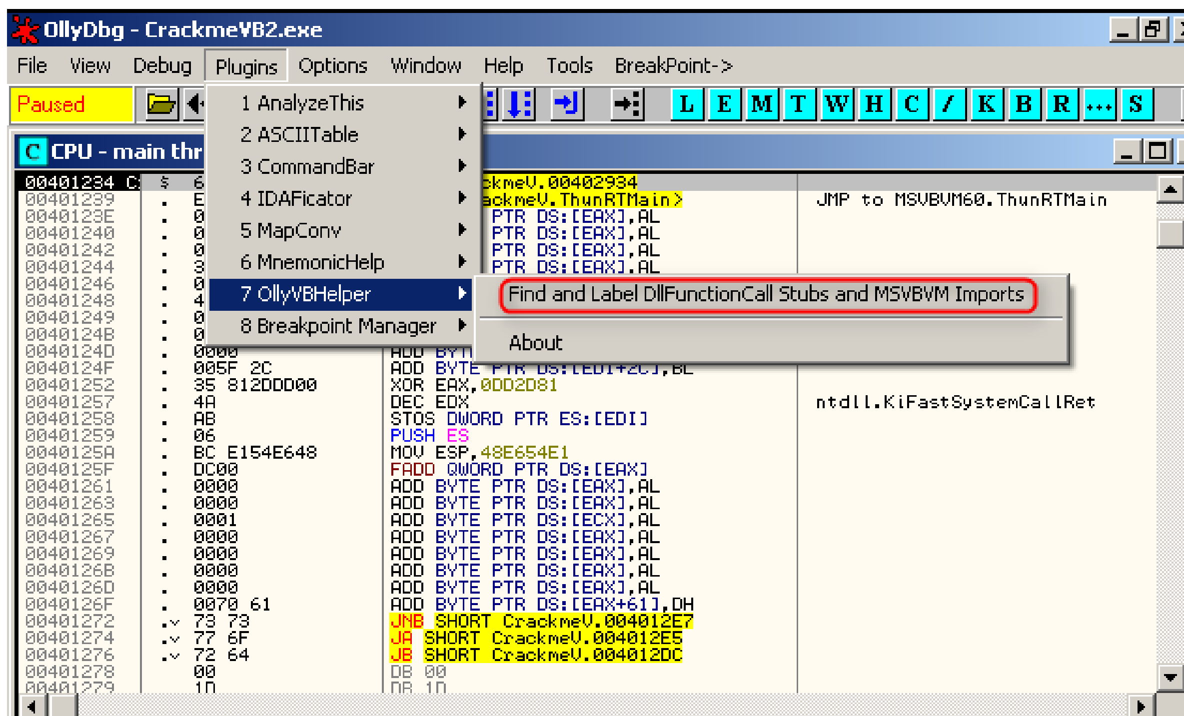
Otra herramienta util en la batalla con Visual Basic es el plugin OllyVBHelper. El proposito de este plugin es encontrar y reediquetar la importaciones compiladas de Visual Basic (DLL’s). Como ejemplo podemos cargar cualquier Crackme, hacemos clic con el botón derecho para seleccionar “Search for” -> “User defined labels”:



Antes de usar el plugin esta ventana está vacía:



Ejecutamos el plugin:



Y podemos ver todas las llamadas a nuestros métodos, similar a cuando importamos el fichero MAP.

User-defined labels

Label	Disassembly	Comment
<__vbaCh	JMP DWORD PTR DS:[<&MSVBVM60.__vb	__vbaChkstk
<__vbaEx	JMP DWORD PTR DS:[<&MSVBVM60.__vb	__vbaExceptHandler
<__vbaFP	JMP DWORD PTR DS:[<&MSVBVM60.__vb	__vbaFPException
<_adj_fd	JMP DWORD PTR DS:[<&MSVBVM60._adj	_adj_fdiv_m16i
<_adj_fd	JMP DWORD PTR DS:[<&MSVBVM60._adj	_adj_fdiv_m32
<_adj_fd	JMP DWORD PTR DS:[<&MSVBVM60._adj	_adj_fdiv_m32i
<_adj_fd	JMP DWORD PTR DS:[<&MSVBVM60._adj	_adj_fdiv_m64
<_adj_fd	JMP DWORD PTR DS:[<&MSVBVM60._adj	_adj_fdiv_r
<_adj_fd	JMP DWORD PTR DS:[<&MSVBVM60._adj	_adj_fdivr_m16i
<_adj_fd	JMP DWORD PTR DS:[<&MSVBVM60._adj	_adj_fdivr_m32
<_adj_fd	JMP DWORD PTR DS:[<&MSVBVM60._adj	_adj_fdivr_m32i
<_adj_fd	JMP DWORD PTR DS:[<&MSVBVM60._adj	_adj_fdivr_m64
<_adj_fp	JMP DWORD PTR DS:[<&MSVBVM60._adj	_adj_fpatan
<_adj_fp	JMP DWORD PTR DS:[<&MSVBVM60._adj	_adj_fprem
<_adj_fp	JMP DWORD PTR DS:[<&MSVBVM60._adj	_adj_fprem1
<_adj_fp	JMP DWORD PTR DS:[<&MSVBVM60._adj	_adj_fptan
<_CIatan	JMP DWORD PTR DS:[<&MSVBVM60._CIa	_CIatan
<_CIexp>	JMP DWORD PTR DS:[<&MSVBVM60._CIe	_CIexp
<_CIlog>	JMP DWORD PTR DS:[<&MSVBVM60._CIl	_CIlog
<_CIsin>	JMP DWORD PTR DS:[<&MSVBVM60._CIs	_CIsin
<_CIsqrt	JMP DWORD PTR DS:[<&MSVBVM60._CIs	_CIsqrt
<_CItan>	JMP DWORD PTR DS:[<&MSVBVM60._CIt	_CItan
<_allmul	JMP DWORD PTR DS:[<&MSVBVM60._all	_allmul
<__vbaFr	JMP DWORD PTR DS:[<&MSVBVM60.__vb	__vbaFreeVarList
<__vbaFr	JMP DWORD PTR DS:[<&MSVBVM60.__vb	__vbaFreeVar
<__vbaEn	JMP DWORD PTR DS:[<&MSVBVM60.__vb	__vbaEnd
<__vbaSt	JMP DWORD PTR DS:[<&MSVBVM60.__vb	__vbaStrCat
<__vbaVa	JMP DWORD PTR DS:[<&MSVBVM60.__vb	__vbaVarDup
<rtcMsgB	JMP DWORD PTR DS:[<&MSVBVM60.#595	rtcMsgBox
<__vbaVa	JMP DWORD PTR DS:[<&MSVBVM60.__vb	__vbaVarMove
<__vbaFr	JMP DWORD PTR DS:[<&MSVBVM60.__vb	__vbaFreeObjList
<__vbaFr	JMP DWORD PTR DS:[<&MSVBVM60.__vb	__vbaFreeStrList

7.21 Caso práctico 21: Técniquas anti-depurador (anti-debugging)

Las técniquas anti-debugging son métodos usados para engañar a los depuradores, haciendo el trabajo para un ingeniero inverso tan dificil que el analista no se siente lo suficientemente motivado como para crackear un programa. Algunos de ellos trabajan sobre desensambladores estáticos (como IDA Pro) y otros sobre depuradores como Olly o SoftICE. Los depuradores se pueden clasificar en dos categorías: de barrido líneal y de recorrido recursivo. Algunas técniquas anti-debugging se emplean especificamente contra ese tipo de depuradores y las otras en aquellos depuradores que explotan fallos en el mecanismo de depuración en general.

Una de las técniquas anti-debugging más obvias es la ofuscación de código, con la que se persigue hacer un código lo más ilegible posible. Aquí se incluyen métodos como el código espagheti (saltando de un sitio para otro sin rumbo definido), cadenas encriptadas, haciendo que el método de las llamadas a nombres sea irrelevante (para código interpretado como Visual Basic y .NET), y la ofuscación del flujo de código, donde el flujo del código no sigue una dirección lineal.

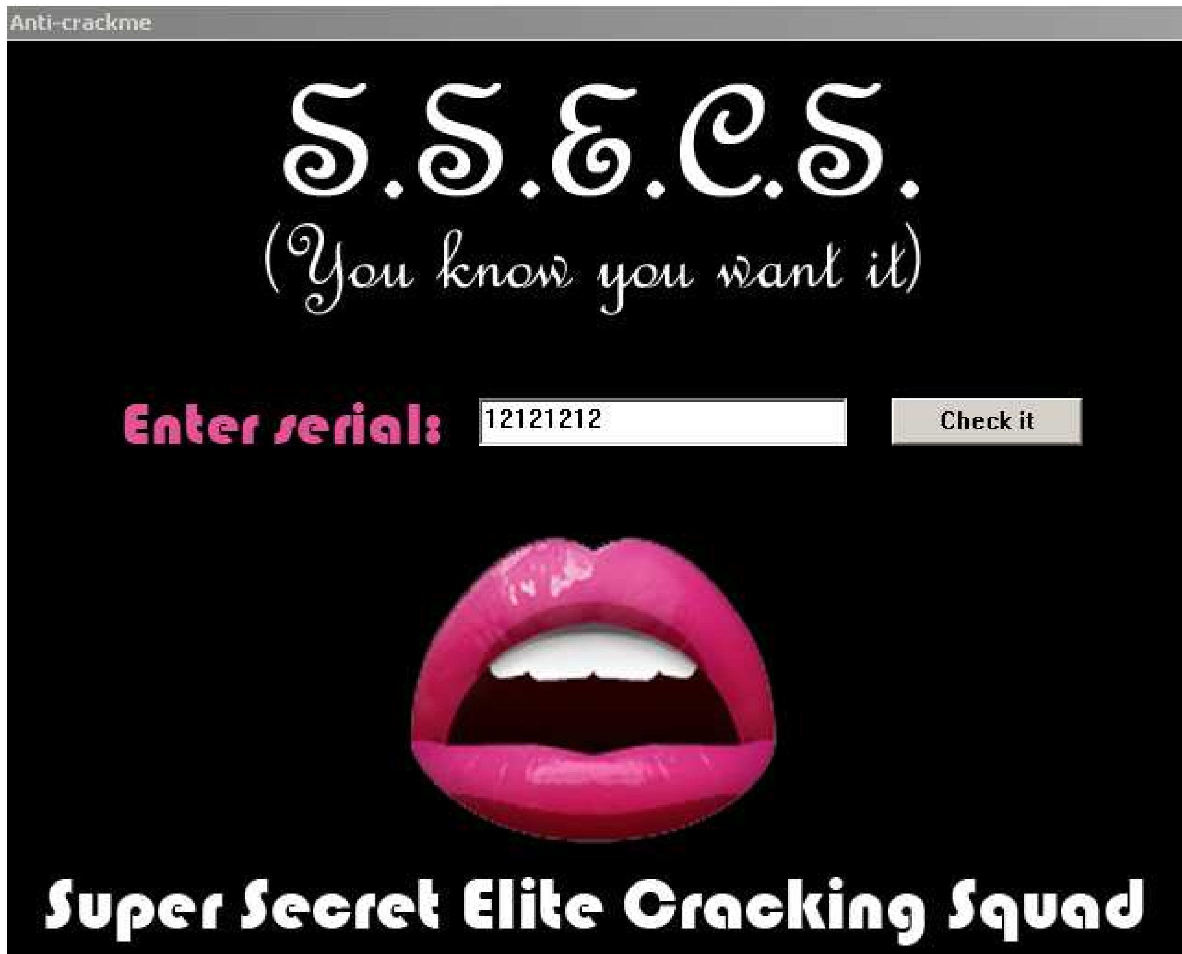
Otro tipo de técniquas son el código auto modificado y los polimorfismos, técniquas que ya se vieron en ejercicios anteriores y que se emplean fundamentalmente en los virus más robustos y en malware en general. El código auto modificado es una técniqua donde los opcodes de un binario son cambiados de forma dinámcia (en tiempo de ejecución), haciendo imposible el estudio del código sin ejecutarlo línea por línea. Polimorfismo es la técniqua de cambiar el código del binario mientras mantiene su funcionalidad cada vez que se copie el binario.

Además existen otras tecniquas que tienen que ver con la forma en que el sistema operativo maneja la depuración. Aquí se incluyen las llamadas a las funciones API de Windows que nos dicen si nuestro objetivo esta siendo depurado, comprueban de forma dinámica la existencia de puntos de ruptura en el código, eliminan puntos de ruptura de hardware y utilizan fallos conocidos para intentar sabotear el depurador.

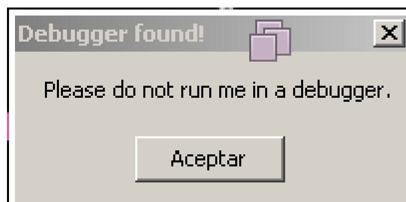
Estudiaremos algunos de estos métodos en este ejercicio aunque podemos estar seguros de que existen formas mucho más complicadas en la vida real.

Ejecutamos AntiCrackme.exe haciendo doble clic sobre el binario:

Introducimos un serial:



Pulsamos sobre el botón “Check it” y recibimos como respuesta:

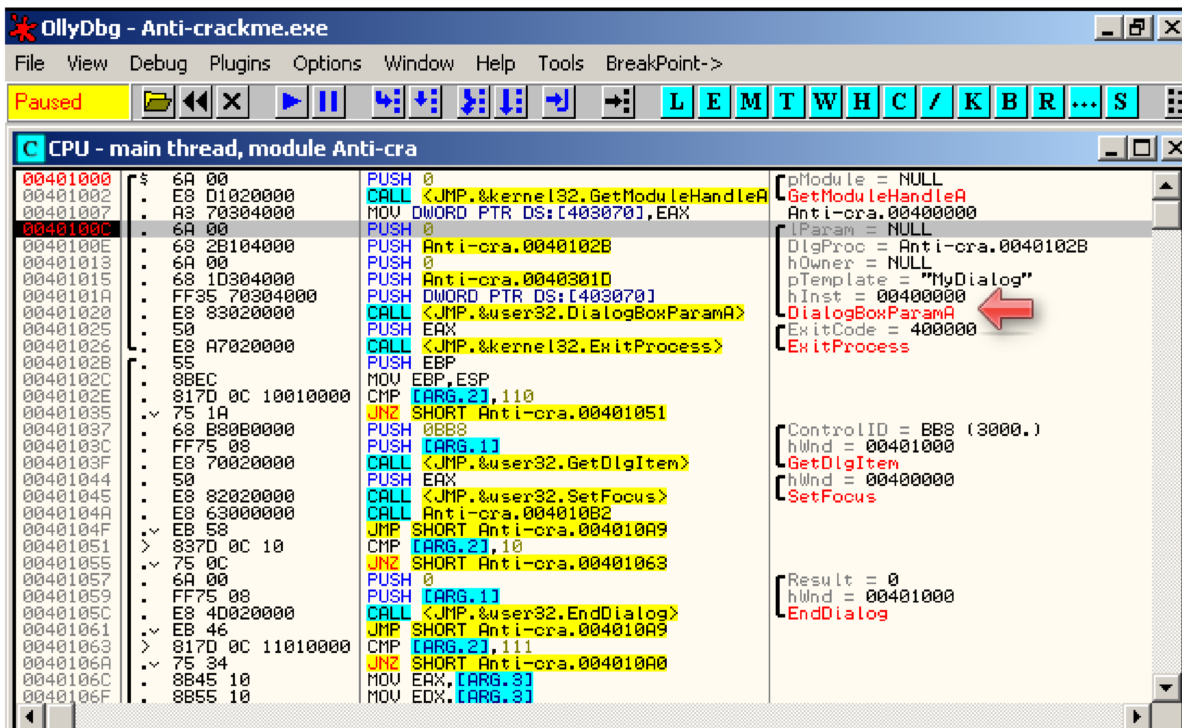


Abrimos Olly y cargamos la aplicación:

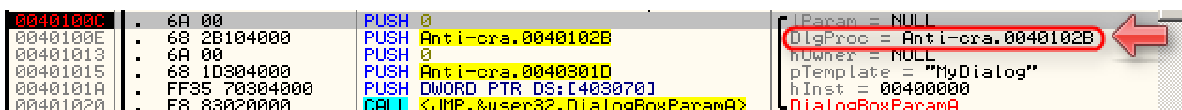
Inspeccionando un poco por el binario vemos rapidamente que hay algo inusual en el:

Ejecutamos la aplicación en Olly pulsando F9. Se abre la ventana siguiente:

Si nos fijamos en el principio de la aplicación vemos que se inicia con un cuadro de dialogo como ventana principal. Si ponemos un Breakpoint en la dirección 40100C y ejecutamos línea por línea (pulsando F8), tampronto pasamos el CALL de la dirección 401020 hacia DialogBoxParamA, verificaremos que toda la aplicación está contenida en ese CALL ya que todo el código es ejecutado desde devoluciones de llamada basadas en los eventos de ese cuadro de dialogo.



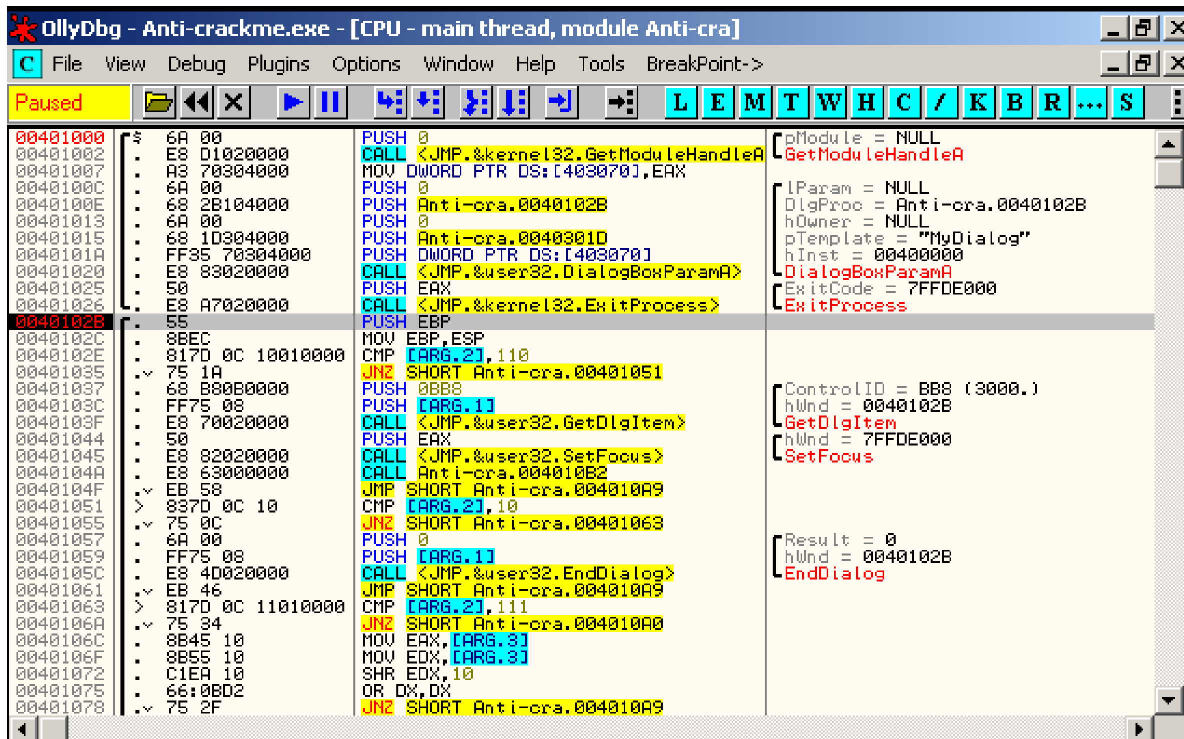
En Windows, cuando un dialogo se usa como ventana principal, ello significa que tiene que haber una devolución de llamada principal que es llamada en primer lugar y que se suele denominar “DlgProc”. Para encontrar la dirección de esta devolución de llamada principal, miraremos las variables que son pasadas a DialogBoxParamA. Como vemos en la siguiente pantalla el “DlgProc” tiene el valor de 40102B. Esta es nuestra principal dirección de la devolución de llamada DlgProc.



Si miramos el código fuento podemos ver que se trata de un asunto bien claro:

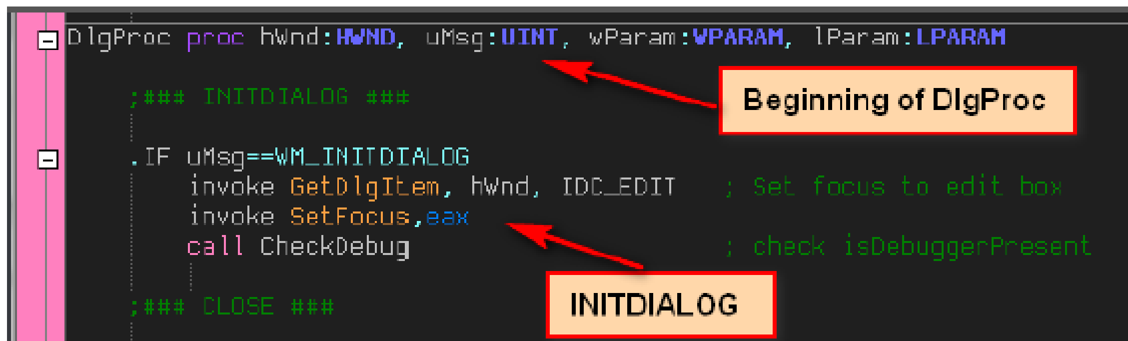
```
.code
start:

    ; set up main dialog window

    invoke GetModuleHandle, NULL
    mov    hInstance,eax
    invoke DialogBoxParam, hInstance, ADDR DlgName,NULL,addr
    invoke ExitProcess,eax
```

A continuación pondremos un Breakpoint en la dirección de DlgProc y ejecutamos la aplicación, dejando que Windows corra hasta que la devolución de llamada es llamada, para detenerse aquí. Ponemos un Breakpoint en 40102B y pulsamos F9. Nos detendremos al comienzo de DlgProc.

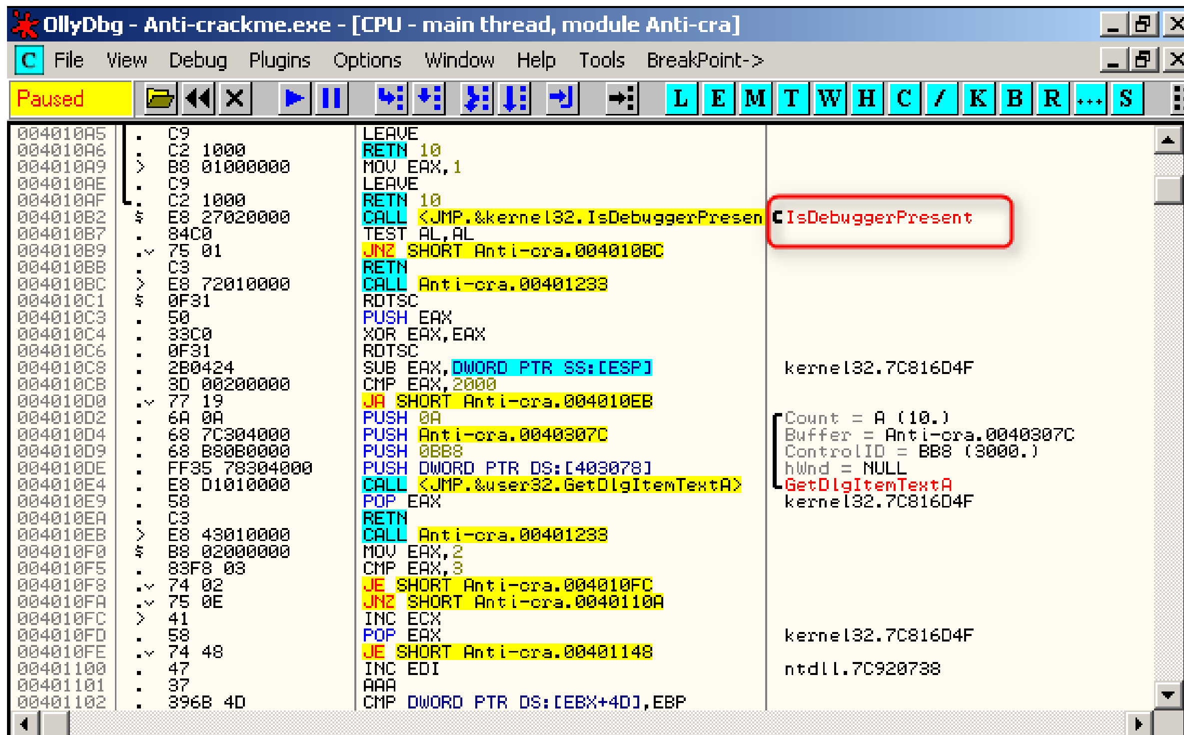


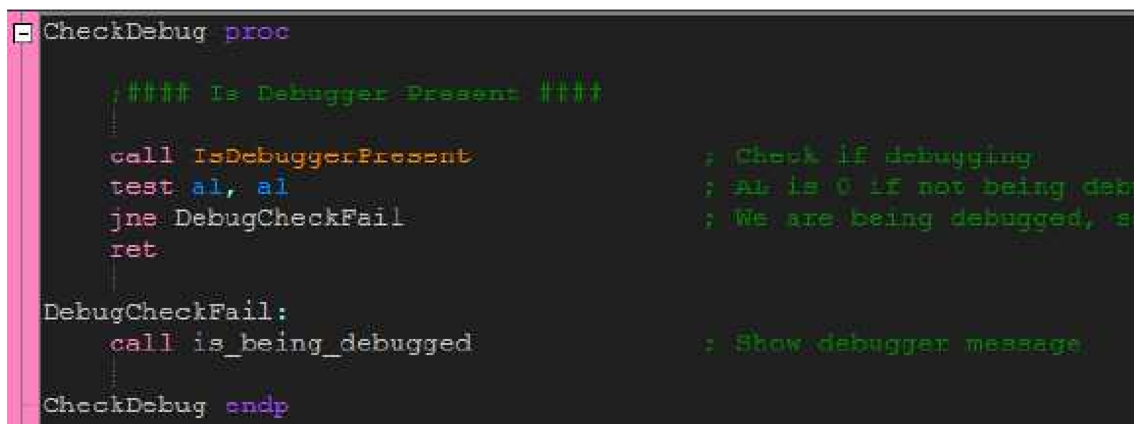
Podemos ver que primero vamos a comparar un argumento con 110h, que es el ID para el mensaje de Windows WM_INITDIALOG, o el código inicializador del dialogo.

En esta sección hay un CALL a GetDigItem y SetFocus. Esto lo que hace es enfocar el cuadro de edición del serial cuando la ventana se carga por primera vez, de tal forma que cuando empezamos a escribir el serial, este saldra en el cuador de edición. Echamos un vistazo al codigo fuento para averiguar como se consigue esto:



Si ahora nos fijamos en el CALL en CheckDebug, vemos que aparece en la dirección 40104A en el desensamblador. Si seguimos a este call, pulsando F7, saltaremos a la rutina del CheckDebug. Y seguiendo el CALL iremos al método CheckDebug que incluye un CALL a IsDebuggerPresent:



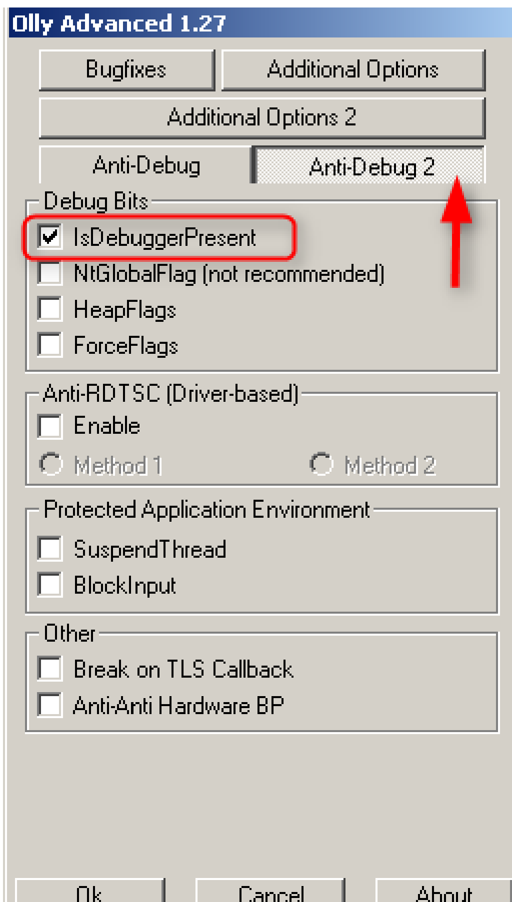


IsDebuggerPresent

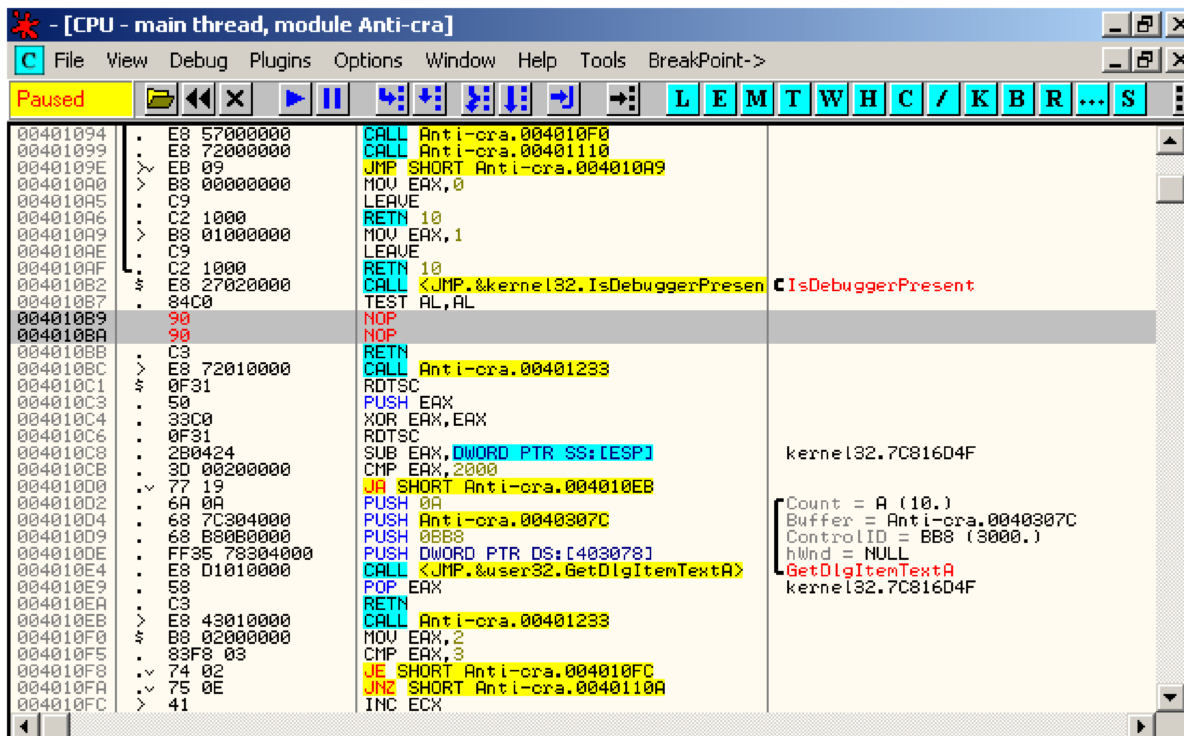
Como podemos ver en el desensamblador estamos llamando a IsDebuggerPresent, y si devuelve 'True' saltaremos al CALL de la dirección 4010BC, lo que mostrará nuestro bad boy 'Found debugger'.

```
OllyDbg - Anti-crackme.exe - [CPU - main thread, module Anti-cra]
C File View Debug Plugins Options Window Help Tools BreakPoint->
Paused                                     L E M T W H C / K B R ... S

004010A5  .  C9             LEAVE
004010A6  .  C2 1000        RETN 10
004010A9  >  B8 01000000    MOV EAX,1
004010AE  .  C9             LEAVE
004010AF  .  C2 1000        RETN 10
004010B2  $  E8 27020000    CALL <JMP.&kernel32.IsDebuggerPresen   IsDebuggerPresent
004010B7  .  84C0           TEST AL,AL
004010B9  .v 75 01          JNZ SHORT Anti-cra.004010BC
004010BB  .  C3             RETN
004010BC  >  E8 72010000    CALL Anti-cra.00401233
004010C1  $  0F31           RDTSC
004010C3  .  50             PUSH EAX
004010C4  .  33C0           XOR EAX,EAX
004010C6  .  0F31           RDTSC
004010C8  .  2B0424         SUB EAX,DWORD PTR SS:[ESP]              kernel32.7C816D4F
004010CB  .  3D 00200000    CMP EAX,2000
004010D0  .v 77 19          JA SHORT Anti-cra.004010EB
004010D2  .  6A 0A          PUSH 0A                                 Count = A (10.)
004010D4  .  68 7C304000    PUSH Anti-cra.0040307C                  Buffer = Anti-cra.0040307C
004010D9  .  68 B80B0000    PUSH 0BB8                               ControlID = BB8 (3000.)
004010DE  .  FF35 78304000  PUSH DWORD PTR DS:[403078]              hWnd = NULL
004010E4  .  E8 D1010000    CALL <JMP.&user32.GetDlgItemTextA>      GetDlgItemTextA
004010E9  .  58             POP EAX                                 kernel32.7C816D4F
004010EA  .  C3             RETN
004010EB  >  E8 43010000    CALL Anti-cra.00401233
004010F0  $  B8 02000000    MOV EAX,2
004010F5  .  83F8 03        CMP EAX,3
004010F8  .v 74 02          JE SHORT Anti-cra.004010FC
004010FA  .v 75 0E          JNZ SHORT Anti-cra.0040110A
004010FC  >  41             INC ECX
004010FD  .  58             POP EAX                                 kernel32.7C816D4F
004010FE  .v 74 48          JE SHORT Anti-cra.00401148
00401100  .  47             INC EDI                                 ntdll.7C920738
00401101  .  37             AAA
00401102  .  396B 4D        CMP DWORD PTR DS:[EBX+4D],EBP
```

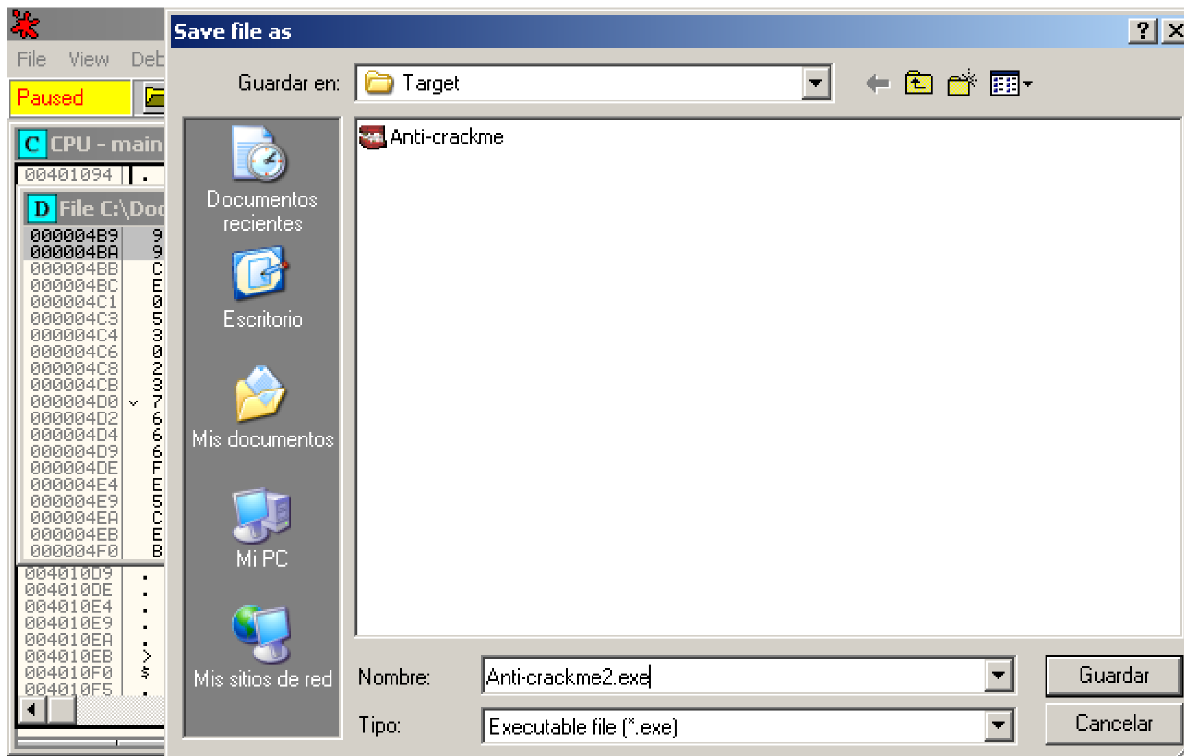
Para no entrar en el call IsDebuggerPresent existen varios plugins que nos ayudarán en alcanzar nuestro objetivo. Podemos utilizar por ejemplo el plugin OllyAdvanced. Una vez seleccionado OllyAdvanced hacemos clic en la pestaña “Anti-Debug 2” y marcamos la casilla IsDebuggerPresent. De esta forma cualquier CALL a IsDebuggerPresent devolverá “False”.



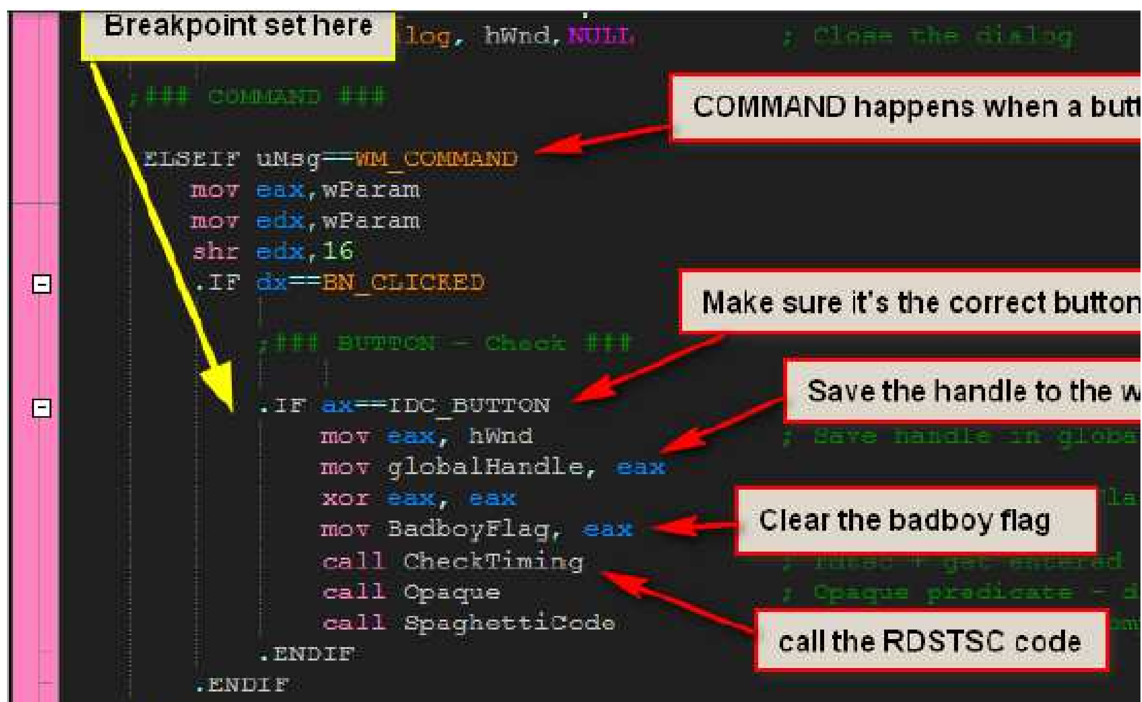
También podemos deshabilitarlo manualmente, parcheando el resultado de la comprobación de IsDebuggerPresent en la dirección 4010B9. Hacemos clic con el botón derecho sobre la instrucción JNZ y seleccionamos “Binary” -> “Fill with NOP’s”:



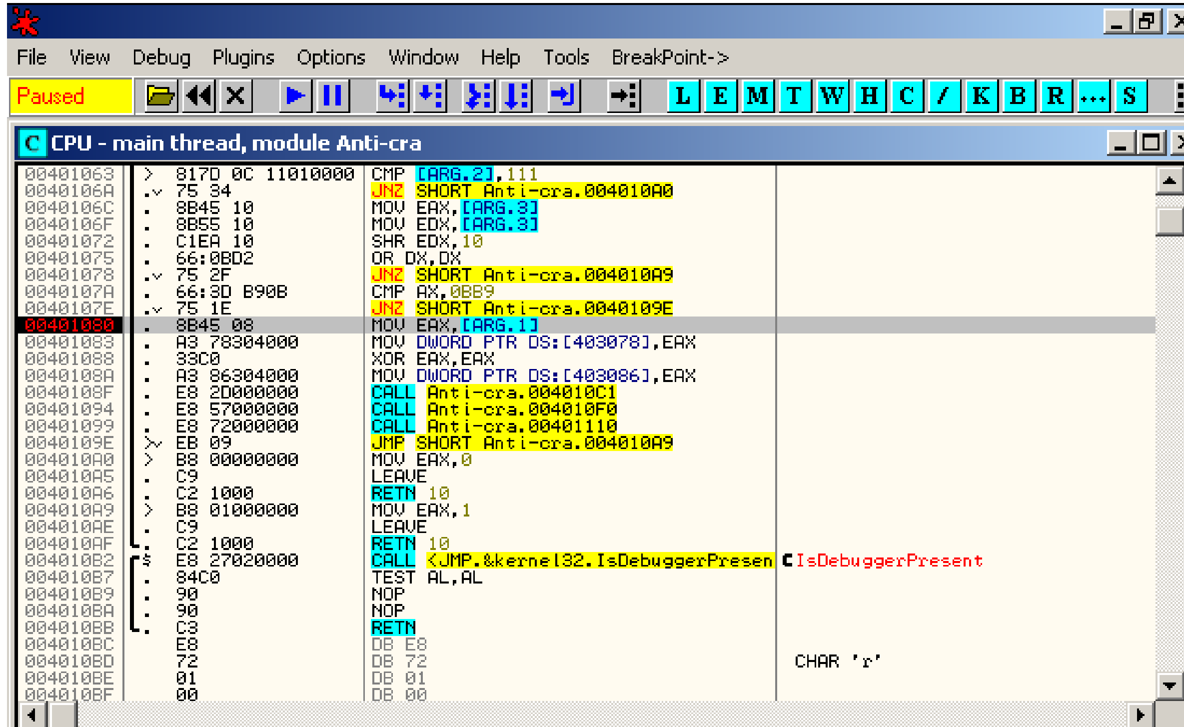
Guardamos el parche como Anti-crackme2.exe



Vamos a cargar este nuevo binario en Olly para continuar. Ponemos un Breakpoint en la dirección 401080, que coincide con la devolución de llamada principal, DlgProc, pero después del códig del mensajes INITDIALOG. Poniendo nuestro Breakpoint en este lugar nos salvará de tener que pulsar F9 varias veces cuando la aplicación se carge por primera vez, ya que la devolución de llamada será llamada cada vez que aparece el mensaje de Windows. El único mensaje que nos preocupa ahora mismo es el mensaje WM_COMMAND para cuando pulsemos el botón "Check it" y nuestro Breakpoint está justo al comienzo:



Cuando pulsamos F9 por primer vez, aparecerá la pantalla principal preguntando por el serial. Introducimos cualquier serial y hacemos clic en el botón “Check it”. Olly se detendrá en nuestro Breakpoint.



Cuando pulsamso el botón “Check it”, se envía un mensaje WM_COMMAND a través de la devolución de llamada DlgProc. Lo primero que haremos es comprobar si el ID coincide con el ID de nuestro botón y como solo hay un botón, coincidirán. Lo siguiente es guardar el controlador de la ventana en una variable global para poder acceder desde otra funciones. También resetearemos la bandera del bad boy devuelta a zero, y después haremos una llamada al método CheckTiming.

Capítulo 8

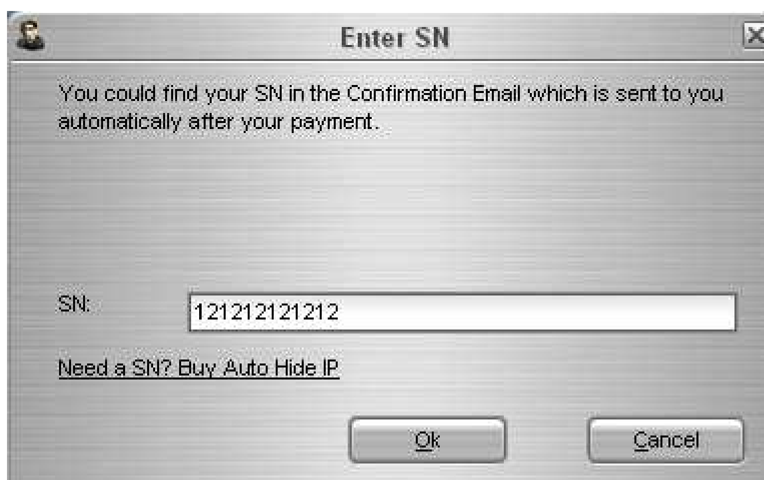
EJERCICIOS COMPLEMENTARIOS

8.1 Otra forma de poner un parche

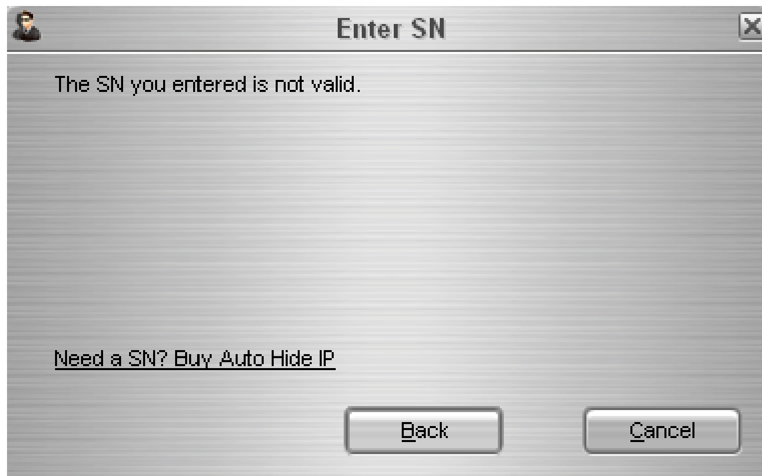
En este ejercicio vamos a utilizar un programa real que podemos descargar de http://www.autohideip.com/. Trataremos de crackear el esquema de registro de la aplicación que tiene un periode de prueba de 1 día. El nag correspondiente comprobará el serial con el servidor pero solo si se introduce un serial.

Una vez instalado el programa hacemos doble clic para ejecutarlo. Nos sale el siguiente mensaje:

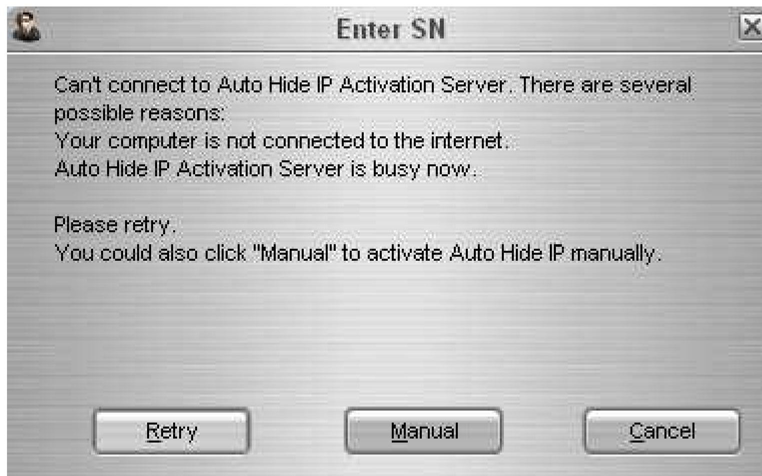
Hacemos clic en "Enter SN" e introducimos un serial cualquiera:



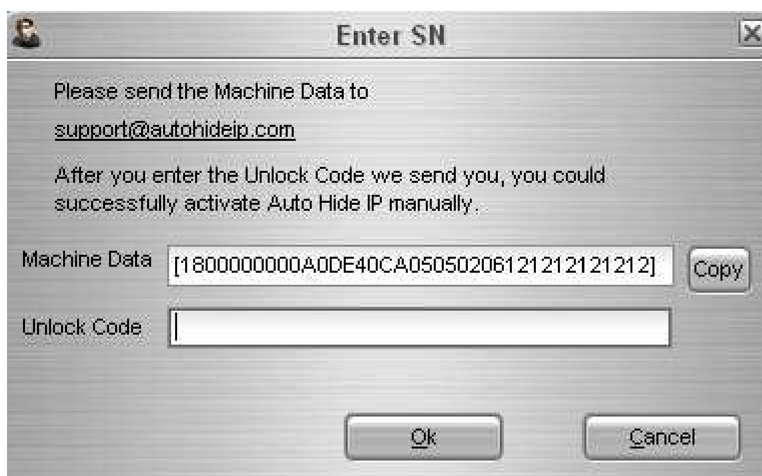
Si no nos hemos desconectado de Internet al pulsar "Ok" nos sale el siguiente mensaje:



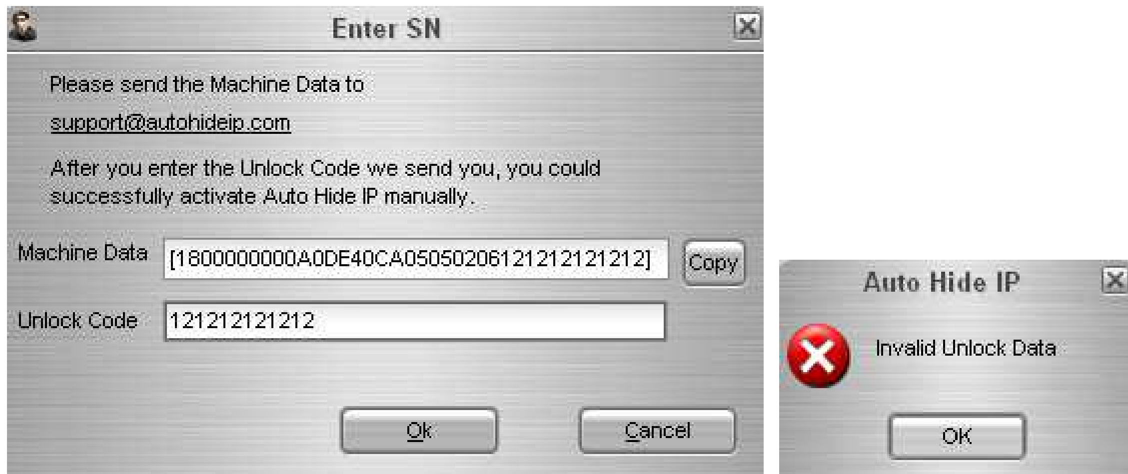
Hacemos clic en “Back” y volvemos a realizar la misma operación, pero esta vez desconectados de internet. Nos sale el siguiente mensaje:



¡ Podemos registrar la aplicación sin estar conectados a internet ! Intentemoslo haciendo clic en “Manual”.

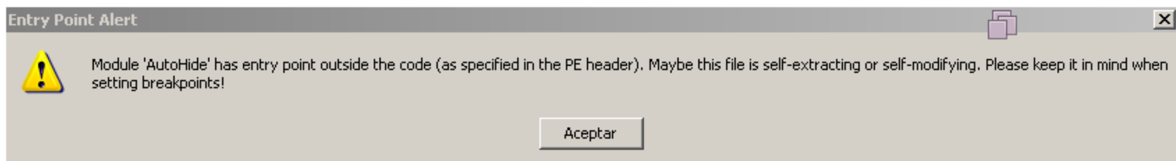


Para obtener el serial correcto debemos enviar los datos de nuestro equipo a la empresa. Probablemente utilizarán esta información para calcular el código que debemos introducir. Vamos instroducir un serial cualquiera y hacemos clic en “Ok”:

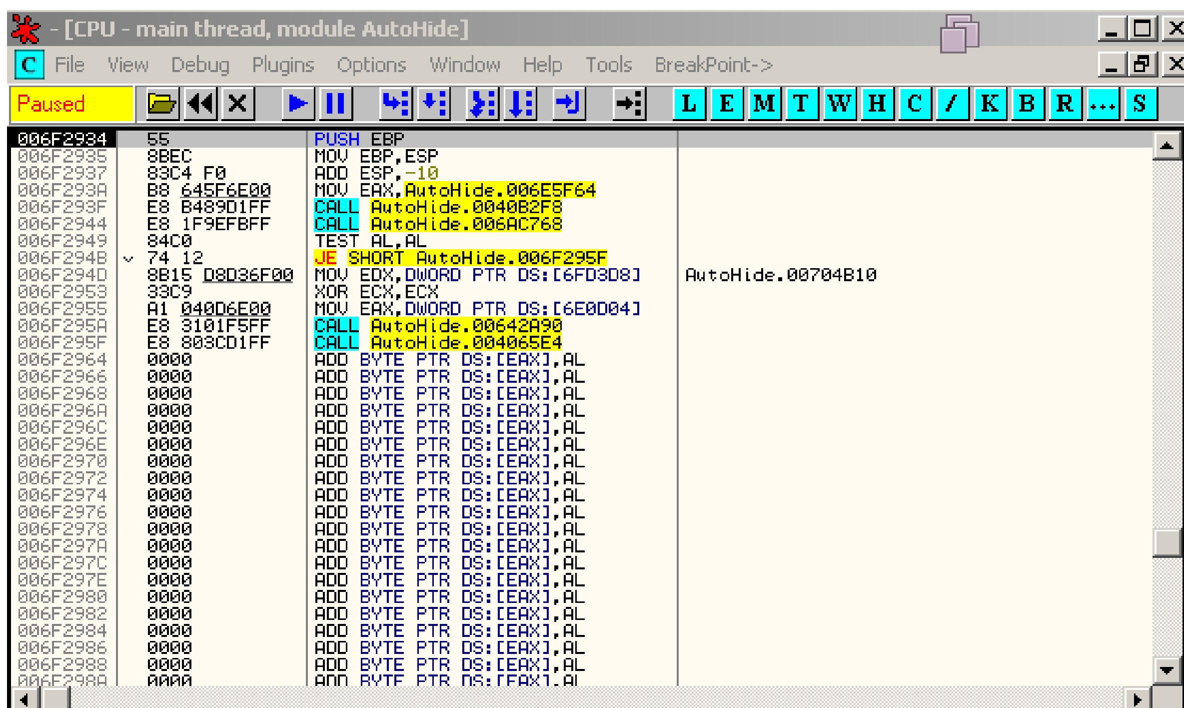


Hemos llegado hasta nuestro “bad boy” sin estar conectados a internet.

Abrimos Olly y cagamos la aplicación. Antes de cargarse nos advierte de que el punto de entrada está fuera del código.



Hacemos clic en “Aceptar” y la aplicación termina de cargarse:



A primera vista podemos concluir que no se trata de un binario “normal”. Si buscamos por cadenas de texto, intermodular calls o call stack, vemos que no obtenemos ninguna información de utilidad. Si cargamos la aplicación en ExeInfoPE veremos el por qué:

Estamos ante un programa "Borland Delphi". Para seguir analizando el binario, cargaremos el programa en DeDe:

DeDe 3.50.02 (c) 1999-2002 by DaFixer

File Dumpers Tools Options About

C:\Archivos de programa\AutoHideIP\AutoHi — Process — AutoHideIP

Classes Info | Units Info | Forms | Procedures | Project | Exports

Class Name	Unit Name	SelfPrt	DFM Offset
:01		005E9378	00000000
:02		0042BDCC	00000000
:1		0040C3F8	00000000
:11		005E9748	00000000
:2		00548968	00000000
:21		004A9FDC	00000000
:3		0068E630	00000000
:31		004AA0F8	00000000
:33		00430B9C	00000000
:35		00411B68	00000000
:41		004AA420	00000000
:45		00411B90	00000000
:55		00437100	00000000
:6		004D55EC	00000000
:7		005E83B0	00000000
:8		005E8830	00000000
:9		005E8FCC	00000000
:96		0041B8A4	00000000
::inflate_blocks_state.:1.:1		00515578	00000000
::inflate_blocks_state.:1.:2		005155F8	00000000
::inflate_codes_state.:1.:1		00515310	00000000
::inflate_codes_state.:1.:2		00515360	00000000
::internal_state.:1.:1		00515844	00000000
:inflate_blocks_state.:1		00515658	00000000
:inflate_codes_state.:1		005153B0	00000000
:internal_state.:1		00515890	00000000
:TApplication.:1		004D9944	00000000

Version:<unknown version>

Unit List (from PACKAGEINFO)

ABAccessibility
AccCtrl
AclAPI
ActiveX
ActnColorMaps
ActnCtrls
ActnList
ActnMan
ActnMenus
AnsiStrings
AxCtrls
BrkrConst
Browse4Folder
Buttons
Character
CLAboutForm
CLActivate
CLActivateForm
Classes
CLBuyNowNotifyPanel
CLConst
CLEngine
CLForm
Clipbrd
CLMain
CLMultiLang
CLMultiLangLan
CLSettings
CLSplashForm
CLTrialNotifyForm
CLUpdateForm
CLWebBanner
CnDebug
ComConst

Después de hacer clic en "Process" y unas cuantas advertencias de errores más tarde, por fin aparece la pantalla principal. No obstante si seleccionamos las pestañas 'Forms' y

'Procedures' veremos campos vacíos. Utilizando esta herramienta tampoco nos ayuda de forma significativa.

Llegado a ese punto vamos a introducir una herramienta nueva. Abrimos internet y descargamos PExplorerR6. Una vez descargado cargamos nuestra aplicación:

PE Explorer - C:\Archivos de programa\AutoHideIP\AutoHideIP.exe

File View Tools Help

HEADERS INFO

Address of Entry Point: 006F2934 Real Image Checksum: 003935ACh

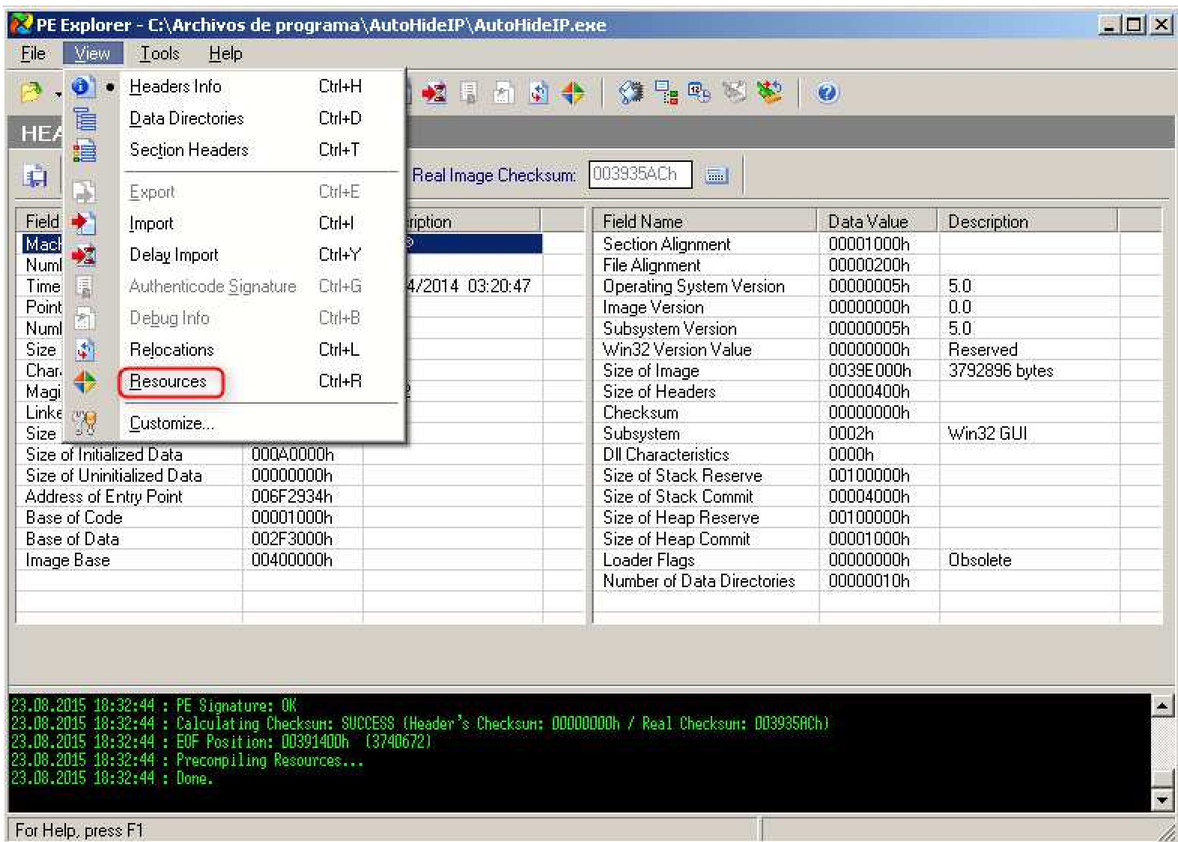
Field Name	Data Value	Description
Machine	014Ch	i386®
Number of Sections	000Ah	
Time Date Stamp	53436B0Fh	08/04/2014 03:20:47
Pointer to Symbol Table	00000000h	
Number of Symbols	00000000h	
Size of Optional Header	00E0h	
Characteristics	818Eh	
Magic	010Bh	PE32
Linker Version	1902h	2.25
Size of Code	002F1000h	
Size of Initialized Data	000A0000h	
Size of Uninitialized Data	00000000h	
Address of Entry Point	006F2934h	
Base of Code	00001000h	
Base of Data	002F3000h	
Image Base	00400000h	

Field Name	Data Value	Description
Section Alignment	00001000h	
File Alignment	00000200h	
Operating System Version	00000005h	5.0
Image Version	00000000h	0.0
Subsystem Version	00000005h	5.0
Win32 Version Value	00000000h	Reserved
Size of Image	0039E000h	3792896 bytes
Size of Headers	00000400h	
Checksum	00000000h	
Subsystem	0002h	Win32 GUI
Dll Characteristics	0000h	
Size of Stack Reserve	00100000h	
Size of Stack Commit	00004000h	
Size of Heap Reserve	00100000h	
Size of Heap Commit	00001000h	
Loader Flags	00000000h	Obsolete
Number of Data Directories	00000010h	

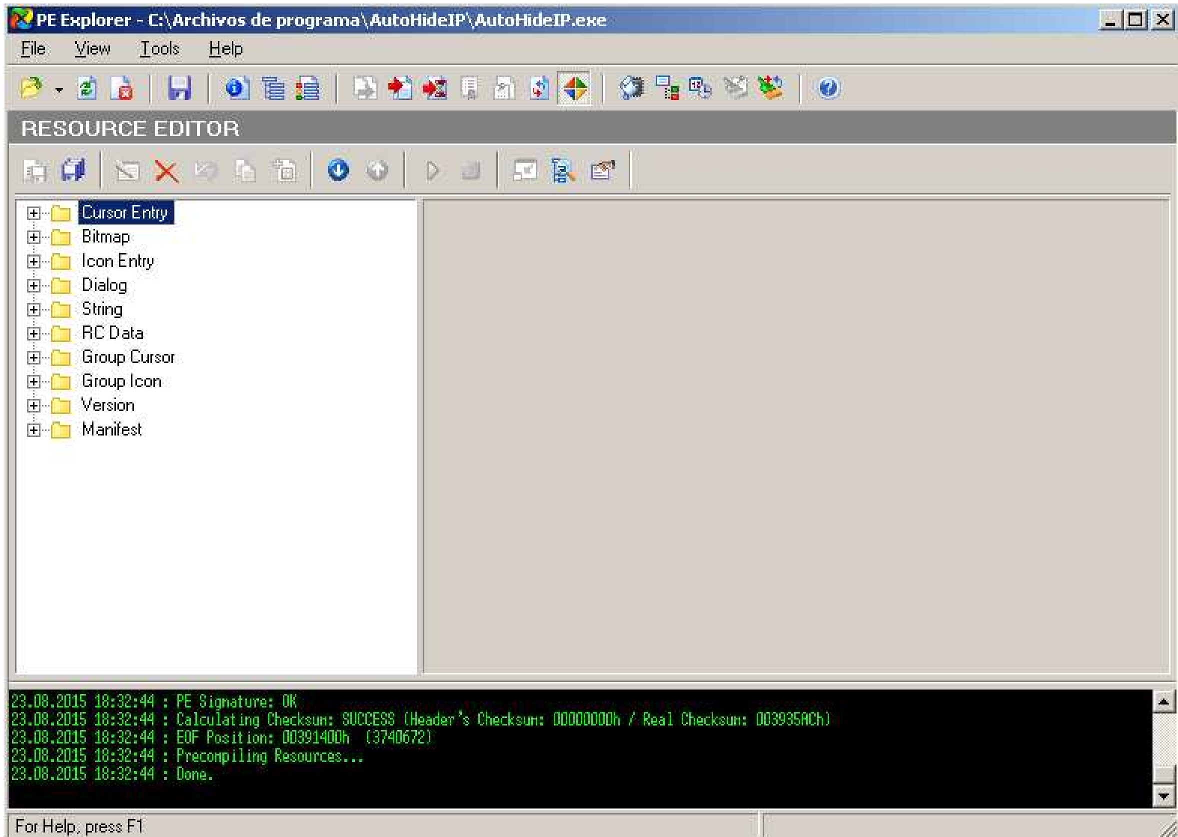
```
23.08.2015 18:32:44 : PE Signature: OK
23.08.2015 18:32:44 : Calculating Checksum: SUCCESS (Header's Checksum: 00000000h / Real Checksum: 003935ACh)
23.08.2015 18:32:44 : EOF Position: 00391400h (3740672)
23.08.2015 18:32:44 : Precompiling Resources...
23.08.2015 18:32:44 : Done.
```

For Help, press F1

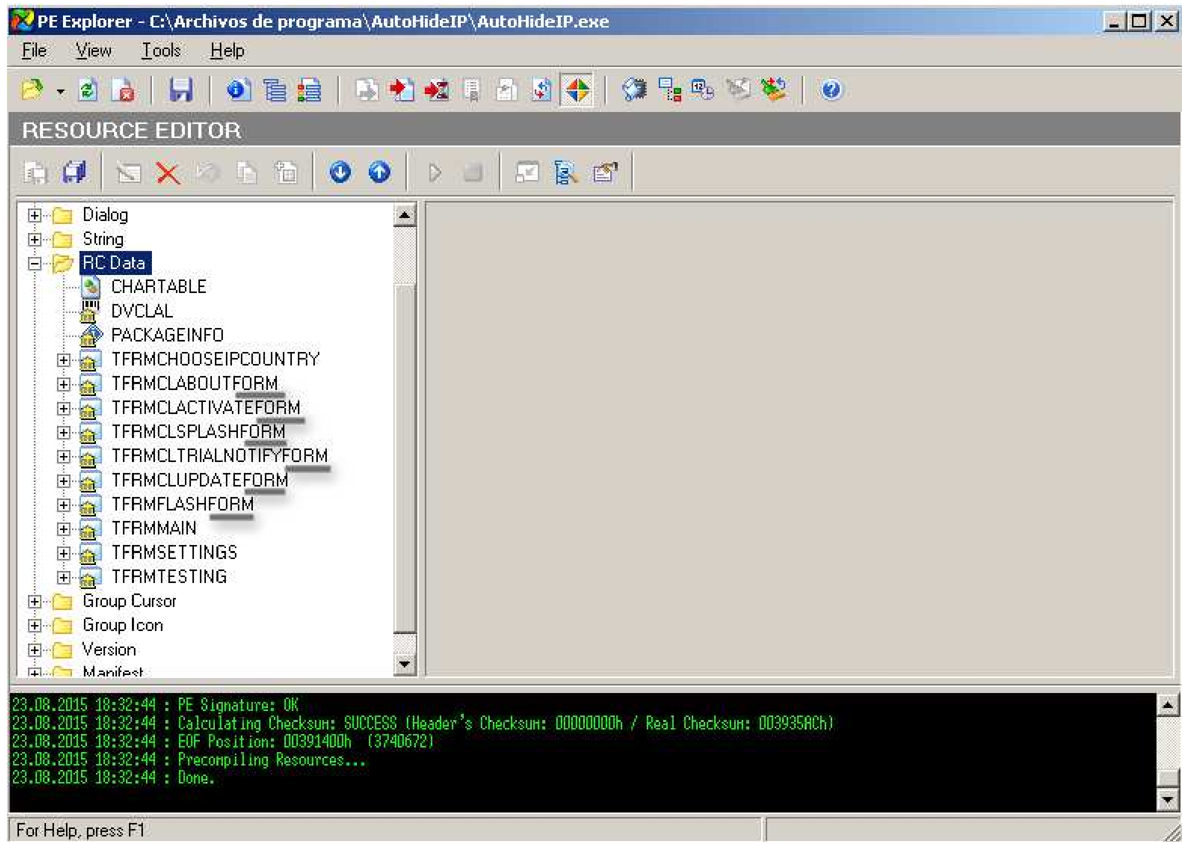
Dentro del menú seleccionamos "View" -> "Resources":



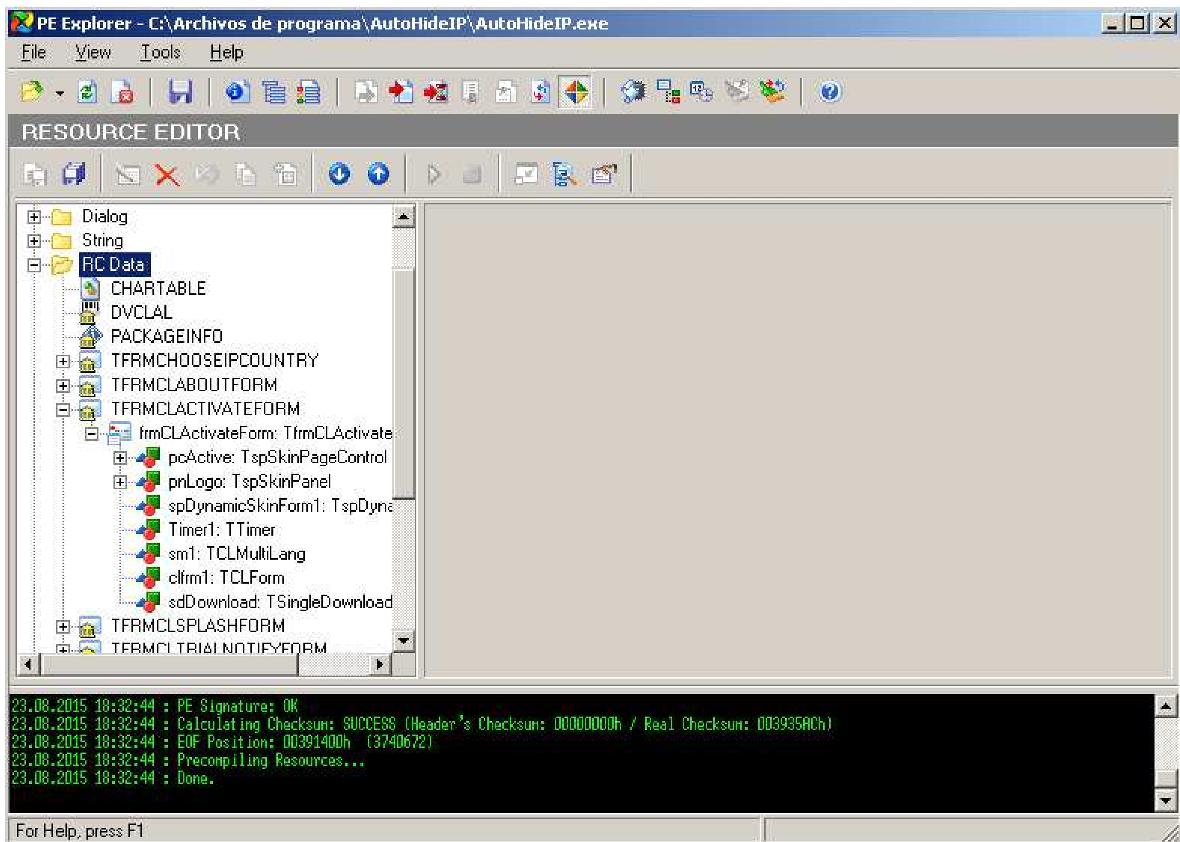
Aparece la pantalla del Resource Editor:



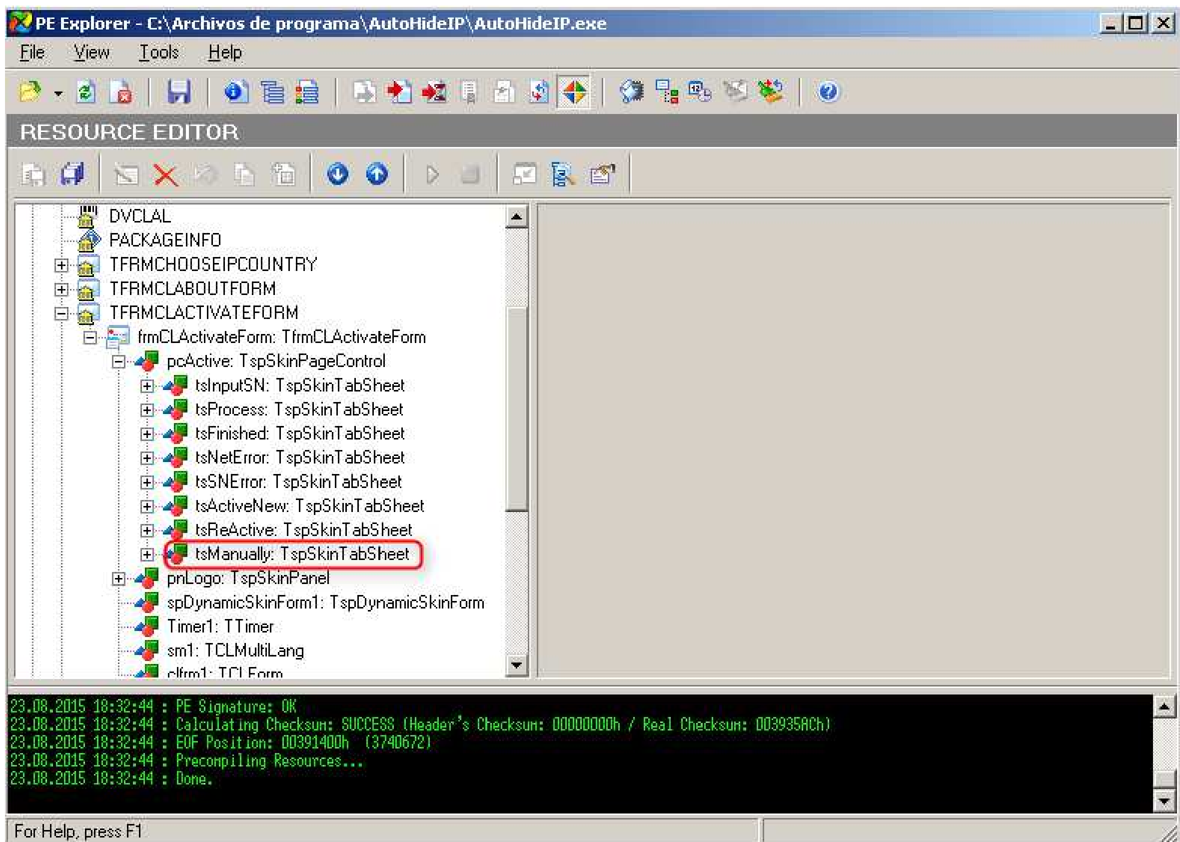
Sabemos que en las carpetas “Dialog” y “String” no vamos a encontrar nada, por lo que empezamos analizar la carpeta RC Data. Desplegamos la carpeta y aquí sí aparecen unos cuantos formularios:



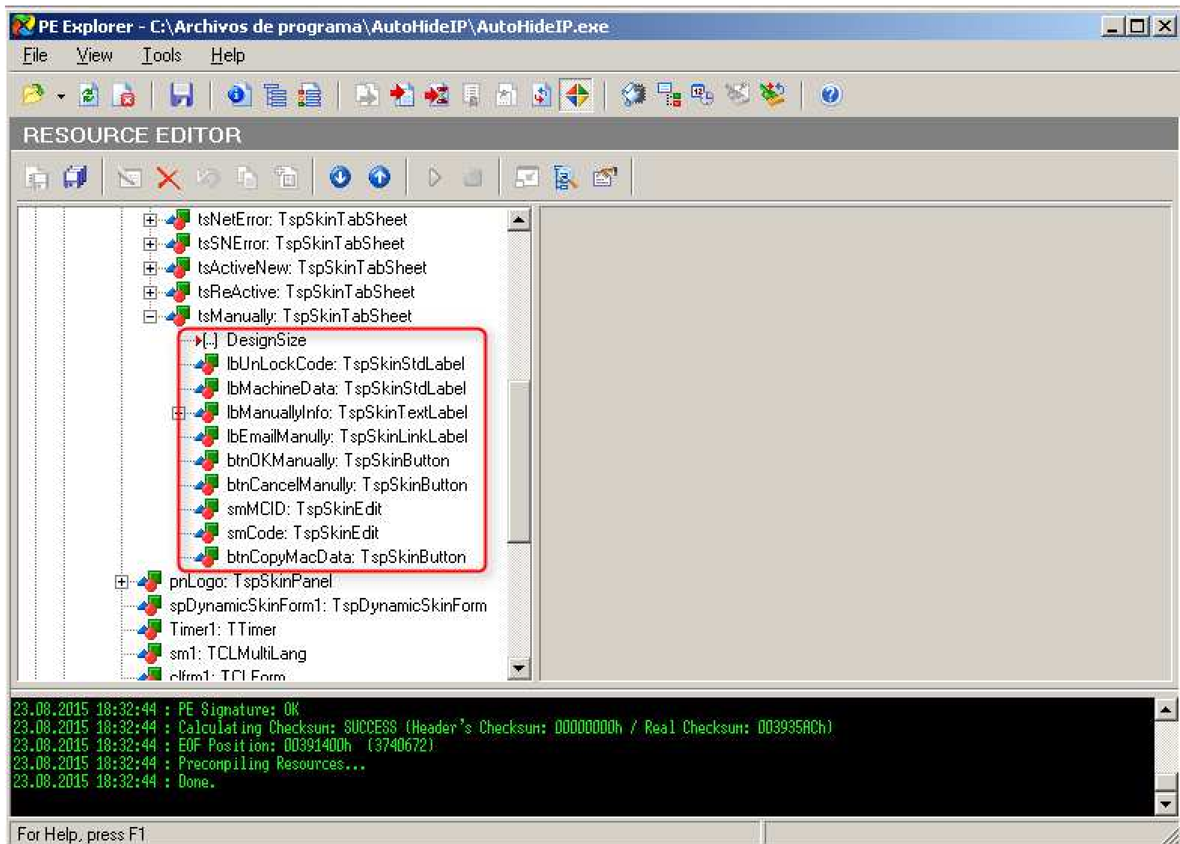
Seleccionamos “ACTIVATEFORM” y desplegamos la pestaña:



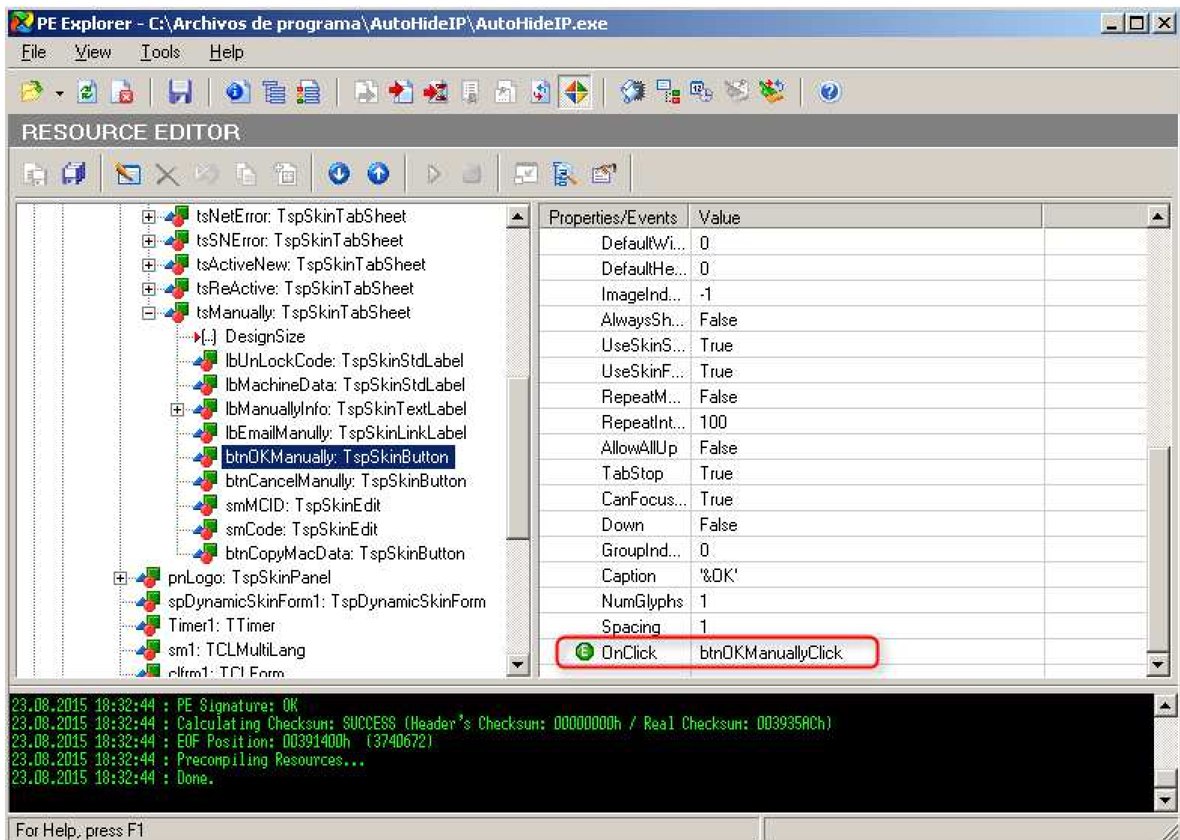
Desplegamos la pestaña “pcActive” para ver lo que esconde:



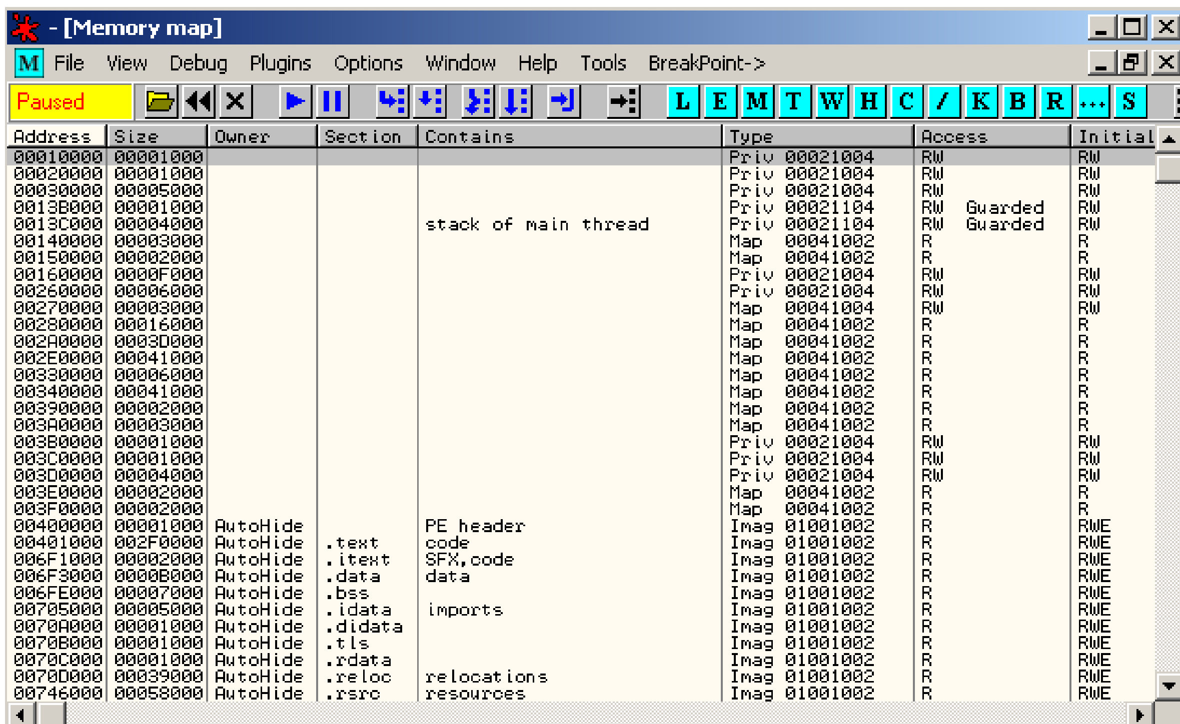
Vemos otra pestaña interesante; “tsManually” ¿Estará aquí el formulario para la activación manual del programa? Desplegamos la pestaña:



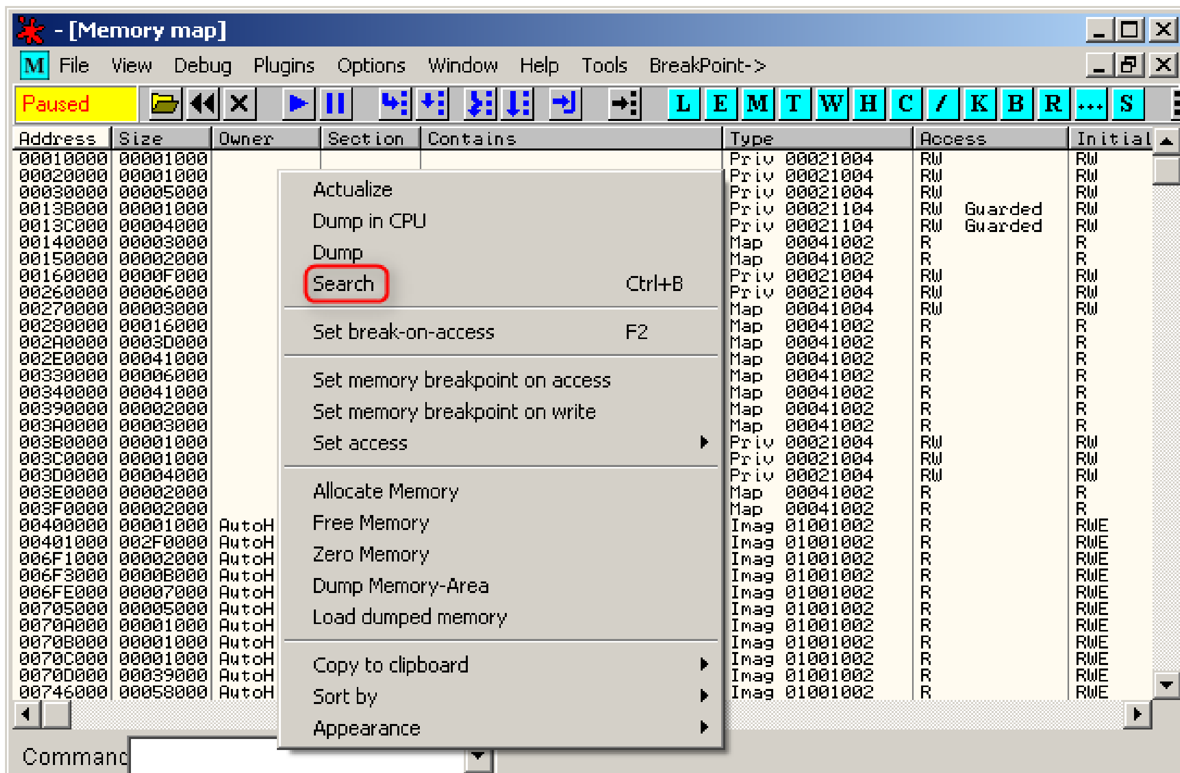
Podemos ver todos los controles del formulario para la activación manual del programa; cuatro etiquetas y dos botones. Seleccionamos el primer botón “btnOKManually” y en la parte derecha de la pantalla bajamos hasta el final:



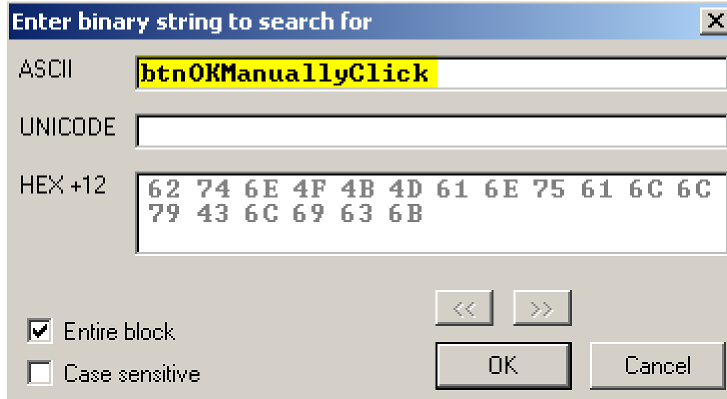
En la última línea aparece el evento "OnClick" = btn0KManuallyClick, que será llamado tampronto se haga clic en el botón "Manual". Como sabemos que Delphi utiliza esos nombres para llamar eventos, solo tendremos que buscar el nombre del evento en el programa. Para ello volvemos a Olly, cargamos el programa y abrimos la ventana de la memoria:



Hacemos clic con el botón derecho y seleccionamos “search”:



En el campo ASCII introducimos el nombre del evento:

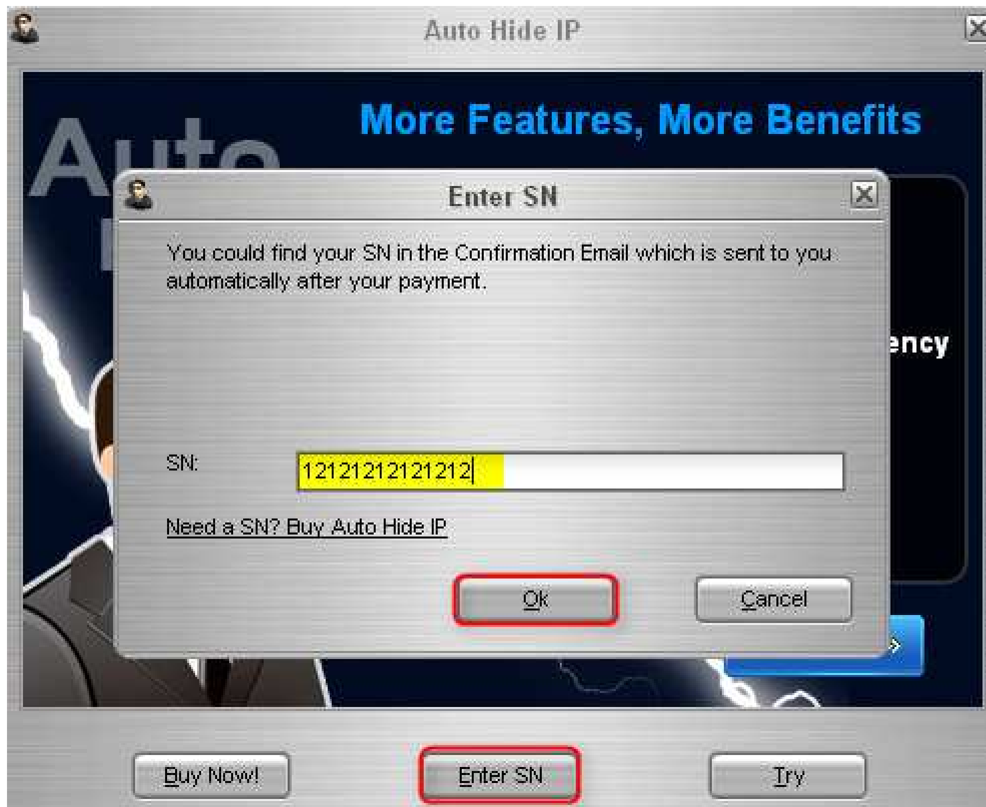


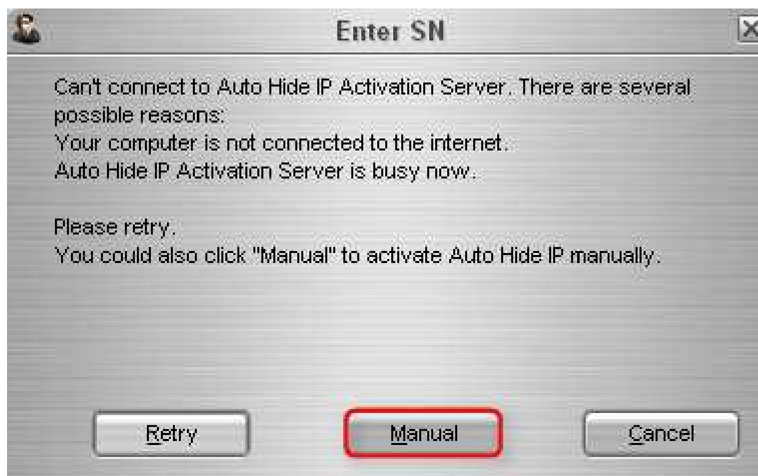
Hacemos clic en “OK”, lo que nos lleva ver el resultado en la ventana dump:

Si subimos un poco veremos la dirección del evento:

Y en little endian: 630F20. Busquemos la dirección en la ventana de desensamblaje:

Nos encontramos en el inicio del evento btnOKManuallyClick. Ponemos un punto de ruptura y ejecutamos el programa (realizaremos el análisis sin conexión a internet). Introducimos un serial y Olly se detiene en nuestro punto de ruptura:





Empezamos a ejecutar el código línea por línea pulsando F8. En la dirección 630F4F se almacena nuestro serial en EAX:

Si hacemos clic en el CALL 630F59 o en el CALL 630F62, veremos dos rutinas larguísimas que son llamadas desde más de diez sitios distintos. Para ser una comprobación del serial es bastante inusual. Esto y con el hecho de que EAX sigue almacenando nuestro serial, y de que no hay saltos alrededor de esa instrucción, son indicios de que no es aquí donde el serial va a ser comprobado. Fijémonos ahora en el CALL 630F69. Justo después hay una instrucción TEST AL,AL seguido por una instrucción JNZ.

Si pulsamos F8 una vez más veremos que nuestra cadena en EAX ha desaparecido y que AL=0, así que este salto no va a ser tomado. Para saltar cambiamos el valor de la bandera Z a cero y pulsamos F9.

Parece que hemos llegado al sitio correcto, pero si pulsamos el botón “Ok”, veremos lo siguiente:

¡ Seguimos con la versión de prueba ! Vemos que este parche no fue lo suficientemente profundo, así que tendremos que analizar el código con más detalle. Reiniciamos el programa para situarnos en el CALL 630F69.

Ahora sabemos que AL debe ser igual a uno para que salte a la dirección 630F70. Pulsamos F7 para entrar en el CALL:

```
- [CPU - main thread, module AutoHide]
C File View Debug Plugins Options Window Help Tools BreakPoint->
Paused                                          L E M T W H C / K B R ... S

0063C393      00            DB 00
0063C394  ┌$  55            PUSH EBP
0063C395  │.  8BEC          MOV EBP,ESP
0063C397  │.  53            PUSH EBX
0063C398  │.  56            PUSH ESI
0063C399  │.  57            PUSH EDI
0063C39A  │.  8BD9          MOV EBX,ECX
0063C39C  │.  8BF2          MOV ESI,EDX
0063C39E  │.  8BF8          MOV EDI,EAX
0063C3A0  │.  8BD6          MOV EDX,ESI
0063C3A2  │.  8BC7          MOV EAX,EDI
0063C3A4  │.  E8 0BFFFFFF   CALL AutoHide.0063C2B4
0063C3A9  │.  84C0          TEST AL,AL
0063C3AB  │.v 74 50         JE SHORT AutoHide.0063C3FD
0063C3AD  │.  33C0          XOR EAX,EAX
0063C3AF  │.  E8 F84A0000   CALL AutoHide.00640EAC
0063C3B4  │.  8BCE          MOV ECX,ESI
0063C3B6  │.  BA 1CC46300   MOV EDX,AutoHide.0063C41C        UNICODE "Activate.UnlockData"
0063C3BB  │.  E8 704E0000   CALL AutoHide.00641230
0063C3C0  │.  33C0          XOR EAX,EAX
0063C3C2  │.  E8 E54A0000   CALL AutoHide.00640EAC
0063C3C7  │.  8BCB          MOV ECX,EBX
0063C3C9  │.  BA 50C46300   MOV EDX,AutoHide.0063C450        UNICODE "Activate.SN"
0063C3CE  │.  E8 5D4E0000   CALL AutoHide.00641230
0063C3D3  │.  33C0          XOR EAX,EAX
0063C3D5  │.  E8 D24A0000   CALL AutoHide.00640EAC
0063C3DA  │.  33C9          XOR ECX,ECX
0063C3DC  │.  BA 74C46300   MOV EDX,AutoHide.0063C474        UNICODE "Activate.NeedReactivated"
0063C3E1  │.  E8 F2500000   CALL AutoHide.006414D8
0063C3E6  │.  33C0          XOR EAX,EAX
0063C3E8  │.  E8 BF4A0000   CALL AutoHide.00640EAC
0063C3ED  │.  33C9          XOR ECX,ECX
0063C3EF  │.  BA B4C46300   MOV EDX,AutoHide.0063C4B4        UNICODE "Activate.Expired"
0063C3F4  │.  E8 DF500000   CALL AutoHide.006414D8
0063C3F9  │.  B3 01         MOV BL,1
0063C3FB  │.v EB 02         JMP SHORT AutoHide.0063C3FF
0063C3FD  │>  33DB          XOR EBX,EBX
0063C3FF  │>  8BC7          MOV EAX,EDI
0063C401  │.  E8 86030000   CALL AutoHide.0063C78C
0063C406  │.  8BC3          MOV EAX,EBX
0063C408  │.  5F            POP EDI                          AutoHide.00630F6E
0063C409  │.  5E            POP ESI                          AutoHide.00630F6E
0063C40A  │.  5B            POP EBX                          AutoHide.00630F6E
0063C40B  │.  5D            POP EBP                          AutoHide.00630F6E
0063C40C  └.  C2 0800       RETN 8
0063C40F      00            DB 00
0063C410      B0            DB B0
0063C411      04            DB 04
```

Esta rutina es llamada solo por dos lugares.

Si ejecutamos el código línea por línea pulsando F8 hasta llegar al salto en 63C3AB veremos que vamos a tomar el salto puesto que AL=0. Saltará a la dirección 63C3FD, donde espera la instrucción XOR EBX, EBX, lo que pone EBX a cero. Es importante tener en cuenta esto porque más abajo en la dirección 63C406, EBX es movido a EAX, lo que volvería poner AL=0, cosa que no queremos.

```
- [CPU - main thread, module AutoHide]
C File View Debug Plugins Options Window Help Tools BreakPoint->
Paused                                          L E M T W H C / K B R ... S

0063C393     00            DB 00
0063C394   $ 55            PUSH EBP
0063C395   . 8BEC          MOV EBP,ESP
0063C397   . 53            PUSH EBX
0063C398   . 56            PUSH ESI
0063C399   . 57            PUSH EDI
0063C39A   . 8BD9          MOV EBX,ECX
0063C39C   . 8BF2          MOV ESI,EDX
0063C39E   . 8BF8          MOV EDI,EAX
0063C3A0   . 8BD6          MOV EDX,ESI
0063C3A2   . 8BC7          MOV EAX,EDI
0063C3A4   . E8 0BFFFFFF   CALL AutoHide.0063C2B4
0063C3A9   . 84C0          TEST AL,AL
0063C3AB   . 74 50         JE SHORT AutoHide.0063C3FD
0063C3AD   . 33C0          XOR EAX,EAX
0063C3AF   . E8 F84A0000   CALL AutoHide.00640EAC
0063C3B4   . 8BCE          MOV ECX,ESI
0063C3B6   . BA 1CC46300   MOV EDX,AutoHide.0063C41C      UNICODE "Activate.UnlockData"
0063C3BB   . E8 704E0000   CALL AutoHide.00641230
0063C3C0   . 33C0          XOR EAX,EAX
0063C3C2   . E8 E54A0000   CALL AutoHide.00640EAC
0063C3C7   . 8BCB          MOV ECX,EBX
0063C3C9   . BA 50C46300   MOV EDX,AutoHide.0063C450      UNICODE "Activate.SN"
0063C3CE   . E8 5D4E0000   CALL AutoHide.00641230
0063C3D3   . 33C0          XOR EAX,EAX
0063C3D5   . E8 D24A0000   CALL AutoHide.00640EAC
0063C3DA   . 33C9          XOR ECX,ECX
0063C3DC   . BA 74C46300   MOV EDX,AutoHide.0063C474      UNICODE "Activate.NeedReactivated"
0063C3E1   . E8 F2500000   CALL AutoHide.006414D8
0063C3E6   . 33C0          XOR EAX,EAX
0063C3E8   . E8 BF4A0000   CALL AutoHide.00640EAC
0063C3ED   . 33C9          XOR ECX,ECX
0063C3EF   . BA B4C46300   MOV EDX,AutoHide.0063C4B4      UNICODE "Activate.Expired"
0063C3F4   . E8 DF500000   CALL AutoHide.006414D8
0063C3F9   . B3 01         MOV BL,1
0063C3FB   . EB 02         JMP SHORT AutoHide.0063C3FF
0063C3FD   > 33DB          XOR EBX,EBX
0063C3FF   > 8BC7          MOV EAX,EDI
0063C401   . E8 86030000   CALL AutoHide.0063C78C
0063C406   . 8BC3          MOV EAX,EBX
0063C408   . 5F            POP EDI                        AutoHide.00630F6E
0063C409   . 5E            POP ESI                        AutoHide.00630F6E
0063C40A   . 5B            POP EBX                        AutoHide.00630F6E
0063C40B   . 5D            POP EBP                        AutoHide.00630F6E
0063C40C   . C2 0800       RETN 8
0063C40F     00            DB 00
0063C410     B0            DB B0
0063C411     04            DB 04
```

Si cambiamos el valor de la bandera Z a uno, y no saltamos veremos como al llegar a la dirección 63C3F9, BL va a ser cargado con el valor 1 y si BL vale 1, como va a ser almacenado en EAX, AEX va a valer 1 después de pasar por la instrucción RETN 8 en 63C40C. Si ejecutamos ahora el programa veremos que este parche tampoco es suficiente, por lo que nos veremos obligados a profundizar más en el estudio del código.

Situemonos en el CALL de la dirección 63C3A4 y pulsamos F7 para entrar.

```
- [CPU - main thread, module AutoHide]
C File View Debug Plugins Options Window Help Tools BreakPoint->
Paused                L E M T W H C / K B R ... S

0063C2B4  $ 55          PUSH EBP
0063C2B5  . 8BEC        MOV EBP,ESP
0063C2B7  . 6A 00       PUSH 0
0063C2B9  . 6A 00       PUSH 0
0063C2BB  . 6A 00       PUSH 0
0063C2BD  . 53          PUSH EBX
0063C2BE  . 8955 FC     MOV [LOCAL.1],EDX
0063C2C1  . 8B45 FC     MOV EAX,[LOCAL.1]
0063C2C4  . E8 7FB2DCFF CALL AutoHide.00407548
0063C2C9  . 33C0        XOR EAX,EAX
0063C2CB  . 55          PUSH EBP
0063C2CC  . 68 2CC36300 PUSH AutoHide.0063C32C
0063C2D1  . 64:FF30     PUSH DWORD PTR FS:[EAX]
0063C2D4  . 64:8920     MOV DWORD PTR FS:[EAX],ESP
0063C2D7  . 33C0        XOR EAX,EAX
0063C2D9  . E8 CE4B0000 CALL AutoHide.00640EAC
0063C2DE  . 8D4D F8     LEA ECX,[LOCAL.2]
0063C2E1  . BA 48C36300 MOV EDX,AutoHide.0063C348      UNICODE "Activate.KEY_M"
0063C2E6  . E8 254D0000 CALL AutoHide.00641010
0063C2EB  . 8B45 F8     MOV EAX,[LOCAL.2]
0063C2EE  . 50          PUSH EAX
0063C2EF  . 33C0        XOR EAX,EAX
0063C2F1  . E8 B64B0000 CALL AutoHide.00640EAC
0063C2F6  . 8D4D F4     LEA ECX,[LOCAL.3]
0063C2F9  . BA 74C36300 MOV EDX,AutoHide.0063C374      UNICODE "Activate.KEY_E"
0063C2FE  . E8 0D4D0000 CALL AutoHide.00641010
0063C303  . 8B55 F4     MOV EDX,[LOCAL.3]
0063C306  . 8B45 FC     MOV EAX,[LOCAL.1]
0063C309  . 59          POP ECX                        AutoHide.0063C3A9
0063C30A  . E8 9995FFFF CALL AutoHide.006358A8
0063C30F  . 8BD8        MOV EBX,EAX
0063C311  . 33C0        XOR EAX,EAX
0063C313  . 5A          POP EDX                        AutoHide.0063C3A9
0063C314  . 59          POP ECX                        AutoHide.0063C3A9
0063C315  . 59          POP ECX                        AutoHide.0063C3A9
0063C316  . 64:8910     MOV DWORD PTR FS:[EAX],EDX
0063C319  . 68 33C36300 PUSH AutoHide.0063C333
0063C31E  > 8D45 F4     LEA EAX,[LOCAL.3]
0063C321  . BA 03000000 MOV EDX,3
0063C326  . E8 2DB2DCFF CALL AutoHide.00407558
0063C32B  . C3          RETN
0063C32C  .^ E9 E39BDCFF JMP AutoHide.00405F14
0063C331  .^ EB EB      JMP SHORT AutoHide.0063C31E
0063C333  . 8BC3        MOV EAX,EBX
0063C335  . 5B          POP EBX                        AutoHide.0063C3A9
0063C336  . 8BE5        MOV ESP,EBP
0063C338  . 5D          POP EBP                        AutoHide.0063C3A9
0063C339  . C3          RETN
```

En esta rutina no hay ninguna combinación de comparar/saltar, pero esto no significa que estemos en el sitio equivocado. Echemos un vistazo a la última parte de esta rutina. Recordemos que que AL tiene que ser igual a uno cuando lleguemos a la instrucción RETN en la dirección 63C32B (ya que el PUSH de 63C319 hace que el RETN solo salte a la dirección 63C333).

Si nos fijamos en la dirección 63C333 vemos que la instrucción MOV mueve EBX dentro de EAX, así que antes de la instrucción PUSH en 63C319, BL tiene que ser uno. En la dirección 63C30F, EAX se mueve hacia EBX, por lo que en el CALL de 63C30A, EAX tiene que valer uno. Vallamos pues un paso más allá y entremos en ese CALL de la dirección 63C30A pulsando F7.

Como vemos esta rutina se parece bastante a la anterior por lo que haremos lo mismo. Es decir entramos en la rutina del CALL 6358F8 y nos aseguramos de que el valor de EAX sea igual a uno.

- [CPU - main thread, module AutoHide]

```
00635938  55             PUSH EBP
00635939  8BEC           MOV EBP,ESP
0063593B  83C4 EC        ADD ESP,-14
0063593E  53             PUSH EBX
0063593F  56             PUSH ESI
00635940  57             PUSH EDI
00635941  33DB           XOR EBX,EBX
00635943  895D F0        MOV [LOCAL.4],EBX
00635946  895D EC        MOV [LOCAL.5],EBX
00635949  894D F4        MOV [LOCAL.3],ECX
0063594C  8955 F8        MOV [LOCAL.2],EDX
0063594F  8945 FC        MOV [LOCAL.1],EAX
00635952  8B45 FC        MOV EAX,[LOCAL.1]
00635955  E8 EE1BDDFF    CALL AutoHide.00407548
0063595A  8B45 F8        MOV EAX,[LOCAL.2]
0063595D  E8 E61BDDFF    CALL AutoHide.00407548
00635962  8B45 F4        MOV EAX,[LOCAL.3]
00635965  E8 DE1BDDFF    CALL AutoHide.00407548
0063596A  8B45 08        MOV EAX,[ARG.1]
0063596D  E8 D61BDDFF    CALL AutoHide.00407548
00635972  33C0           XOR EAX,EAX
00635974  55             PUSH EBP
00635975  68 4D5A6300    PUSH AutoHide.00635A4D
0063597A  64:FF30        PUSH DWORD PTR FS:[EAX]
0063597D  64:8920        MOV DWORD PTR FS:[EAX],ESP
00635980  8D45 F0        LEA EAX,[LOCAL.4]
00635983  50             PUSH EAX
00635984  8B4D 08        MOV ECX,[ARG.1]
00635987  8B55 F4        MOV EDX,[LOCAL.3]
0063598A  8B45 FC        MOV EAX,[LOCAL.1]
0063598D  E8 EAFDFFFF    CALL AutoHide.0063577C
00635992  8B45 F0        MOV EAX,[LOCAL.4]                 AutoHide.0063C3A9
00635995  85C0           TEST EAX,EAX
00635997  74 16          JE SHORT AutoHide.006359AF
00635999  8BD0           MOV EDX,EAX
0063599B  83EA 0A        SUB EDX,0A
0063599E  66:833A 02     CMP WORD PTR DS:[EDX],2
006359A2  74 0B          JE SHORT AutoHide.006359AF
006359A4  8D45 F0        LEA EAX,[LOCAL.4]
006359A7  8B55 F0        MOV EDX,[LOCAL.4]                 AutoHide.0063C3A9
006359AA  E8 1910DDFF    CALL AutoHide.004069C8
006359AF  8BD8           MOV EBX,EAX
006359B1  85DB           TEST EBX,EBX
006359B3  74 05          JE SHORT AutoHide.006359BA
006359B5  83EB 04        SUB EBX,4
006359B8  8B1B           MOV EBX,DWORD PTR DS:[EBX]
006359BA  8B45 F8        MOV EAX,[LOCAL.2]
006359BD  85C0           TEST EAX,EAX
```

Vemos que se trata de una rutina bastante larga con muchos saltos… Pero a nosotros solo nos interesa salir de esta rutina con EAX=1. Bajamos por lo tanto hasta el final de la rutina.

```
- [CPU - main thread, module AutoHide]
File  View  Debug  Plugins  Options  Window  Help  Tools  BreakPoint->
Paused

006359E9  .  8BD0          MOV EDX,EAX
006359EB  .  83EA 0A       SUB EDX,0A
006359EE  .  66:833A 02    CMP WORD PTR DS:[EDX],2
006359F2  .v 74 0B         JE SHORT AutoHide.006359FF
006359F4  .  8D45 F8       LEA EAX,[LOCAL.2]
006359F7  .  8B55 F8       MOV EDX,[LOCAL.2]
006359FA  .  E8 C90FDDFF   CALL AutoHide.004069C8
006359FF  >  8BF8          MOV EDI,EAX
00635A01  .  85FF          TEST EDI,EDI
00635A03  .v 74 05         JE SHORT AutoHide.00635A0A
00635A05  .  83EF 04       SUB EDI,4
00635A08  .  8B3F          MOV EDI,DWORD PTR DS:[EDI]      AutoHide.00638238
00635A0A  >  8D45 EC       LEA EAX,[LOCAL.5]
00635A0D  .  50            PUSH EAX
00635A0E  .  8BD3          MOV EDX,EBX
00635A10  .  2BD6          SUB EDX,ESI
00635A12  .  42            INC EDX
00635A13  .  8BCF          MOV ECX,EDI
00635A15  .  8B45 F0       MOV EAX,[LOCAL.4]               AutoHide.0063C3A9
00635A18  .  E8 9322DDFF   CALL AutoHide.00407CB0
00635A1D  .  8B55 F8       MOV EDX,[LOCAL.2]
00635A20  .  8B45 EC       MOV EAX,[LOCAL.5]               AutoHide.0063C30F
00635A23  .  E8 B8F1DDFF   CALL AutoHide.00414BE0
00635A28  .  8BD8          MOV EBX,EAX
00635A2A  .  33C0          XOR EAX,EAX
00635A2C  .  5A            POP EDX                         AutoHide.006358FD
00635A2D  .  59            POP ECX                         AutoHide.006358FD
00635A2E  .  59            POP ECX                         AutoHide.006358FD
00635A2F  .  64:8910       MOV DWORD PTR FS:[EAX],EDX
00635A32  .  68 545A6300   PUSH AutoHide.00635A54
00635A37  >  8D45 EC       LEA EAX,[LOCAL.5]
00635A3A  .  BA 05000000   MOV EDX,5
00635A3F  .  E8 141BDDFF   CALL AutoHide.00407558
00635A44  .  8D45 08       LEA EAX,[ARG.1]
00635A47  .  E8 041BDDFF   CALL AutoHide.00407550
00635A4C  L. C3            RETN
00635A4D  .^ E9 C204DDFF   JMP AutoHide.00405F14
00635A52  .^ EB E3         JMP SHORT AutoHide.00635A37
00635A54  .  8BC3          MOV EAX,EBX
00635A56  .  5F            POP EDI                         AutoHide.006358FD
00635A57  .  5E            POP ESI                         AutoHide.006358FD
00635A58  .  5B            POP EBX                         AutoHide.006358FD
00635A59  .  8BE5          MOV ESP,EBP
00635A5B  .  5D            POP EBP                         AutoHide.006358FD
00635A5C  .  C2 0400       RETN 4
00635A5F     90            DB 90
00635A60  .  B85A6300      DD AutoHide.00635AB8
00635A64     00            DB 00
```

A primera vista esta rutina acaba de la misma forma que las dos anteriores, sin embargo hay una diferencia crucial y es que si entramos en el CALL de la dirección 635A23 veremos como estamos entrando en una rutina que es llamada por más de 70 sitios (se utiliza para decidir que objetos van a ser cargados). Si hacemos algún cambio aquí no solo se verá afectado la comprobación del serial sino también todo lo demás. Así que no entraremos en este CALL de la dirección 635A23. Hemos llegado a lo más profundo que se podia llegar, y es por ello que decidimos a realizar el parche en este lugar.

Recordamos que necesitamos que el valor de EAX sea igual a uno. Para ello cambiamos la instrucción del CALL en 635A23 por MOV EAX, 1:

```
- [CPU - main thread, module AutoHide]
C File View Debug Plugins Options Window Help Tools BreakPoint->
Paused                                      L E M T W H C / K B R ... S

006359E9  .   8BD0          MOV EDX,EAX
006359EB  .   83EA 0A       SUB EDX,0A
006359EE  .   66:833A 02    CMP WORD PTR DS:[EDX],2
006359F2  .v  74 0B         JE SHORT AutoHide.006359FF
006359F4  .   8D45 F8       LEA EAX,[LOCAL.2]
006359F7  .   8B55 F8       MOV EDX,[LOCAL.2]
006359FA  .   E8 C90FDDFF   CALL AutoHide.004069C8
006359FF  >   8BF8          MOV EDI,EAX
00635A01  .   85FF          TEST EDI,EDI
00635A03  .v  74 05         JE SHORT AutoHide.00635A0A
00635A05
00635A08                                                  Hide.00638238
00635A0A
00635A0D
00635A0E
00635A10
00635A12
00635A13
00635A15                                                  Hide.0063C3A9
00635A18
00635A1D
00635A20                                                  Hide.0063C30F
00635A23  .   E8 B8F1DDFF   CALL AutoHide.00414BE0
00635A28  .   8BD8          MOV EBX,EAX
00635A2A  .   33C0          XOR EAX,EAX
00635A2C  .   5A            POP EDX                       AutoHide.006358FD
00635A2D  .   59            POP ECX                       AutoHide.006358FD
00635A2E  .   59            POP ECX                       AutoHide.006358FD
00635A2F  .   64:8910       MOV DWORD PTR FS:[EAX],EDX
00635A32  .   68 545A6300   PUSH AutoHide.00635A54
00635A37  >   8D45 EC       LEA EAX,[LOCAL.5]
00635A3A  .   BA 05000000   MOV EDX,5
00635A3F  .   E8 141BDDFF   CALL AutoHide.00407558
00635A44  .   8D45 08       LEA EAX,[ARG.1]
00635A47  .   E8 041BDDFF   CALL AutoHide.00407550
00635A4C  .   C3            RETN
00635A4D  .^  E9 C204DDFF   JMP AutoHide.00405F14
00635A52  .^  EB E3         JMP SHORT AutoHide.00635A37
00635A54  .   8BC3          MOV EAX,EBX
00635A56  .   5F            POP EDI                       AutoHide.006358FD
00635A57  .   5E            POP ESI                       AutoHide.006358FD
00635A58  .   5B            POP EBX                       AutoHide.006358FD
00635A59  .   8BE5          MOV ESP,EBP
00635A5B  .   5D            POP EBP                       AutoHide.006358FD
00635A5C  .   C2 0400       RETN 4
00635A5F      90            DB 90
00635A60  .   B85A6300      DD AutoHide.00635AB8
00635A64      00            DB 00

Assemble at 00635A23
MOV EAX, 1
[x] Fill with NOP's        Assemble   Cancel
```

- [CPU - main thread, module AutoHide]

File View Debug Plugins Options Window Help Tools BreakPoint->

Paused

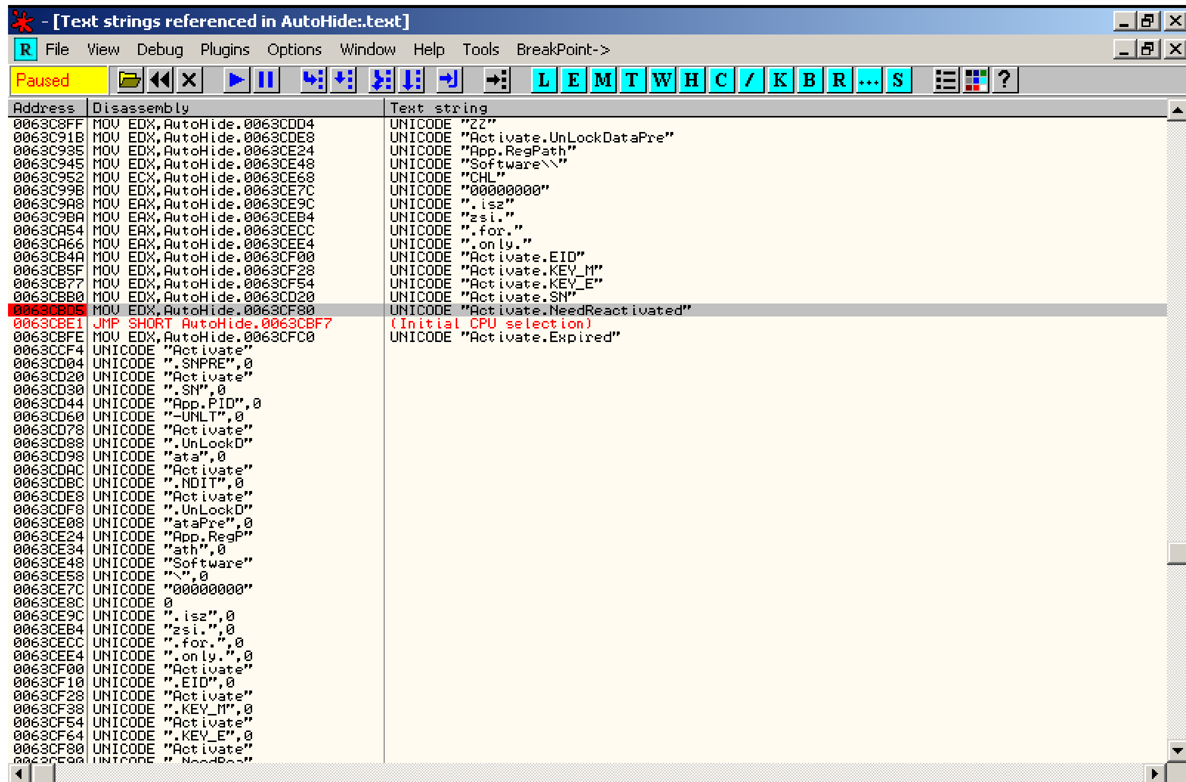
```
006359E9   .   8BD0          MOV EDX,EAX
006359EB   .   83EA 0A       SUB EDX,0A
006359EE   .   66:833A 02    CMP WORD PTR DS:[EDX],2
006359F2   .v  74 0B         JE SHORT AutoHide.006359FF
006359F4   .   8D45 F8       LEA EAX,[LOCAL.2]
006359F7   .   8B55 F8       MOV EDX,[LOCAL.2]
006359FA   .   E8 C90FDDFF   CALL AutoHide.004069C8
006359FF   >   8BF8          MOV EDI,EAX
00635A01   .   85FF          TEST EDI,EDI
00635A03   .v  74 05         JE SHORT AutoHide.00635A0A
00635A05   .   83EF 04       SUB EDI,4
00635A08   .   8B3F          MOV EDI,DWORD PTR DS:[EDI]           AutoHide.00638238
00635A0A   >   8D45 EC       LEA EAX,[LOCAL.5]
00635A0D   .   50            PUSH EAX
00635A0E   .   8BD3          MOV EDX,EBX
00635A10   .   2BD6          SUB EDX,ESI
00635A12   .   42            INC EDX
00635A13   .   8BCF          MOV ECX,EDI
00635A15   .   8B45 F0       MOV EAX,[LOCAL.4]                    AutoHide.0063C3A9
00635A18   .   E8 9322DDFF   CALL AutoHide.00407CB0
00635A1D   .   8B55 F8       MOV EDX,[LOCAL.2]
00635A20   .   8B45 EC       MOV EAX,[LOCAL.5]                    AutoHide.0063C30F
00635A23       B8 01000000   MOV EAX,1
00635A28   .   8BD8          MOV EBX,EAX
00635A2A   .   33C0          XOR EAX,EAX
00635A2C   .   5A            POP EDX                              AutoHide.006358FD
00635A2D   .   59            POP ECX                              AutoHide.006358FD
00635A2E   .   59            POP ECX                              AutoHide.006358FD
00635A2F   .   64:8910       MOV DWORD PTR FS:[EAX],EDX
00635A32   .   68 545A6300   PUSH AutoHide.00635A54
00635A37   >   8D45 EC       LEA EAX,[LOCAL.5]
00635A3A   .   BA 05000000   MOV EDX,5
00635A3F   .   E8 141BDDFF   CALL AutoHide.00407558
00635A44   .   8D45 08       LEA EAX,[ARG.1]
00635A47   .   E8 041BDDFF   CALL AutoHide.00407550
00635A4C   .   C3            RETN
00635A4D   .^  E9 C204DDFF   JMP AutoHide.00405F14
00635A52   .^  EB E3         JMP SHORT AutoHide.00635A37
00635A54   .   8BC3          MOV EAX,EBX
00635A56   .   5F            POP EDI                              AutoHide.006358FD
00635A57   .   5E            POP ESI                              AutoHide.006358FD
00635A58   .   5B            POP EBX                              AutoHide.006358FD
00635A59   .   8BE5          MOV ESP,EBP
00635A5B   .   5D            POP EBP                              AutoHide.006358FD
00635A5C   .   C2 0400       RETN 4
00635A5F       90            DB 90
00635A60   .   B85A6300      DD AutoHide.00635AB8
00635A64       00            DB 00
```

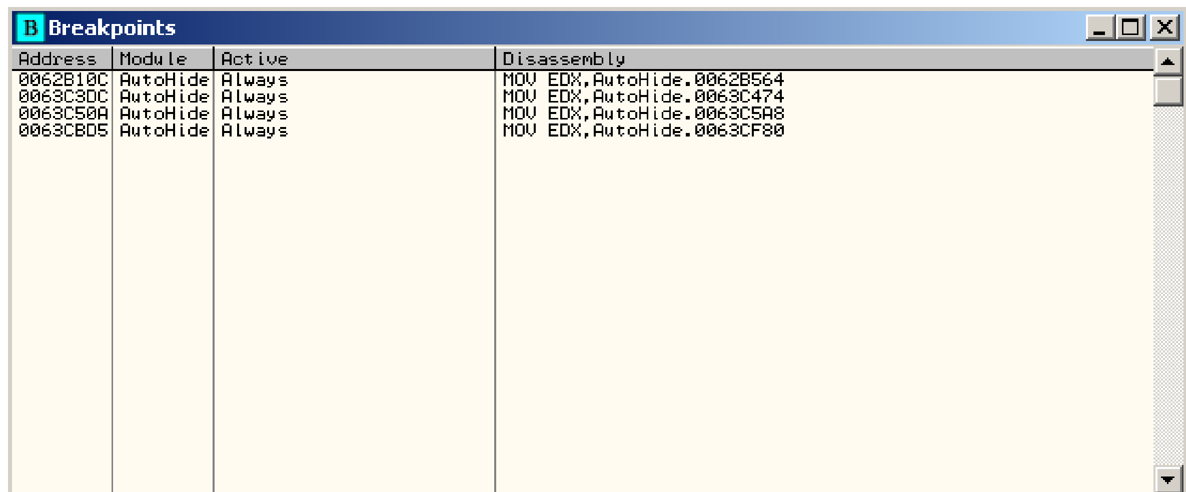
Reiniciamos y ejecutamos el programa:

Parece que todo ha salido bien. El programa ha aceptado nuestro serial y el periodo de prueba de

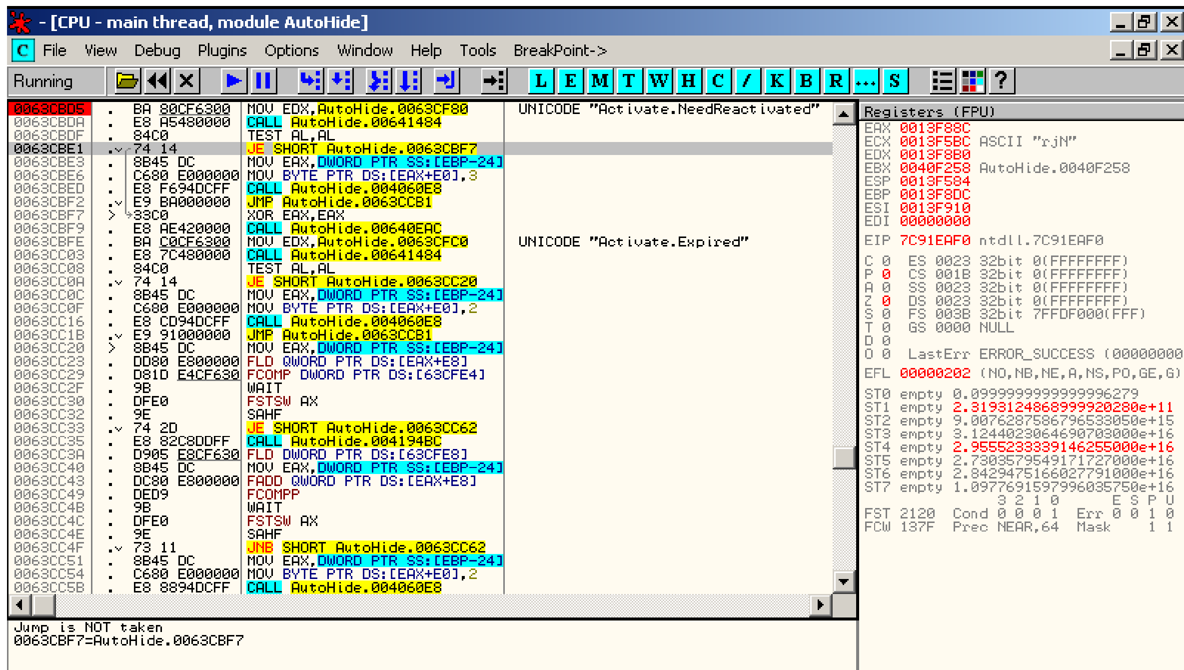
un día ha desaparecido. Pero hay un problema; cuando reiniciamos el equipo y ejecutamos el programa, el proceso de registración queda reseteado. Lo que significa que debe haber algún fichero que se cambia cuando cerramos el programa.

Si ponemos un punto de ruptura en “btnOKManuallyClick”, reiniciamos el programa y pulsamos F9, Olly se detendrá en nuestro punto de ruptura. Ahora sí podemos buscar por cadenas de texto. Buscamos todas la cadenas de texto “Activate.NeedReactivated” (hay cuatro en total) y ponemos un punto de ruptura en cada uno de ellos.

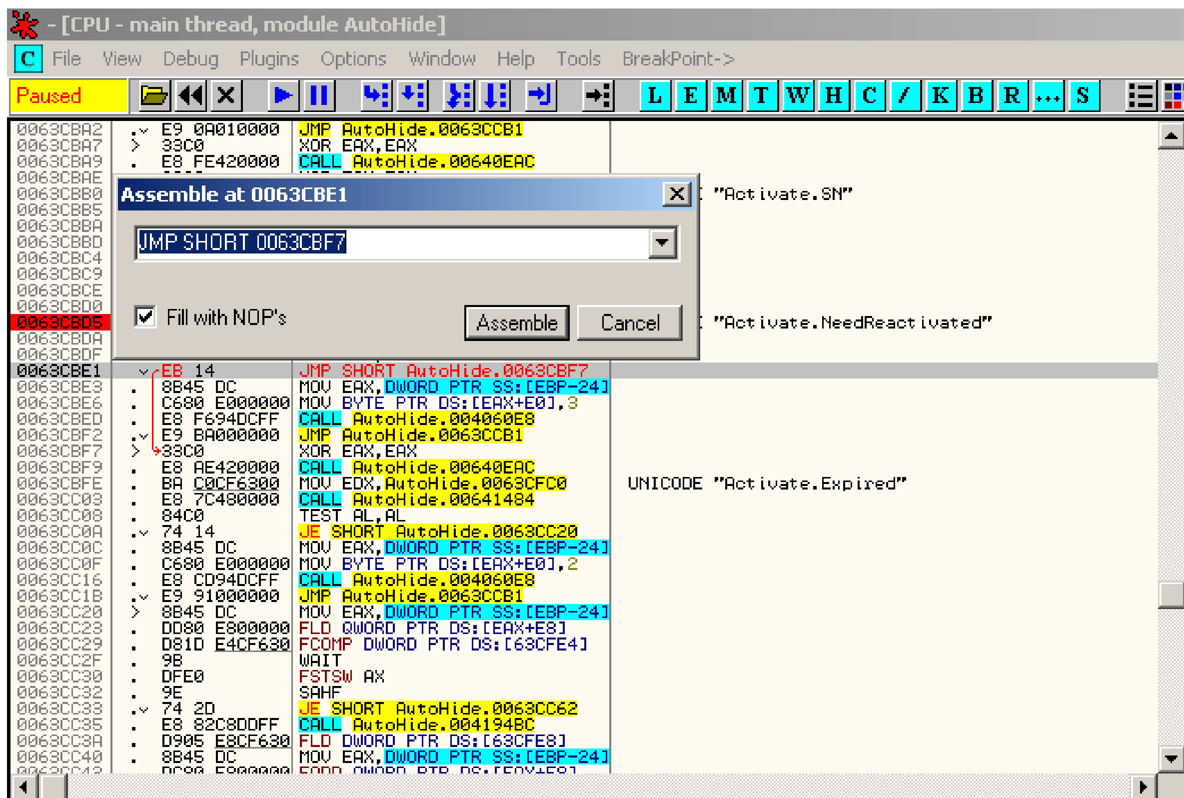




Reiniciamos el programa y pulsamos F9. Olly se detendrá en el punto de ruptura 63CB05.



Vemos un CALL y después un TEST AL,AL. Parece que comprueba si AL vale cero o uno. A continuación hay un salto. Cambiemos la instrucción para que salte siempre:

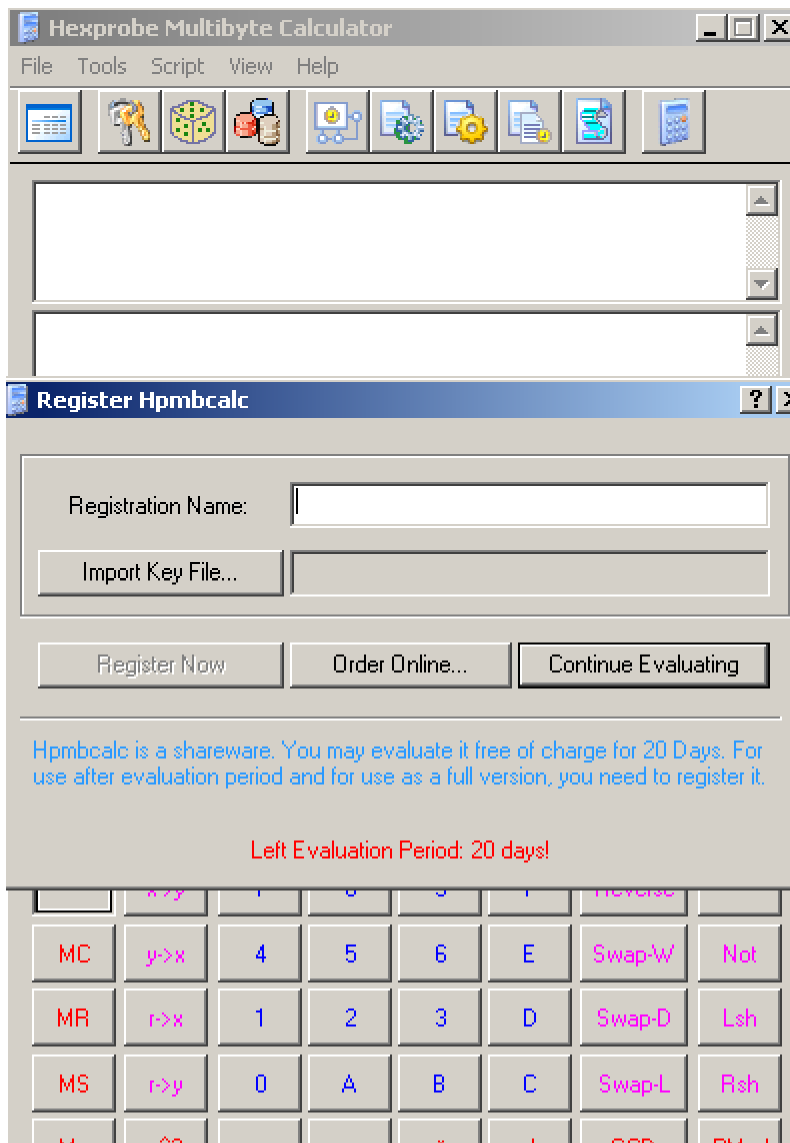


El resultado vuelve a ser alentador. Guardamos el ejecutable en disco, y lo ejecutamos (sin estar conectado a internet).

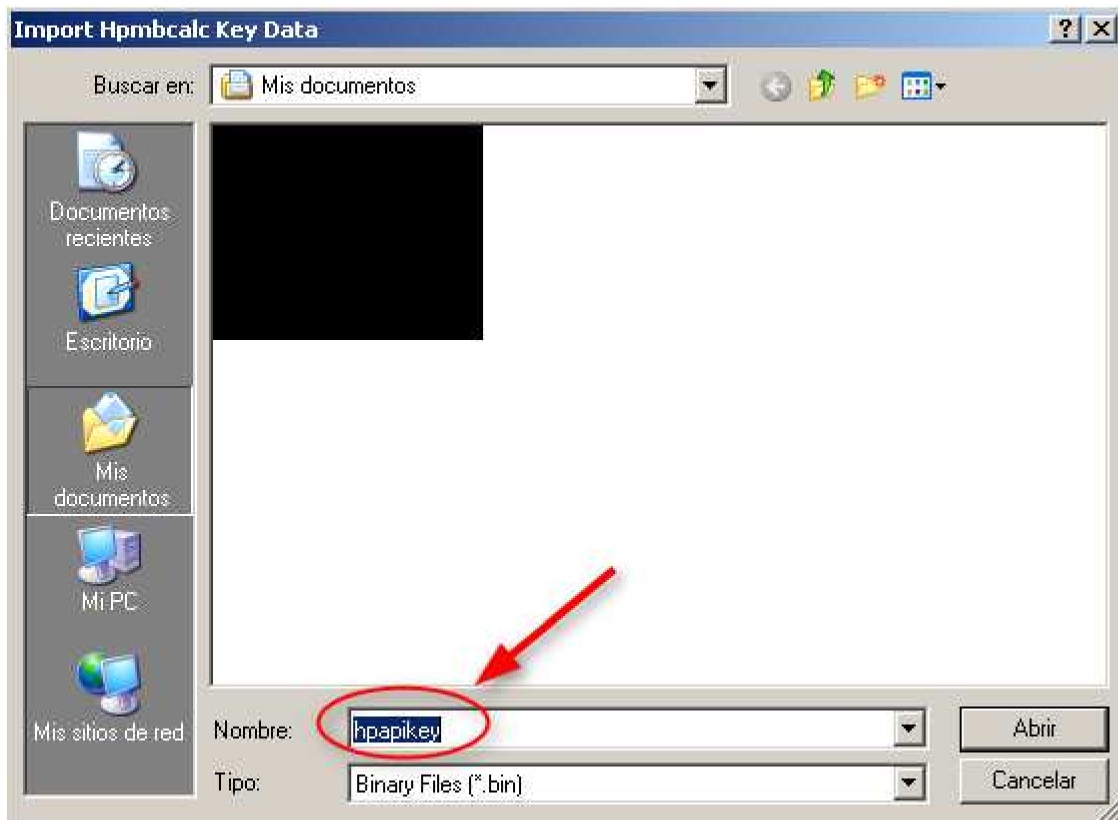
Y ahora la prueba definitiva. Nos conectamos a internet:

8.2 Bypasear un archivo de claves

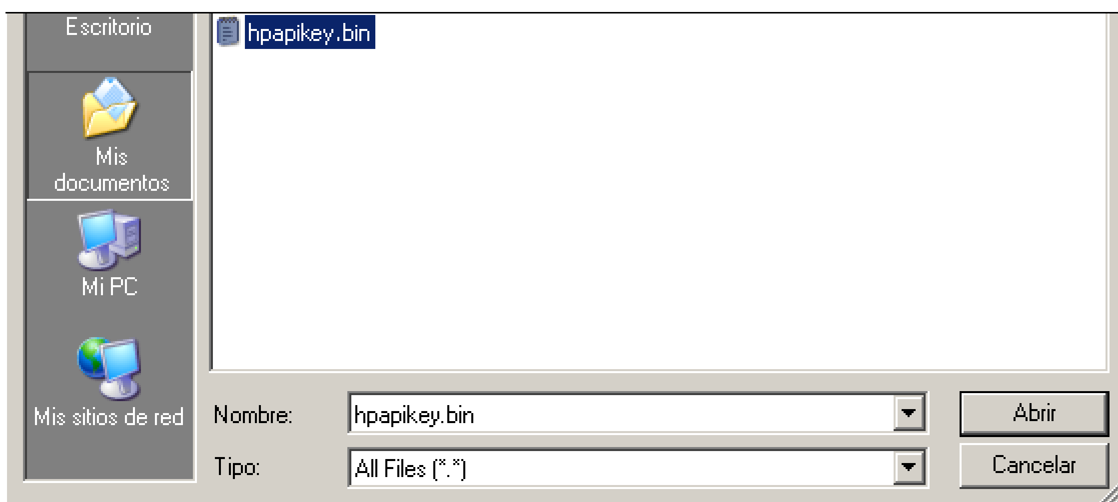
Descargamos el programa Hpmbcalc 4.22 de internet. Una vez instalado lo ejecutamos:



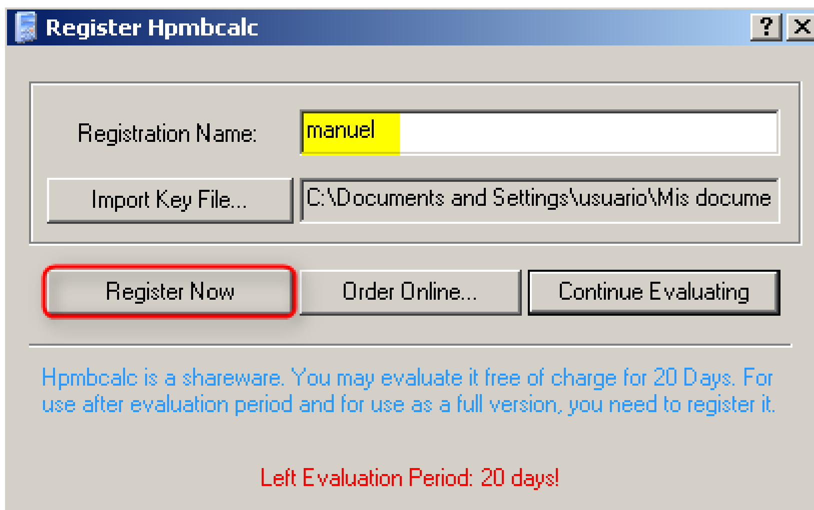
Vemos que el botón clave parece ser el "Import Key File". Por el nombre podemos deducir que lo que nos hace falta es un archivo de claves. Hacemos clic en el botón y se abre una ventana **con el nombre del archivo**:



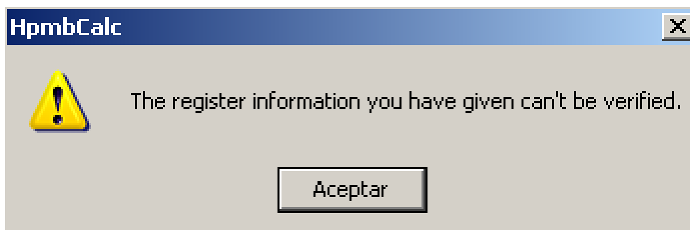
Abrimos un documento de texto y lo guardamos con ese nombre: hpapikey.bin. Volvemos hacer clic en el botón “Import Key File” y seleccionamos el documento de texto que acabamos de crear:



A continuación escribimos nuestro nombre y hacemos clic en “Register Now” (antes de seleccionar el archivo de claves este botón no estaba activo).

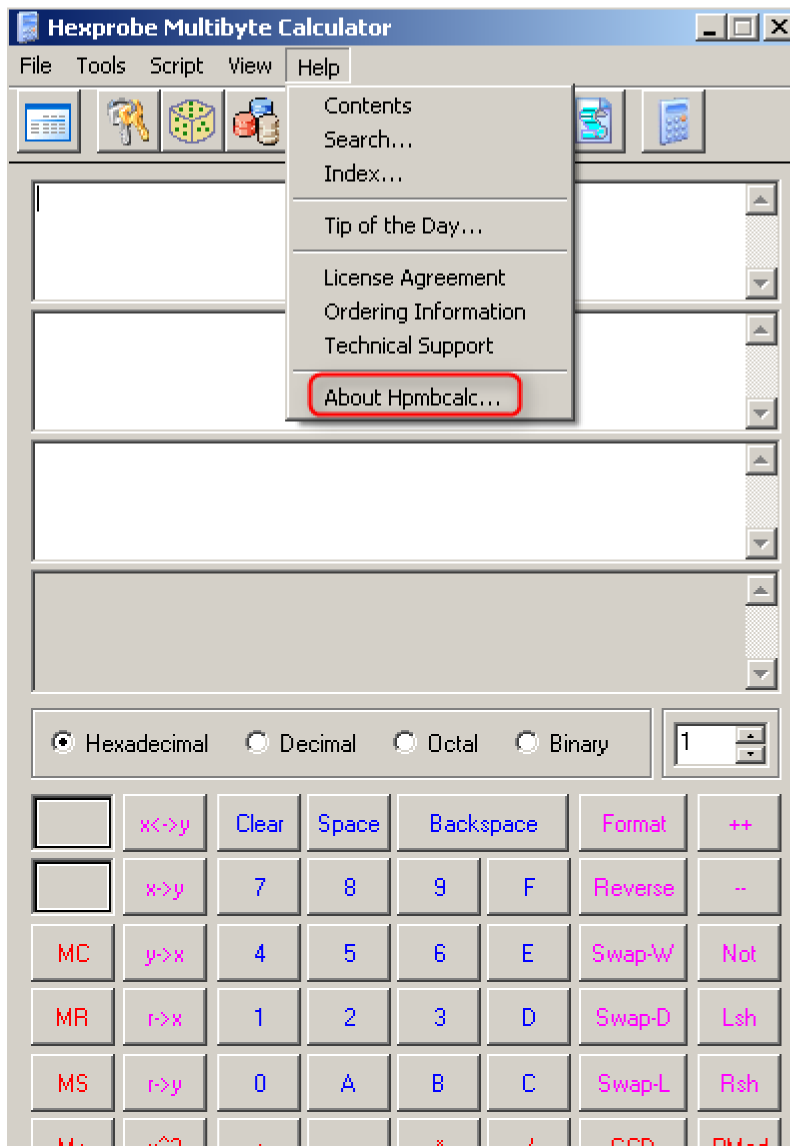


Ya tenemos nuestro "bad boy"



Hacemos clic en "Aceptar" y volvemos a la ventana de inicio.

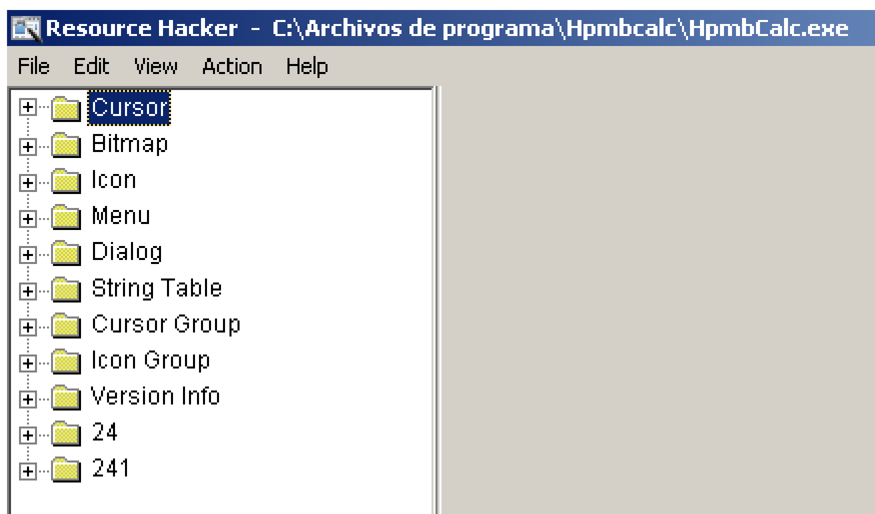
Pulsemos ahora "Continue Evaluating". Seleccionamos "Help" -> "About Hpmbcalc…"



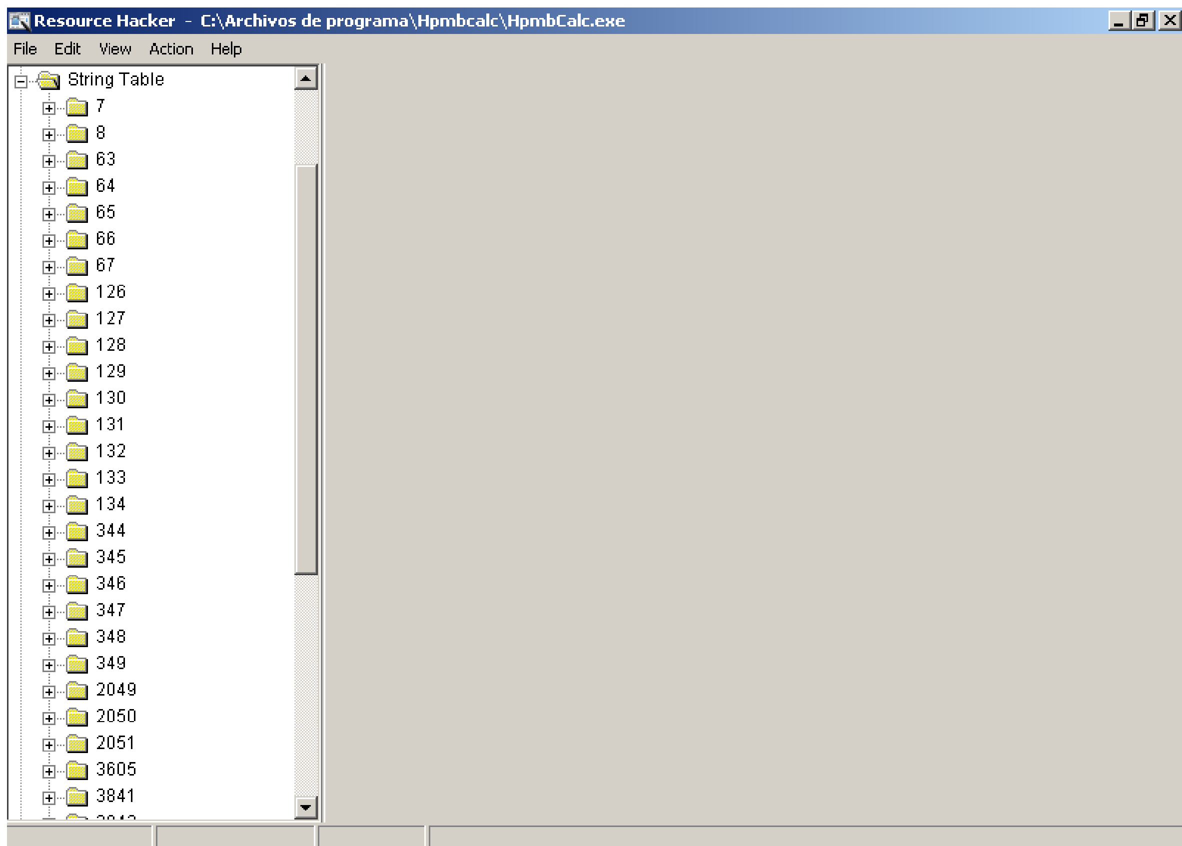
Y nos aparece la siguiente pantalla:

Esto es todo lo que necesitamos saber. Abrimos Olly y cargamos el programa. Si buscamos nuestro "bad boy" por cadenas de texto no vamos a tener éxito… Lo mismo ocurre si empleamos la técniqua del "Call Stack".

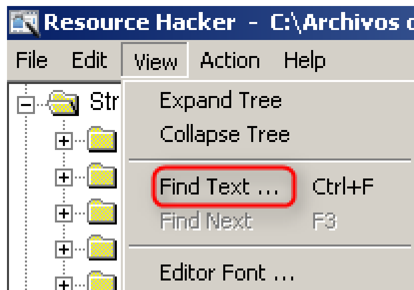
Así que cargamos el programa en "Resource Hacker".



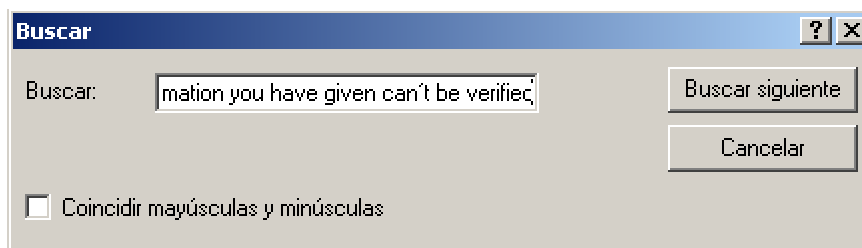
Desplegamos la carpeta de "String Table" (estamos buscando una cadena de texto).

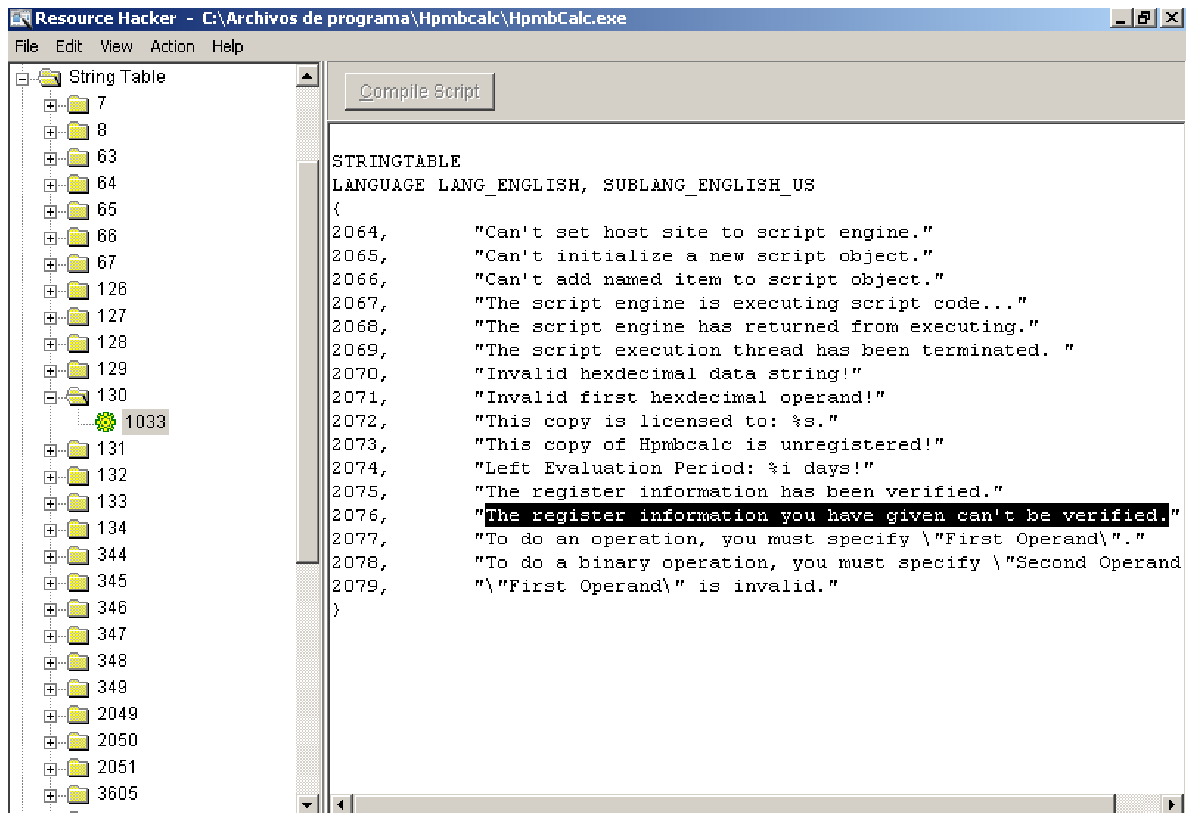


Podemos seleccionar carpeta por carpeta para buscar nuestra cadena de texto, pero lo más saludable es ir al menú y seleccionar "View" -> "Find Text":

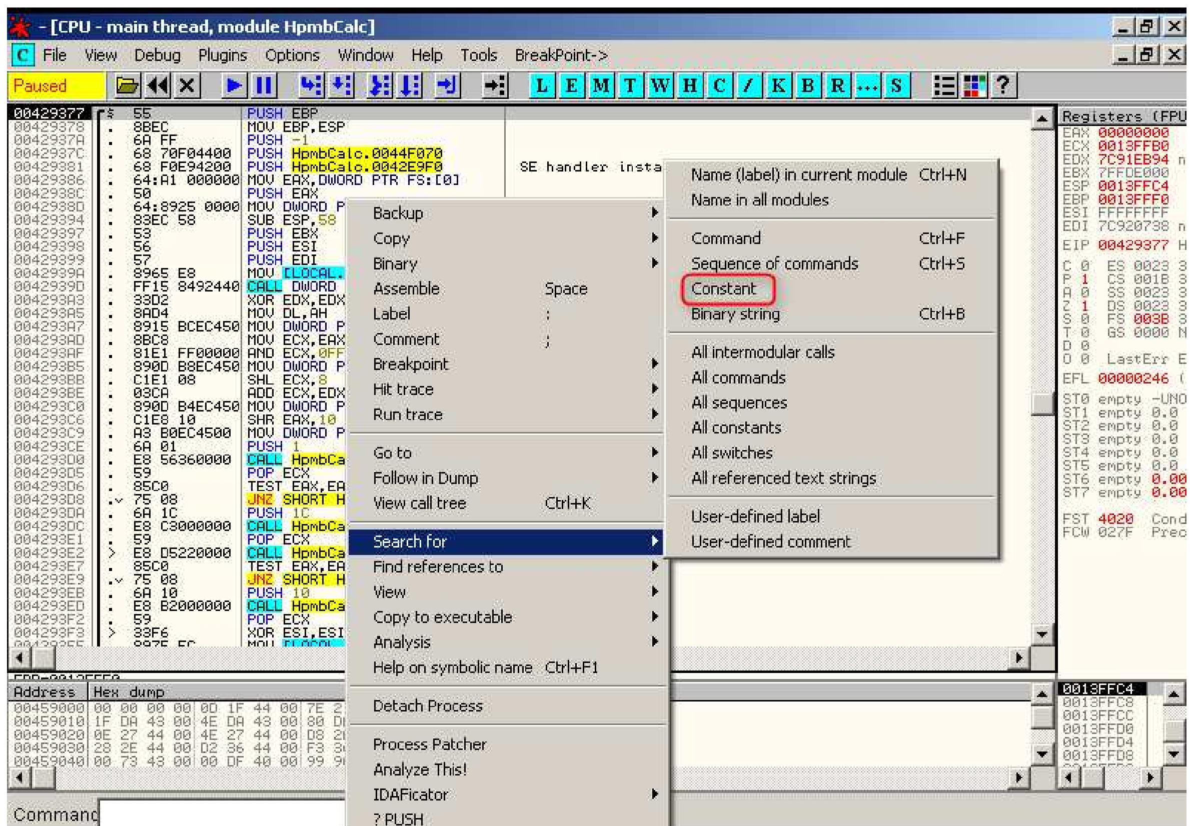


En el campo de texto que aparece a continuación escribimos el texto de nuestro "bad boy":

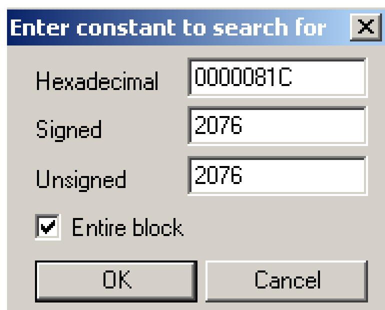




Una vez encontrado el “bad boy” fijemonos ahora en el número delante de la cadena de texto; 2076. Es el número que utiliza el programa para empujar (PUSH) esta cadena. Busquemos ahora ese número en Olly. Hacemos clic con el botón derecho y seleccionamos “Search for” -> “Constant”:



Introducimos en número en los campos “Signed” o “Unsigned” y Olly lo convierte en hexadecimal:

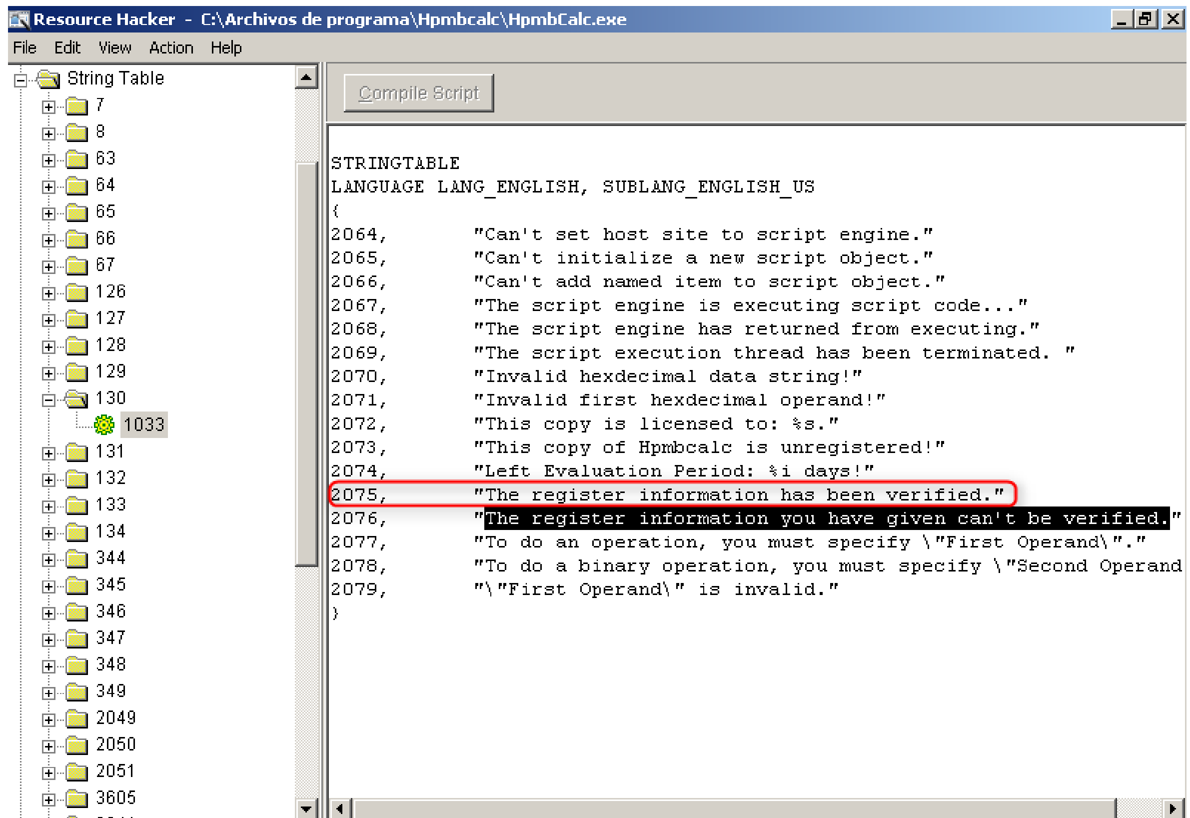


Hacemos clic en “Ok” y Olly nos lleva a la siguiente sección de código:

Vemos que en la dirección 419BF5 tenemos la instrucción PUSH 81C, y 81C como vimos antes es nuestro “bad boy”. Dos líneas antes tenemos un RETN, por lo que debe haber algún salto que nos lleve a la dirección 419BF5. También sabemos (por la ‘flecha’ gris delante del opcode) que hay un salto hacia la dirección 419BF3. Si hacemos clic en esa línea vemos que efectivamente hay un salto que proviene de la dirección 419BDC.

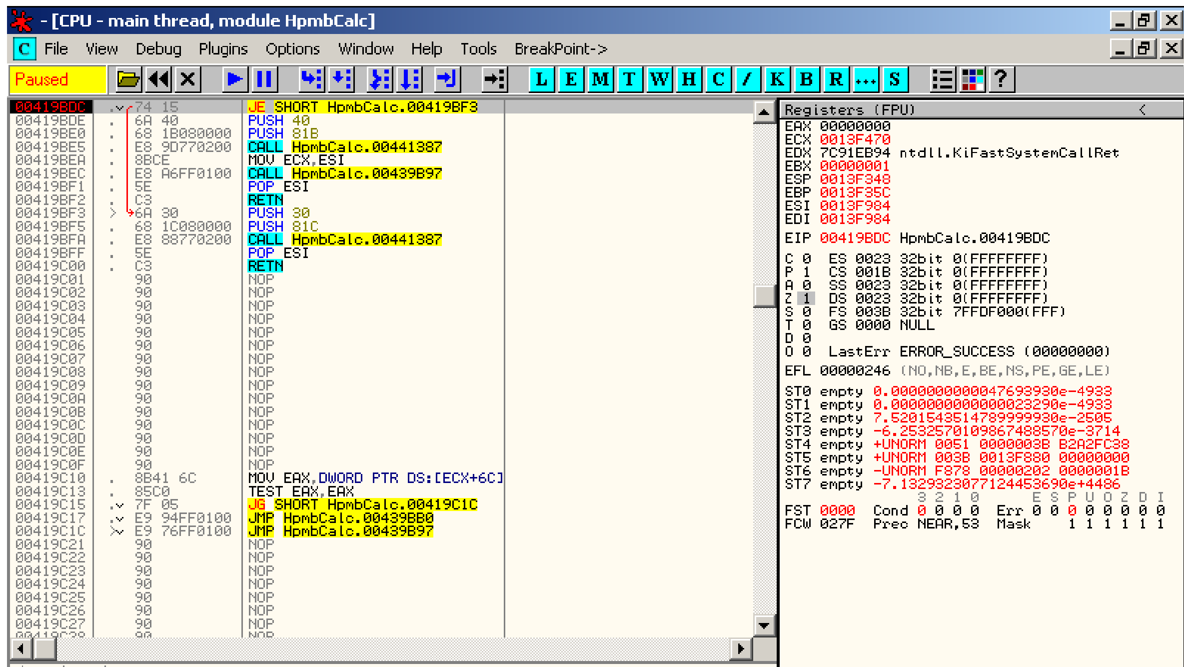
Podríamos cambiar el valor de la bandera para no saltar, pero ya que tenemos abierto el Resource Hacker tratemos de sacarle más información. Dos líneas por debajo del salto en la

dirección 419BE0 tenemos la instrucción PUSH 81B. Si convertimos el hexadecimal 81B en decimal nos da 2075. Y en el Resource Hacker este número está justo por encima de nuestro “bad boy”:

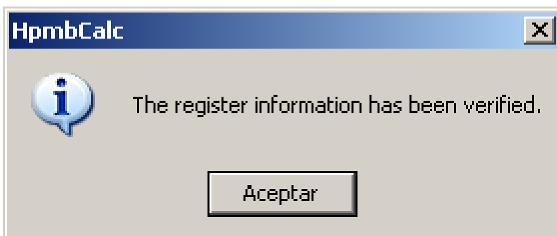


¡Hemos encontrado al “good boy”!

Pongamos pues un punto de ruptura en el salto 419BDC y cambiemos el valor de la bandera Z para que no efectué el salto. Reiniciamos, pulsamos F9, seleccionamos nuestro archivo de claves, introducimos nuestro nombre y hacemos clic en “Register Now”. Olly se detiene en nuestro punto de ruptura.



Volvemos a cambiar el valor de la bandera Z para evitar el salto y pulsamos F9:



Esto suena bien. Hacemos clic en 'Aceptar' y comoprobamos en la ventana "About" el resultado de nuestro parche:

Llegados a este punto me paro un momento para reflexionar sobre el sentido de la vida esperando que ello me llene con energía suficiente como para seguir buscando ("Entre las dificultades se esconde la oportunidad" Albert Einstein).

Volvamos al punto de ruptura y analicemos el código con más detenimiento:

```
- [CPU - main thread, module HpmbCalc]
C File View Debug Plugins Options Window Help Tools BreakPoint->
Running                                   L E M T W H C / K B R ...

00419BCB     90          NOP
00419BCC     90          NOP
00419BCD     90          NOP
00419BCE     90          NOP
00419BCF     90          NOP
00419BD0  .  56          PUSH ESI
00419BD1  .  8BF1        MOV ESI,ECX
00419BD3  .  E8 88FEFFFF CALL HpmbCalc.00419A60
00419BD8  .  85C0        TEST EAX,EAX
00419BDA  .  6A FF       PUSH -1
00419BDC  .v 74 15       JE SHORT HpmbCalc.00419BF3
00419BDE  .  6A 40       PUSH 40
00419BE0  .  68 1B080000 PUSH 81B
00419BE5  .  E8 9D770200 CALL HpmbCalc.00441387
00419BEA  .  8BCE        MOV ECX,ESI
00419BEC  .  E8 A6FF0100 CALL HpmbCalc.00439B97
00419BF1  .  5E          POP ESI
00419BF2  .  C3          RETN
00419BF3  >  6A 30       PUSH 30
00419BF5  .  68 1C080000 PUSH 81C
00419BFA  .  E8 88770200 CALL HpmbCalc.00441387
00419BFF  .  5E          POP ESI
00419C00  .  C3          RETN
00419C01     90          NOP
00419C02     90          NOP
00419C03     90          NOP
00419C04     90          NOP
00419C05     90          NOP
00419C06     90          NOP
00419C07     90          NOP
00419C08     90          NOP
00419C09     90          NOP
00419C0A     90          NOP
00419C0B     90          NOP
00419C0C     90          NOP
00419C0D     90          NOP
00419C0E     90          NOP
00419C0F     90          NOP
00419C10  .  8B41 6C     MOV EAX,DWORD PTR DS:[ECX+6C]
00419C13  .  85C0        TEST EAX,EAX
```

Vemos que antes del salto hay una combinación de llamada/comprobación. Pongamos pues un punto de ruptura en la instrucción CALL en 419BD3. Reiniciamos la aplicación, rellenamos los datos del registro y cuando Olly se detenga en el Call, pulsaremos F7 para entrar en él:

Vemos que se trata de una rutina bastante larga. A primera vista podemos identificar tres subrutinas; CreateFileA, GetFileSize y ReadFile.

El CALL a CreateFileA abrirá nuestro archivo de claves:

GetFileSize hace lo que indica su nombre: contar el número de bytes de nuestro archivo de claves para almacenarlo en EAX.

```
00419ACA  > 6A 00          PUSH 0                              ┌pFileSizeHigh = NULL
00419ACC  . 57             PUSH EDI                            │hFile = 00000160
00419ACD  . FF15 1493440   CALL DWORD PTR DS:[<&KERNEL32       └GetFileSize
00419AD3  . 83F8 08        CMP EAX,8
00419AD6  . 894424 0C      MOV DWORD PTR SS:[ESP+C],EAX
00419ADA  .v 0F82 C800000  JB HpmbCalc.00419BA8
00419AE0  . 3D 00010000    CMP EAX,100
00419AE5  .v 0F87 BD00000  JA HpmbCalc.00419BA8
00419AEB  . 8D4424 0C      LEA EAX,DWORD PTR SS:[ESP+C]
```

Si EAX no es igual a 8, entonces el programa salta al CALL ReadFile. Pero esto no es lo que queremos puesto que ReadFile también hace lo que indica su nombre y leerá los bytes de nuestro archivo de claves, así que no queremos saltar por encima de esa rutina.

Editemos nuestro archivo de claves para escribir en el 8 caracteres (letras o números) y así completar los 8 bytes.

Pasamos por ReadFile pulsando F8 y vemos como lee los 8 bytes de nuestro archivo. Luego se mueven los datos a los registros para ser empujados posteriormente. Y así llegamos al CALL en 419B11:

```
- [CPU - main thread, module HpmbCalc]
File View Debug Plugins Options Window Help Tools BreakPoint->
Paused

00419ACD  . FF15 1493440   CALL DWORD PTR DS:[<&KERNEL32       └GetFileSize
00419AD3  . 83F8 08        CMP EAX,8
00419AD6  . 894424 0C      MOV DWORD PTR SS:[ESP+C],EAX
00419ADA  .v 0F82 C800000  JB HpmbCalc.00419BA8
00419AE0  . 3D 00010000    CMP EAX,100
00419AE5  .v 0F87 BD00000  JA HpmbCalc.00419BA8
00419AEB  . 8D4424 0C      LEA EAX,DWORD PTR SS:[ESP+C]
00419AEF  . 6A 00          PUSH 0                              ┌pOverlapped = NULL
00419AF1  . 50             PUSH EAX                            │pBytesRead = 003F6E98
00419AF2  . 8D4C24 1C      LEA ECX,DWORD PTR SS:[ESP+1C]       │
00419AF6  . 68 00040000    PUSH 400                            │BytesToRead = 400 (1024.)
00419AFB  . 51             PUSH ECX                            │Buffer = 0013EF3C
00419AFC  . 57             PUSH EDI                            │hFile = 00000160
00419AFD  . FF15 2493440   CALL DWORD PTR DS:[<&KERNEL32       └ReadFile
00419B03  . 8B5424 0C      MOV EDX,DWORD PTR SS:[ESP+C]
00419B07  . 8B43 60        MOV EAX,DWORD PTR DS:[EBX+60]
00419B0A  . 8D4C24 14      LEA ECX,DWORD PTR SS:[ESP+14]
00419B0E  . 52             PUSH EDX
00419B0F  . 51             PUSH ECX
00419B10  . 50             PUSH EAX
00419B11  . E8 EAA2FFFF    CALL HpmbCalc.00413E00
00419B16  . 8BF0           MOV ESI,EAX
00419B18  . 83C4 0C        ADD ESP,0C
00419B1B  . 85F6           TEST ESI,ESI
00419B1D  .v 75 50         JNZ SHORT HpmbCalc.00419B6F
00419B1F  . 8B15 F8AA450   MOV EDX,DWORD PTR DS:[45AAF8]       HpmbCalc.0045AB0C
00419B25  . 895424 10      MOV DWORD PTR SS:[ESP+10],EDX
00419B29  . 898424 1C040   MOV DWORD PTR SS:[ESP+41C],EA
00419B30  . 8B43 60        MOV EAX,DWORD PTR DS:[EBX+60]
00419B33  . 8D4C24 10      LEA ECX,DWORD PTR SS:[ESP+10]
00419B37  . 51             PUSH ECX
00419B38  . 50             PUSH EAX
00419B39  . E8 D2F9FFFF    CALL HpmbCalc.00419510
00419B3E  . 8B5424 14      MOV EDX,DWORD PTR SS:[ESP+14]
00419B42  . 8B4C24 18      MOV ECX,DWORD PTR SS:[ESP+18]

00413E00=HpmbCalc.00413E00

Registers (FPU)
EAX 003F6E98 ASCII "manuel"
ECX 0013EF3C
EDX 00000008
EBX 0013F984
ESP 0013EF1C
EBP 0013F35C
ESI 00000000
EDI 00000160
EIP 00419B11 HpmbCalc.00419B11
C 0  ES 0023 32bit 0(FFFFFFFF)
P 0  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDF000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)
EFL 00000202 (NO,NB,NE,A,NS,PO,GE,G)
ST0 empty 0.0000000000047693930e-4933
ST1 empty 0.0000000000000023290e-4933
ST2 empty 7.5021067012617041210e-2505
ST3 empty -6.2532570109818664250e-3714
ST4 empty +UNORM 0054 0000003B B2353C38
ST5 empty +UNORM 003B 0013F880 00000000
ST6 empty -UNORM F878 00000202 0000001B
ST7 empty 5.9954135328367932730e-446
               3 2 1 0      E S P U O Z
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0
FCW 027F  Prec NEAR,53  Mask    1 1 1 1
```

Llegados a este punto sabemos que el programa acaba de leer nuestro archivo de claves, y tiene el resultado almacenado en los registros. Existe la posibilidad que este CALL en 419B11 compruebe esos datos con el código correcto. Pulsemos F7 para entrar en el CALL:

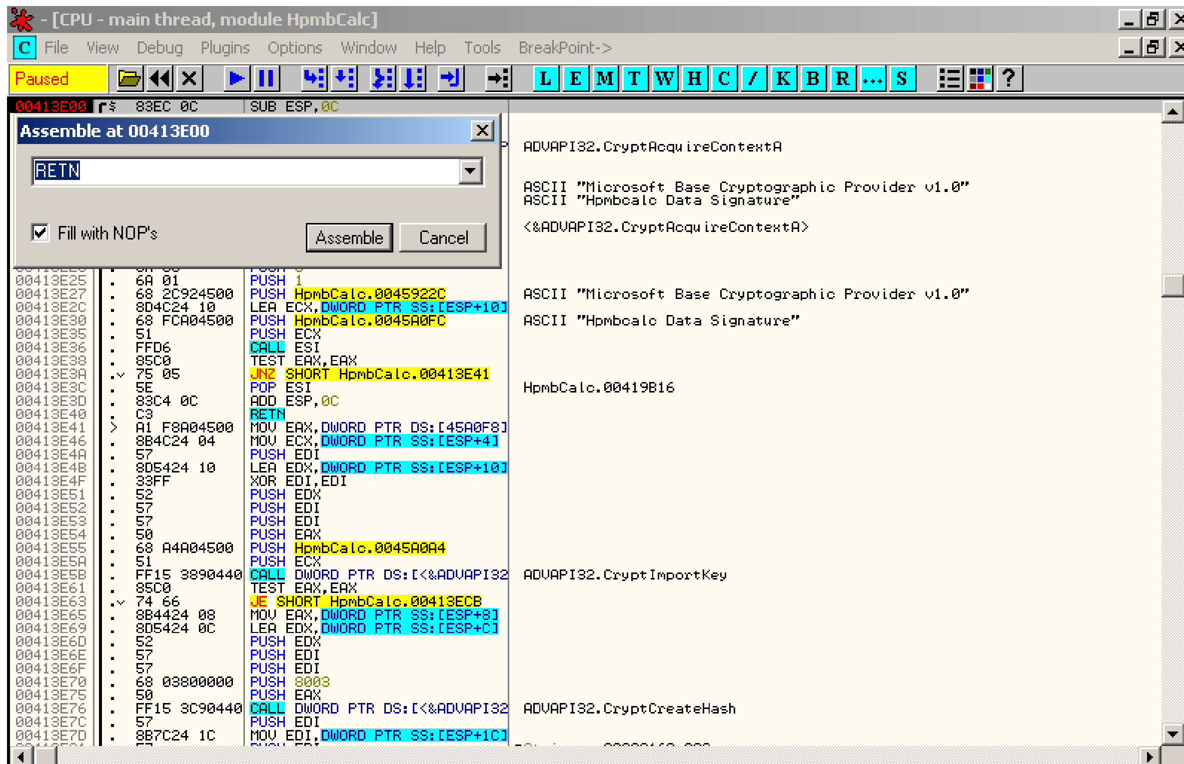
```
- [CPU - main thread, module HpmbCalc]
File View Debug Plugins Options Window Help Tools BreakPoint->
Paused

00413E00  $  83EC 0C         SUB ESP,0C
00413E03  .  8D4424 00       LEA EAX,DWORD PTR SS:[ESP]
00413E07  .  56              PUSH ESI
00413E08  .  8B35 0490440    MOV ESI,DWORD PTR DS:[<&ADVAP   ADVAPI32.CryptAcquireContextA
00413E0E  .  6A 00           PUSH 0
00413E10  .  6A 01           PUSH 1
00413E12  .  68 2C924500     PUSH HpmbCalc.0045922C          ASCII "Microsoft Base Cryptographic Provider v1.0"
00413E17  .  68 FCA04500     PUSH HpmbCalc.0045A0FC          ASCII "Hpmbcalc Data Signature"
00413E1C  .  50              PUSH EAX
00413E1D  .  FFD6            CALL ESI                        <&ADVAPI32.CryptAcquireContextA>
00413E1F  .  85C0            TEST EAX,EAX
00413E21  .v 75 1E           JNZ SHORT HpmbCalc.00413E41
00413E23  .  6A 08           PUSH 8
00413E25  .  6A 01           PUSH 1
00413E27  .  68 2C924500     PUSH HpmbCalc.0045922C          ASCII "Microsoft Base Cryptographic Provider v1.0"
00413E2C  .  8D4C24 10       LEA ECX,DWORD PTR SS:[ESP+10]
00413E30  .  68 FCA04500     PUSH HpmbCalc.0045A0FC          ASCII "Hpmbcalc Data Signature"
00413E35  .  51              PUSH ECX
00413E36  .  FFD6            CALL ESI
00413E38  .  85C0            TEST EAX,EAX
00413E3A  .v 75 05           JNZ SHORT HpmbCalc.00413E41
00413E3C  .  5E              POP ESI                         HpmbCalc.00419B16
00413E3D  .  83C4 0C         ADD ESP,0C
00413E40  .  C3              RETN
00413E41  >  A1 F8A04500     MOV EAX,DWORD PTR DS:[45A0F8]
00413E46  .  8B4C24 04       MOV ECX,DWORD PTR SS:[ESP+4]
00413E4A  .  57              PUSH EDI
00413E4B  .  8D5424 10       LEA EDX,DWORD PTR SS:[ESP+10]
00413E4F  .  33FF            XOR EDI,EDI
00413E51  .  52              PUSH EDX
00413E52  .  57              PUSH EDI
00413E53  .  57              PUSH EDI
00413E54  .  50              PUSH EAX
00413E55  .  68 A4A04500     PUSH HpmbCalc.0045A0A4
00413E5A  .  51              PUSH ECX
00413E5B  .  FF15 3890440    CALL DWORD PTR DS:[<&ADVAPI32   ADVAPI32.CryptImportKey
00413E61  .  85C0            TEST EAX,EAX
00413E63  .v 74 66           JE SHORT HpmbCalc.00413ECB
00413E65  .  8B4424 08       MOV EAX,DWORD PTR SS:[ESP+8]
00413E69  .  8D5424 0C       LEA EDX,DWORD PTR SS:[ESP+C]
00413E6D  .  52              PUSH EDX
00413E6E  .  57              PUSH EDI
00413E6F  .  57              PUSH EDI
00413E70  .  68 03800000     PUSH 8003
00413E75  .  50              PUSH EAX
00413E76  .  FF15 3C90440    CALL DWORD PTR DS:[<&ADVAPI32   ADVAPI32.CryptCreateHash
00413E7C  .  57              PUSH EDI
00413E7D  .  8B7C24 1C       MOV EDI,DWORD PTR SS:[ESP+1C]
```

Parece que el serial está encriptado…y que la rutina es llamada de 8 sitios diferentes.!!

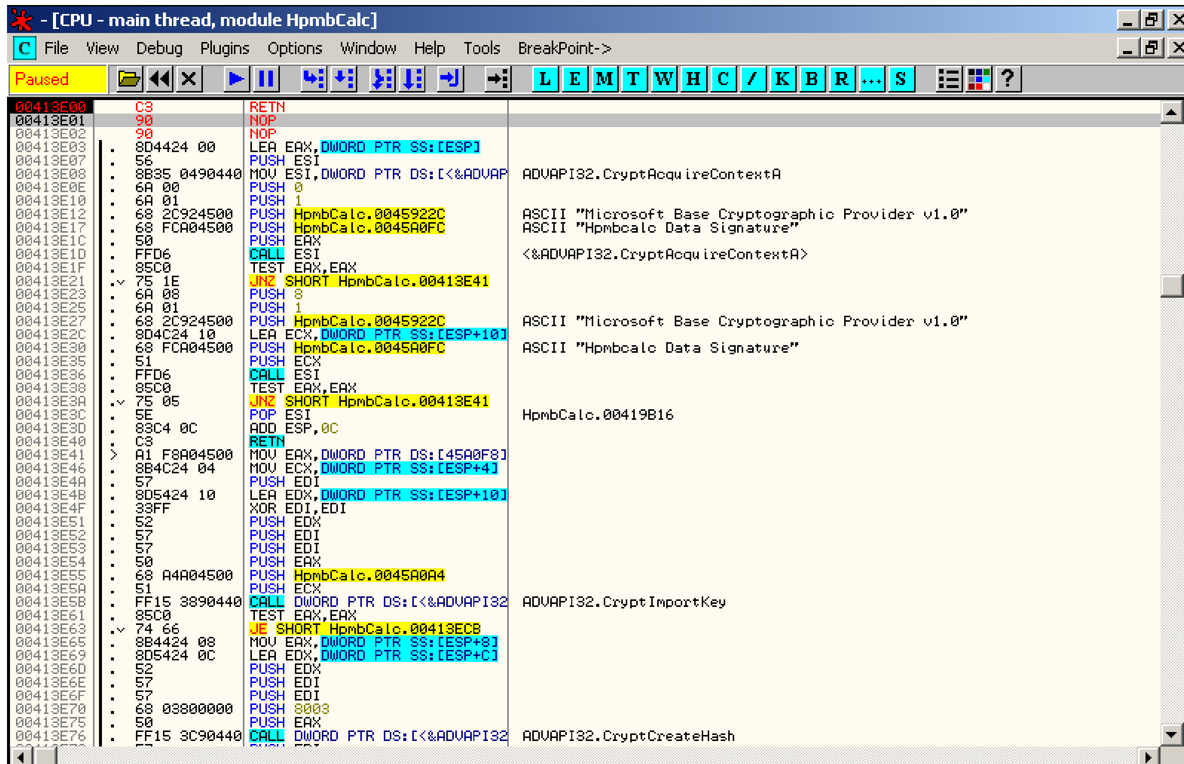
Call tree

Called from	Procedure	Calls	Comment
HpmbCalc.0040A7C3	HpmbCalc.00413E00	ADVAPI32.CryptAcquireContextA	Sys
HpmbCalc.0040A7ED		ADVAPI32.CryptReleaseContext	Sys
HpmbCalc.0040E9C4		ADVAPI32.CryptHashData	Sys
HpmbCalc.0040E9F0		ADVAPI32.CryptDestroyHash	Sys
HpmbCalc.0040F1C7		ADVAPI32.CryptCreateHash	Sys
HpmbCalc.0040F1F1		ADVAPI32.CryptDestroyKey	Sys
HpmbCalc.00419B11		ADVAPI32.CryptImportKey	Sys
HpmbCalc.00419B4D		ADVAPI32.CryptVerifySignatureA	Sys
		kernel32.lstrlenA	Sys
		Unknown destination(s)	

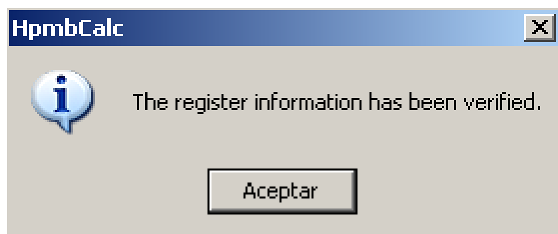
Podríamos cambiar todos los saltos para llegar al primer RETN, sin embargo el código después del RETN tampoco parece ser prometedor. Así que vamos a poner un RETN en la primera línea, 413E00, a la vez que ponemos un punto de ruptura:



Hacemos clic en “Assemble” y en “Cancel”.



Y pulsamos F9:



Este parche también nos lleva al "good boy". Hacemos clic en "Aceptar" y veamos lo que nos devuelve la ventana "About":

Si ahora guardamos el nuevo ejecutable parcheado y volvemos abrirlo, puede que se venga a bajo. Tal vez no en la primera vez pero si en sucesivas veces, al intentar ejecutar el programa, este se cierra de forma inesperada. Esto puede ser devido a que en algún lugar del código se esté llevando a cabo un segunda comprobación del serial. Podríamos intentar buscar esa comprobación y parchearla pero esto nos llevaría una eternidad. Lo que si haremos es un cargador para nuestro programa utilizando para ello dUP.

Abrimos dUP, seleccionamos un nuevo proyecto y cubrimos el "Patch Info":

Patch Info

Patcher Caption: Cargador

Application:

Filename (s): hpmbc422.exe ...

URL: visit

Author: manuel

Release Date: August 24, 2015 today

Release Info:

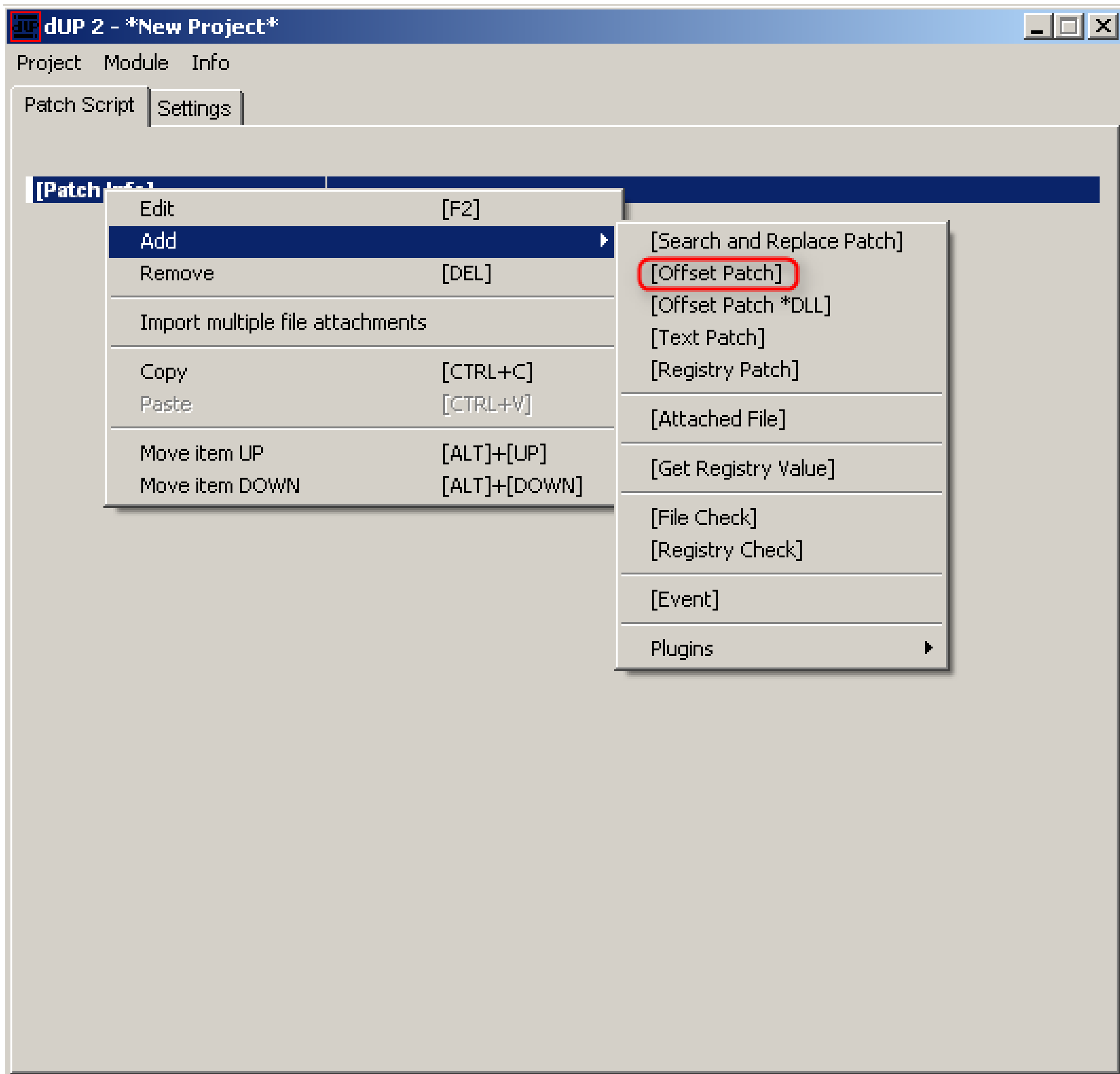
About Box Message: For more patches contact with, manuel.rey.vilar@gmail.com

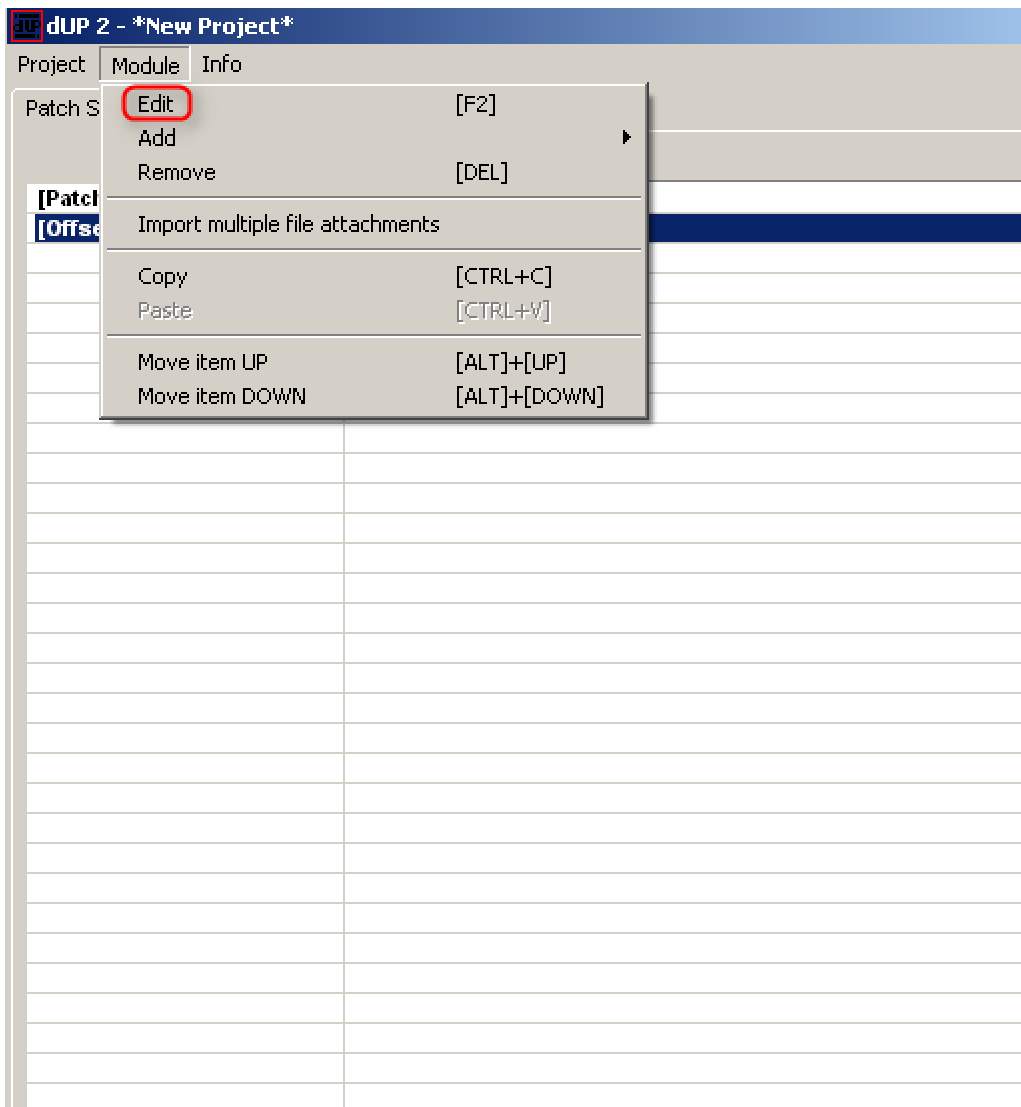
Scrolltext:

- [x] Show this dialog when create a new project
- [ ] Run patch with administrator rights
- [ ] No Backup by default

Cancel Save

Pulsamos “Save”, hacemos clic con el botón derecho sobre Patch Info y seleccionamos “Add” -> “Offset Patch”:


dUP 2 - *New Project*
Project
Module
Info
Patch Script
Settings
Edit [F2]
Add
Remove [DEL]
Import multiple file attachments
Copy [CTRL+C]
Paste [CTRL+V]
Move item UP [ALT]+[UP]
Move item DOWN [ALT]+[DOWN]
[Search and Replace Patch]
[Offset Patch]
[Offset Patch *DLL]
[Text Patch]
[Registry Patch]
[Attached File]
[Get Registry Value]
[File Check]
[Registry Check]
[Event]
Plugins



Marcamos la segunda línea y seleccionamos “Module” -> “Edit”:

Offset Patchdata

Target File

File: hpmbc422.exe

Old Filesize 001754C7 New Filesize

CRC32: A26DAE80 Enable CRC32 check

Don't check original (old) bytes Try patching used file

Keep original file time Fix PE CheckSum

Disable WOW64 FS Redirector

Patchmode

File Offset

Virtual Address (VA)

Relative Virtual Address (RVA

Compare Files

Original File:

Patched File:

Compare

VA	Old Byte	New Byte
00413E00	83	C3
00413E01	EC	90
00413E02	0C	90

Add and Edit

Offset

Original Byte

Patched Byte

Edit Add

Remove Clear List

MemCheck [for Loaders]

Enable MemCheck ?

MemoryAdress

MemoryValue:

Up Items

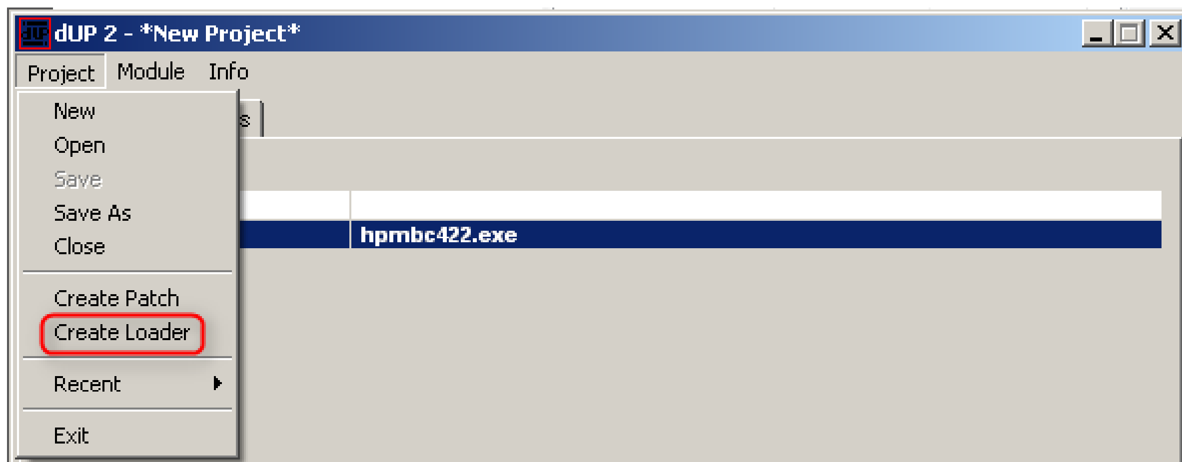
Down 3

File to load in Ollydbg

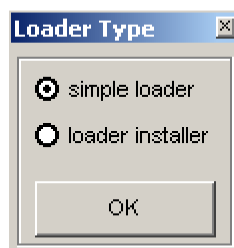
Follow in Ollydbg Cancel Save

Los datos del “Offset”, “Original Byte” y “Patched Byte” son tomados de Olly:

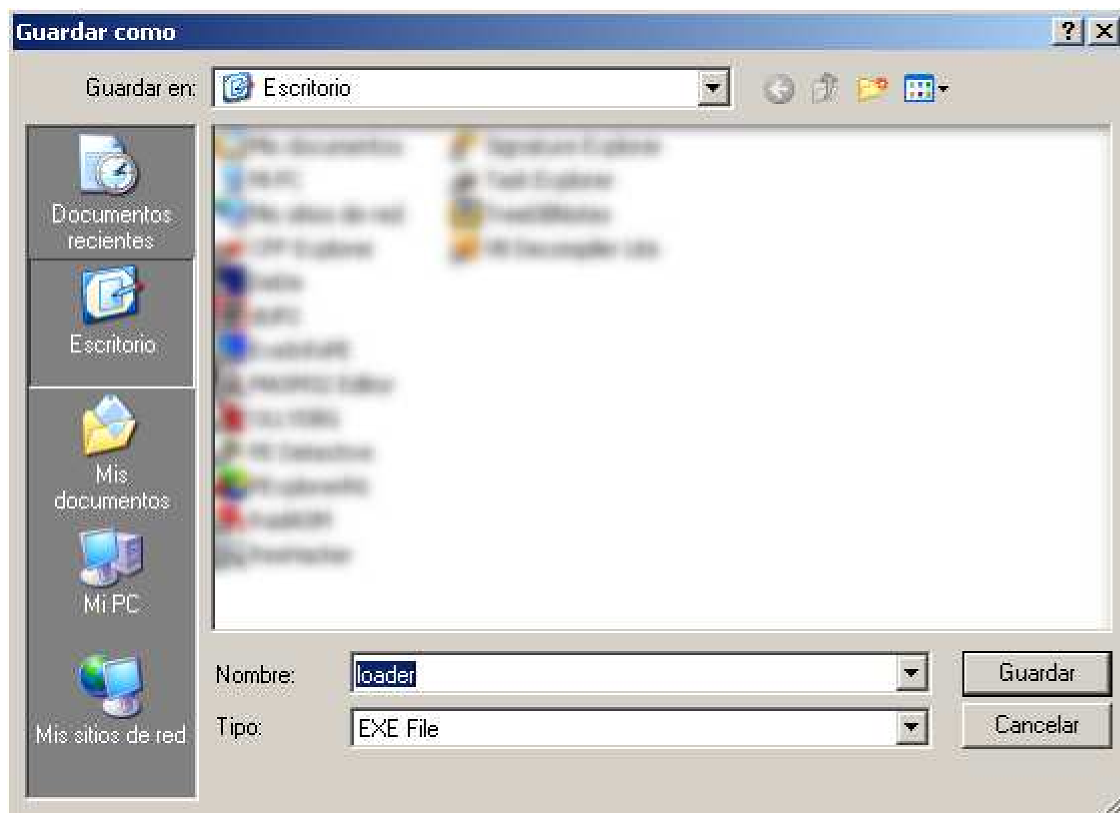
Después de cubrir el “Offset Patchdata” pulsamos “Save”:



Marcamos la línea y seleccionamos “Project” -> “Create Loader” -> “simple loader”:



Pulsamos en OK, le damos un nombre al cargador y guardamos:



8.3 Bypasear un serial en delphi

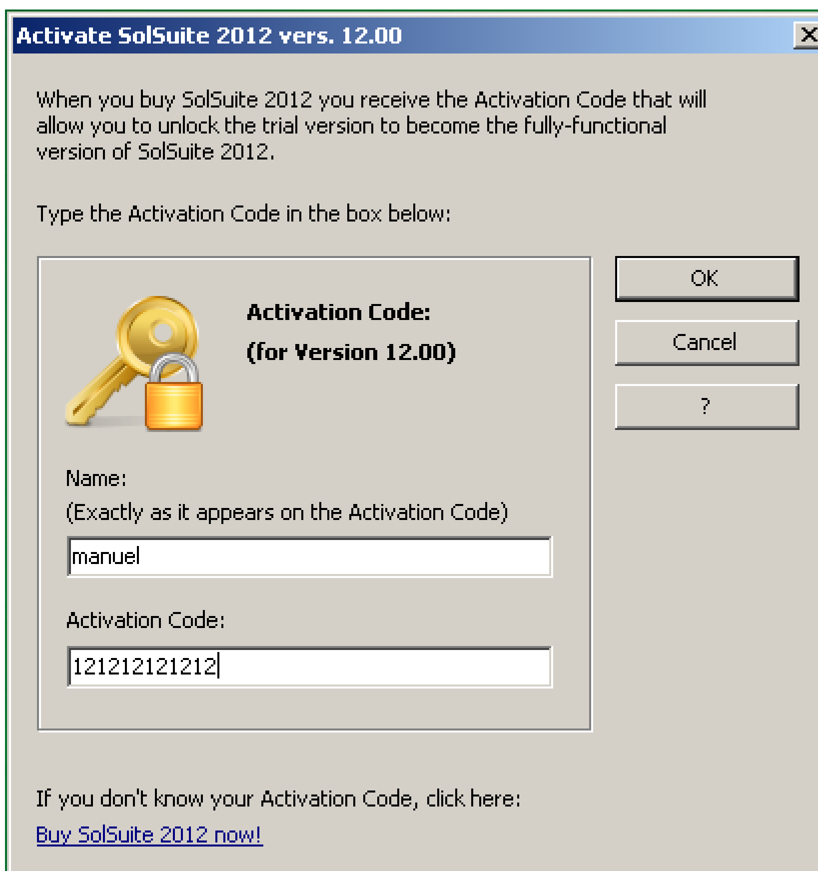
Descargamos SolSuite Solitaire de internet. Instalamos y ejecutamos el programa.

Vemos que estamos en periodo de prueba.

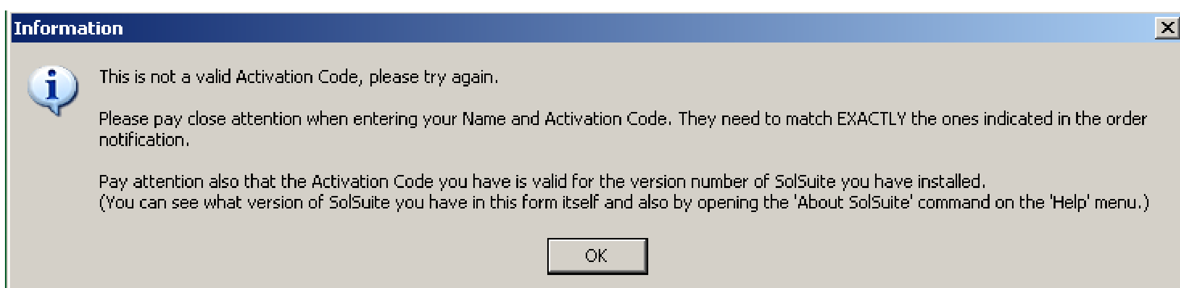
Después de cerrar todos los pop ups conseguimos acceder a la barra de menus. Hacemos clic en "Help" -> "About":

Dentro de la pestaña “Help” vamos intentar activar el programa:

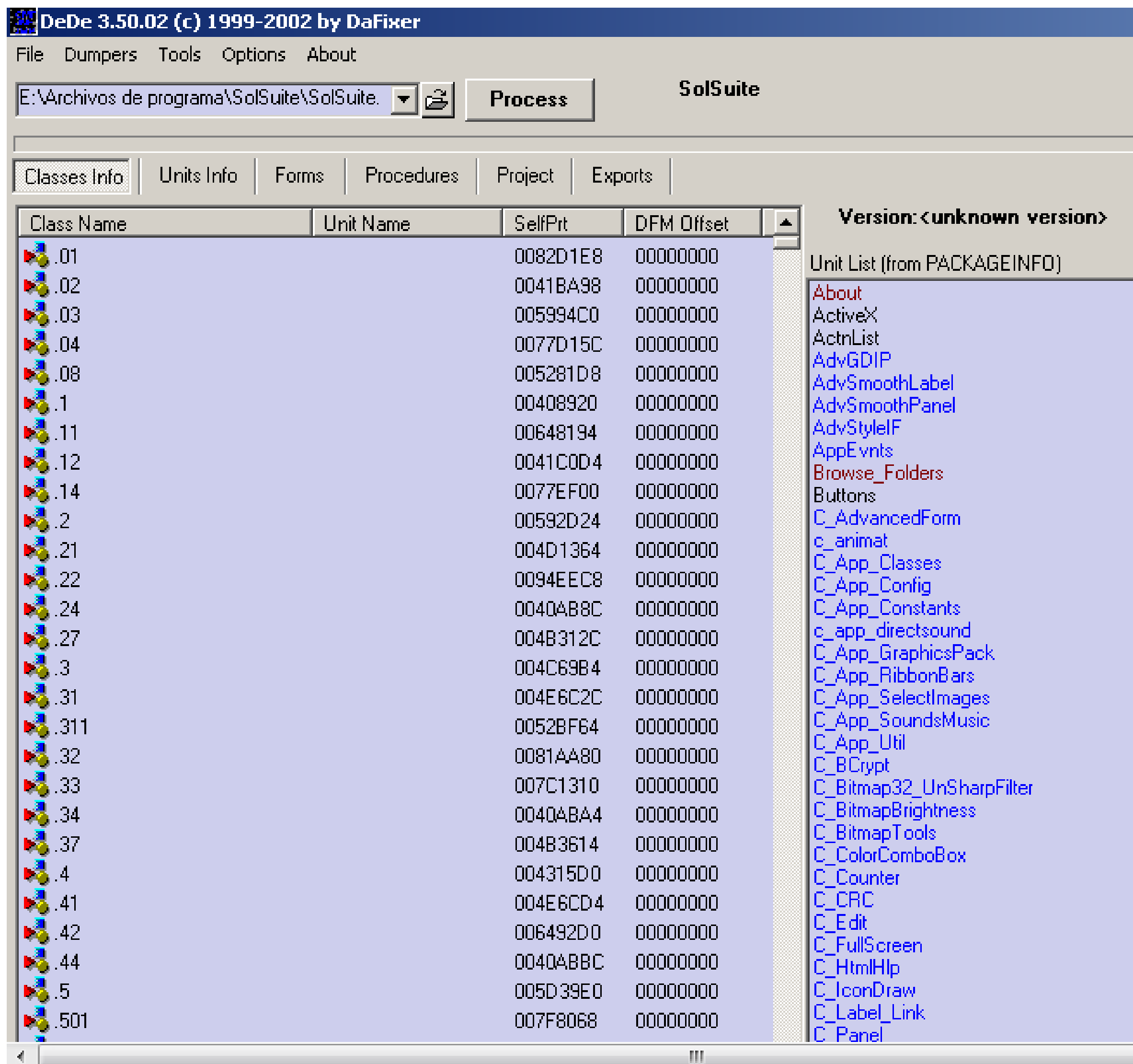
Introducimos un nombre y un código de activación:



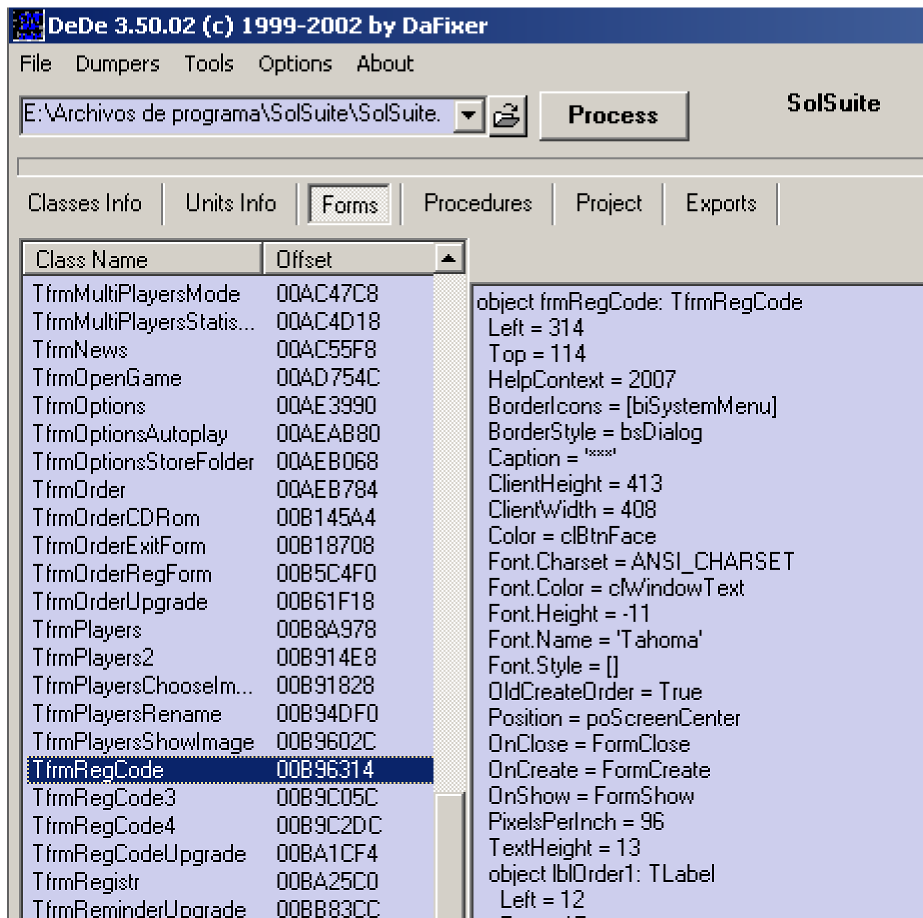
Hacemos clic en “OK” y aparece nuestro “bad boy”:



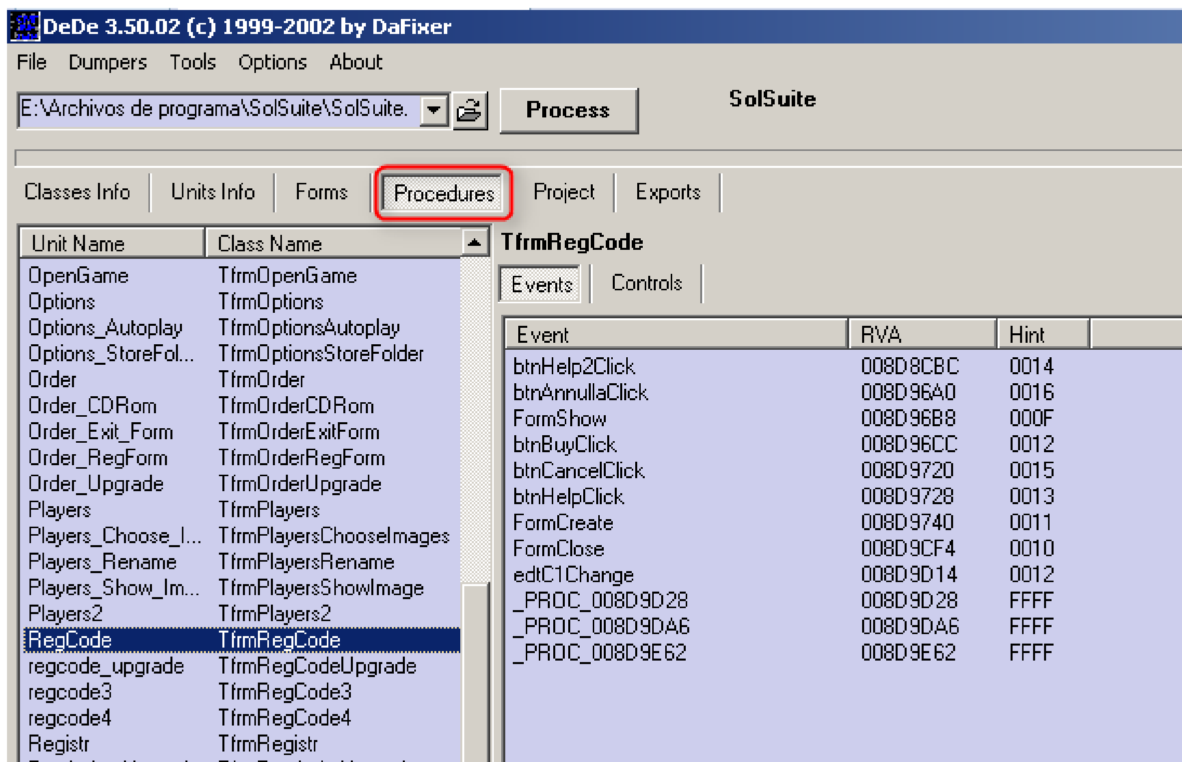
Normalmente cargaríamos ahora nuestro programa en Olly pero sabiendo que se trata de un programa en Delphi lo cargaremos en DeDe:



Una vez procesado, hacemos clic en la pestaña Forms y buscamos TfrmRegCode. Esta es nuestra ventana de registro (si dudamos hacemos doble clic).

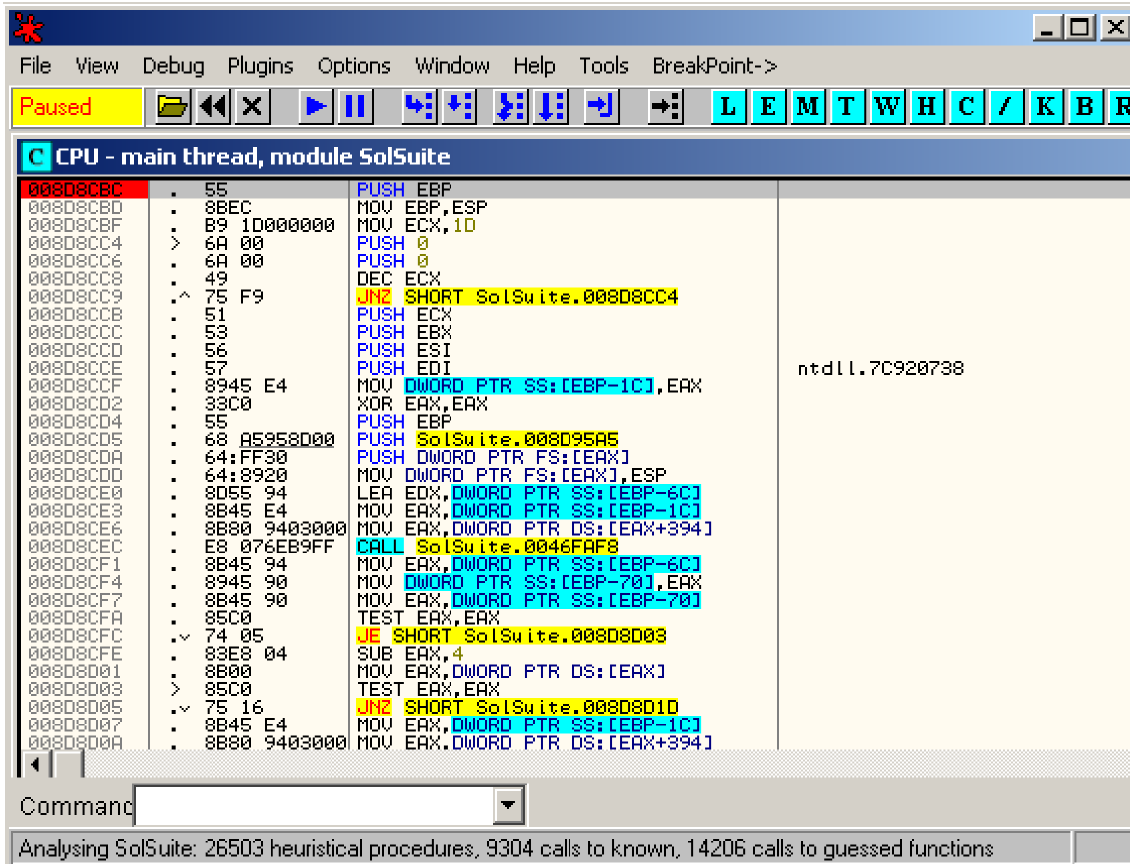


A continuación vamos a la pestaña "Procedure" y buscamos TfrmRegCode en la ventana de la izquierda:



Si nos fijamos en la ventana de la derecha parece que el autor del programa está intentando liar las cosas. Vemos que no hay ningún btnOkClick y sin embargo un btnHelp2Click y btnHelpClick. Veamos lo que hay en el primero de los dos en la dirección 8D8CBC.

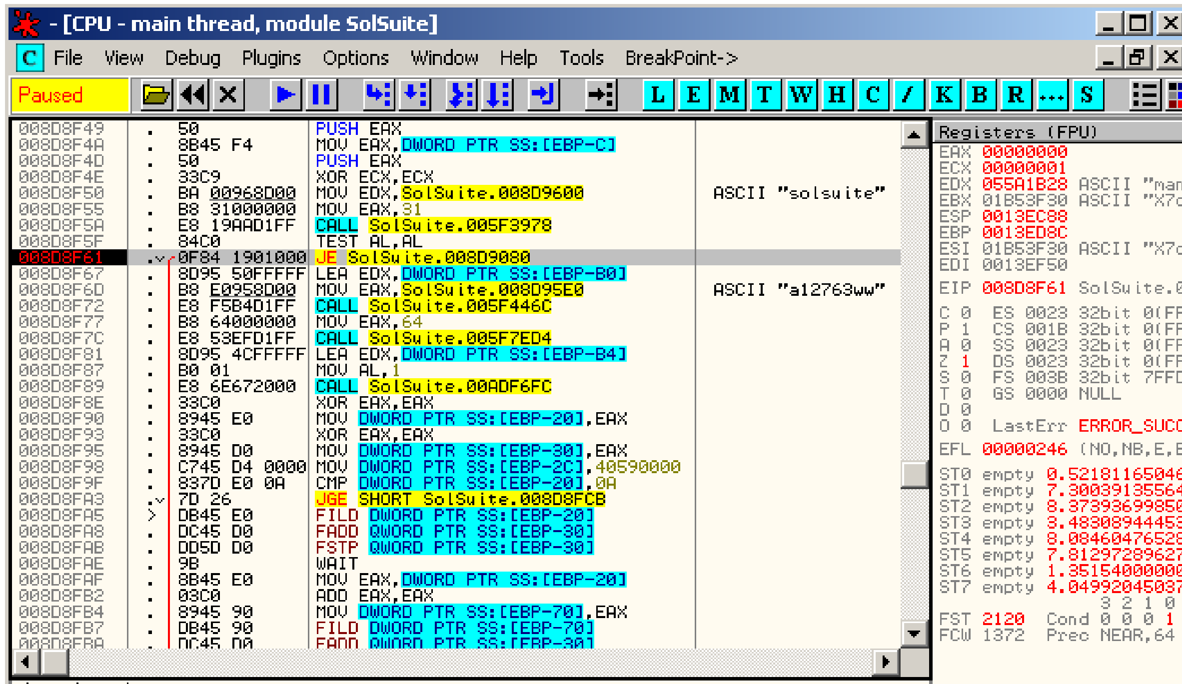
Abrimos Olly, cargamos el programa, vamos a la dirección 8D8CBC y ponemos un punto de ruptura:



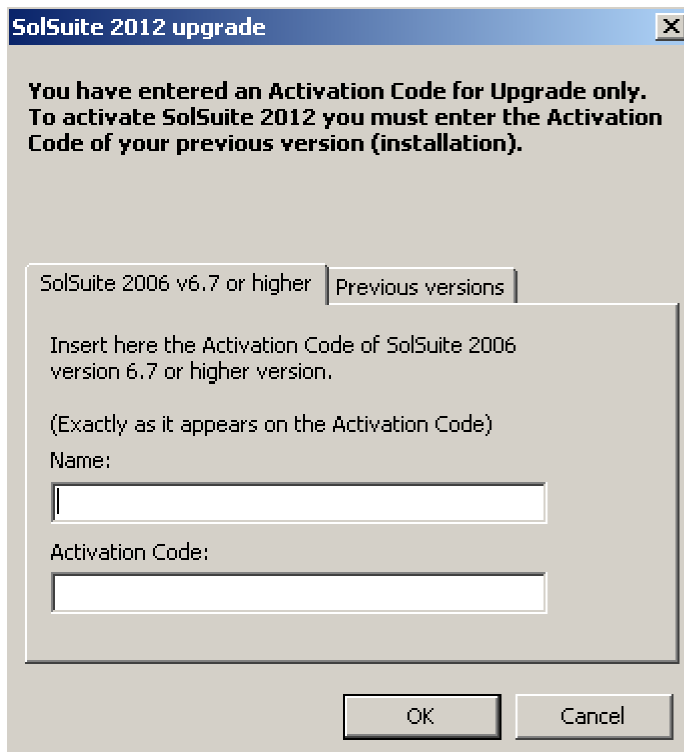
Reiniciamos Olly, introducimos un nombre y serial en la ventana de registro y Olly se detiene en nuestro punto de ruptura:

A continuación vamos a tener que hacer un rastreo largo de código incluido unos cuantos loops. Hasta que finalmente lleguemos a la siguiente instrucción TEST AL,AL:

Después de de unos cuantos tests, llegamos a este que llama la atención. Ponemos un punto de ruptura en el salto que hay a continuación y ejecutamos el programa:



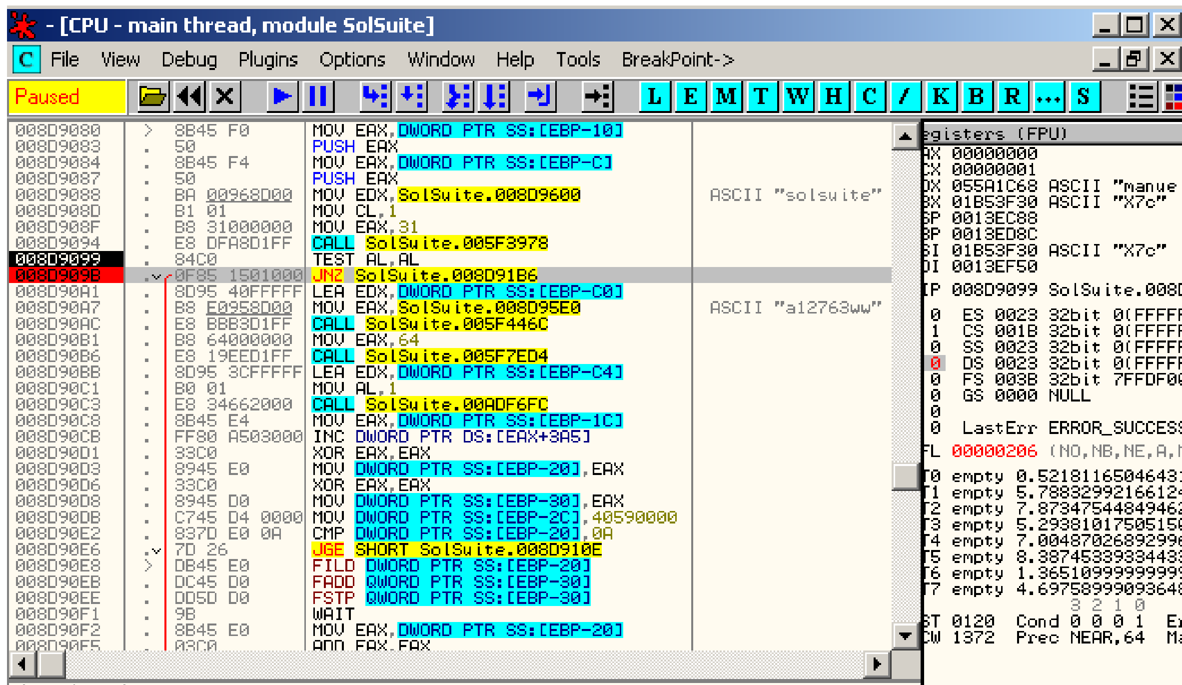
Cambiamos el valor de la bandera Z para que Olly no tome el salto y pulsamos F9:



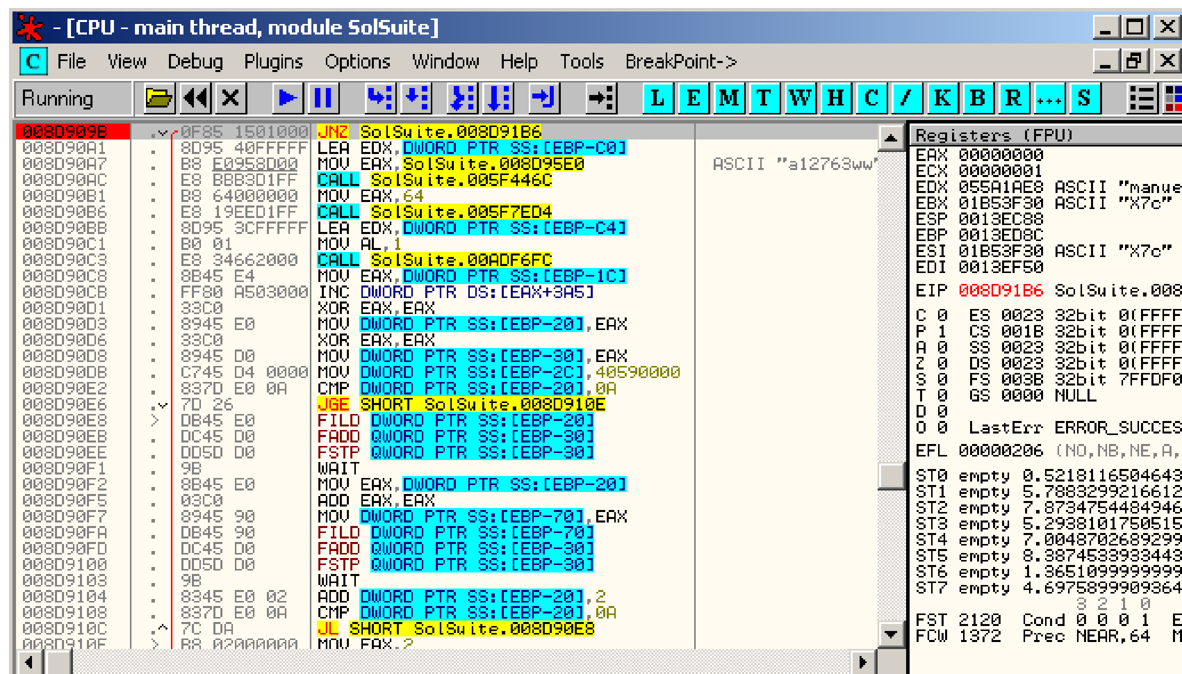
Esto no es del todo cierto, pero vamos por buen camino. Hacemos clic primero en “Cancel” luego en “OK” y dejamos que Olly corra hasta nuestro nuevo punto de ruptura:

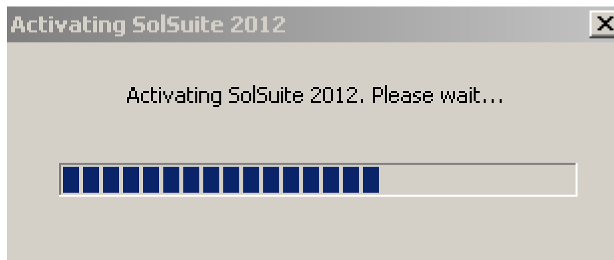
Quitamos el punto de ruptura (ya que no estaba situado en el lugar apropiado) y seguimos ejecutando código pulsando F8. Cogemos el salto y unas lineas más abajo aparece otro TEST AL, AL:

La última vez que pasamos alrededor de un TEST AL, AL, apareció un cuadro de dialogo por eso es lógico pensar que aquí suceda lo mismo. Ponemos pues un punto de ruptua en el salto 8D909B,y ejecutamos la aplicación:

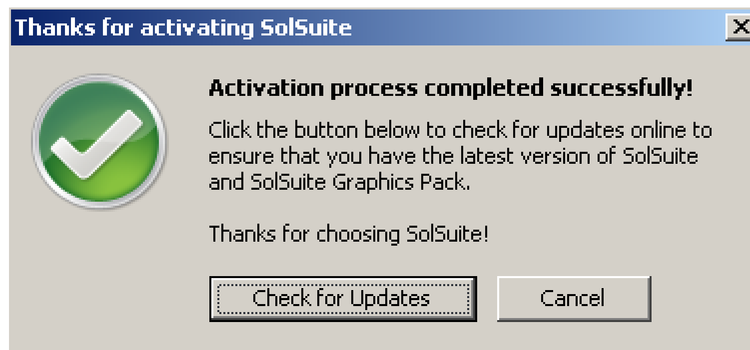


Cuando Olly se detenga en el punto de ruptura cambiamos el valor de la bandera Z y seguimos ejecutando el programa:

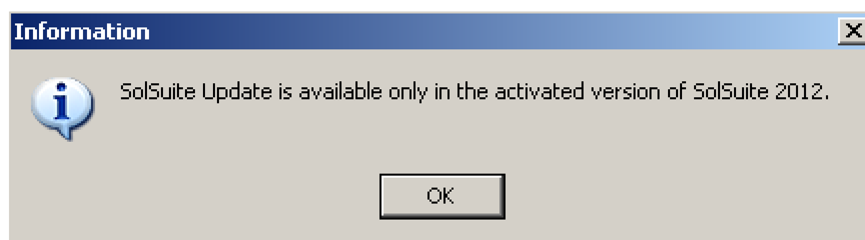




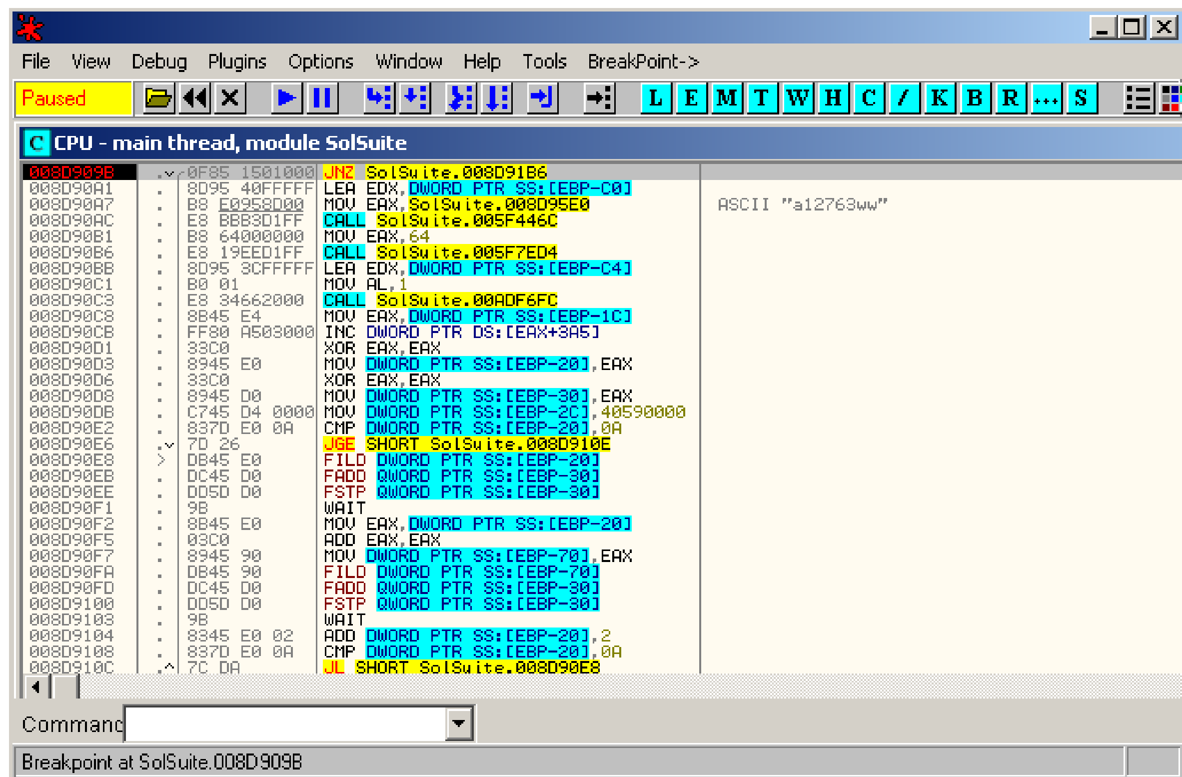
Y aparece nuestro “good boy”.



Si ahora pulsamos “Cancel” todo irá bien, pero si hacemos clic en “Check for Updates” aparecerá el siguiente cuadro de texto:

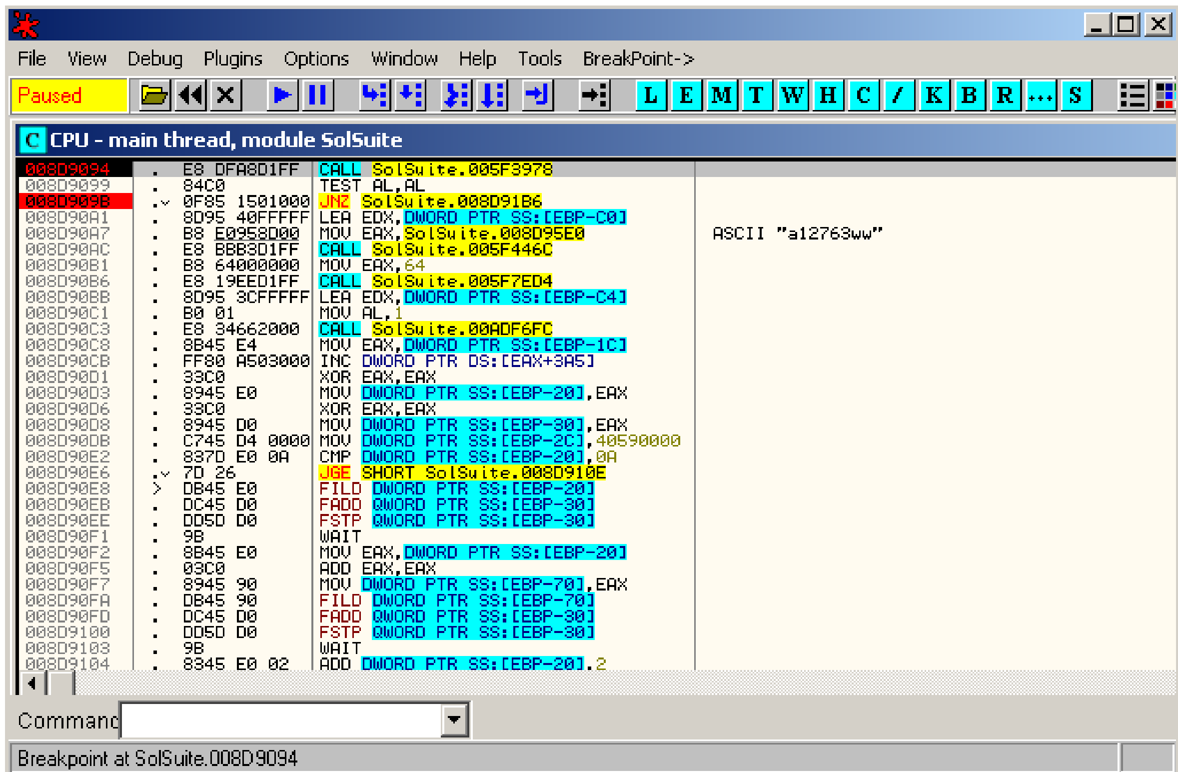


Parece ser que el parche que le pusimos no es suficiente. Reiniciamos Olly para volver a detenerso en el último punto de ruptura:

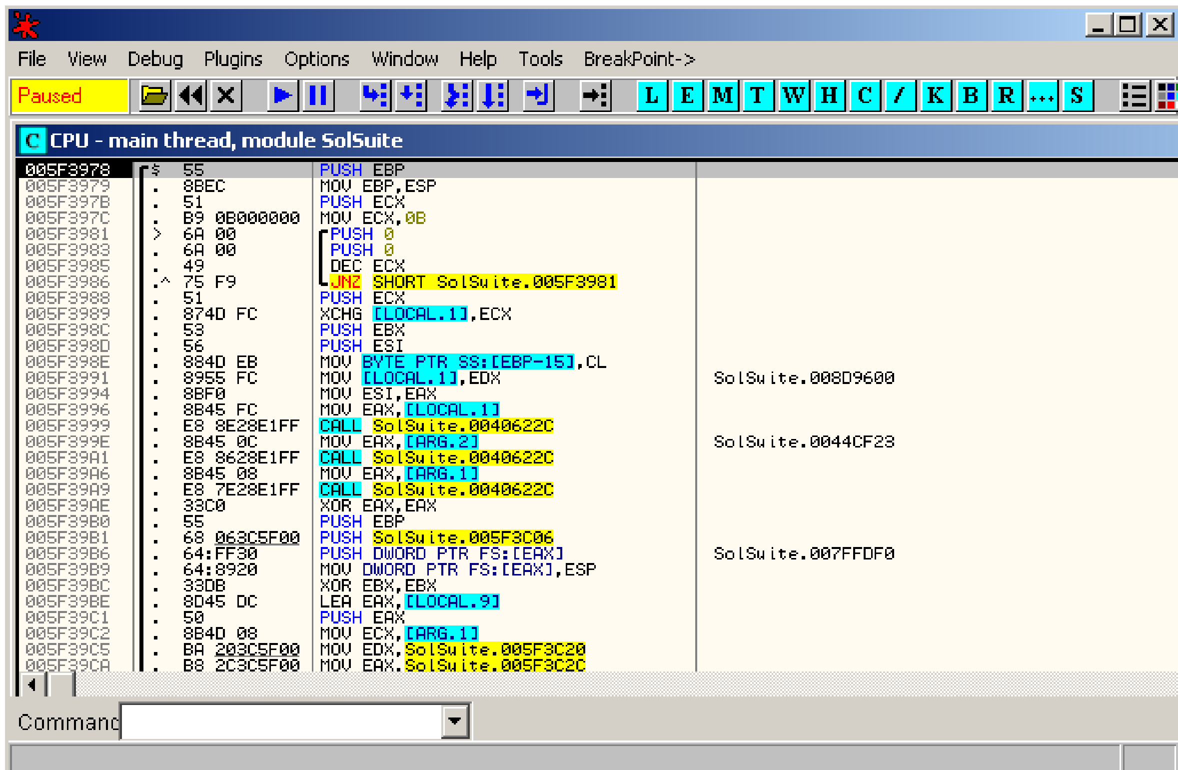


Como el parche no funcionó vamos tener que analizar el código con más profundidad a partir de aquí. Sabemos que este salto está condicionado por la instrucción TEST AL, AL de la

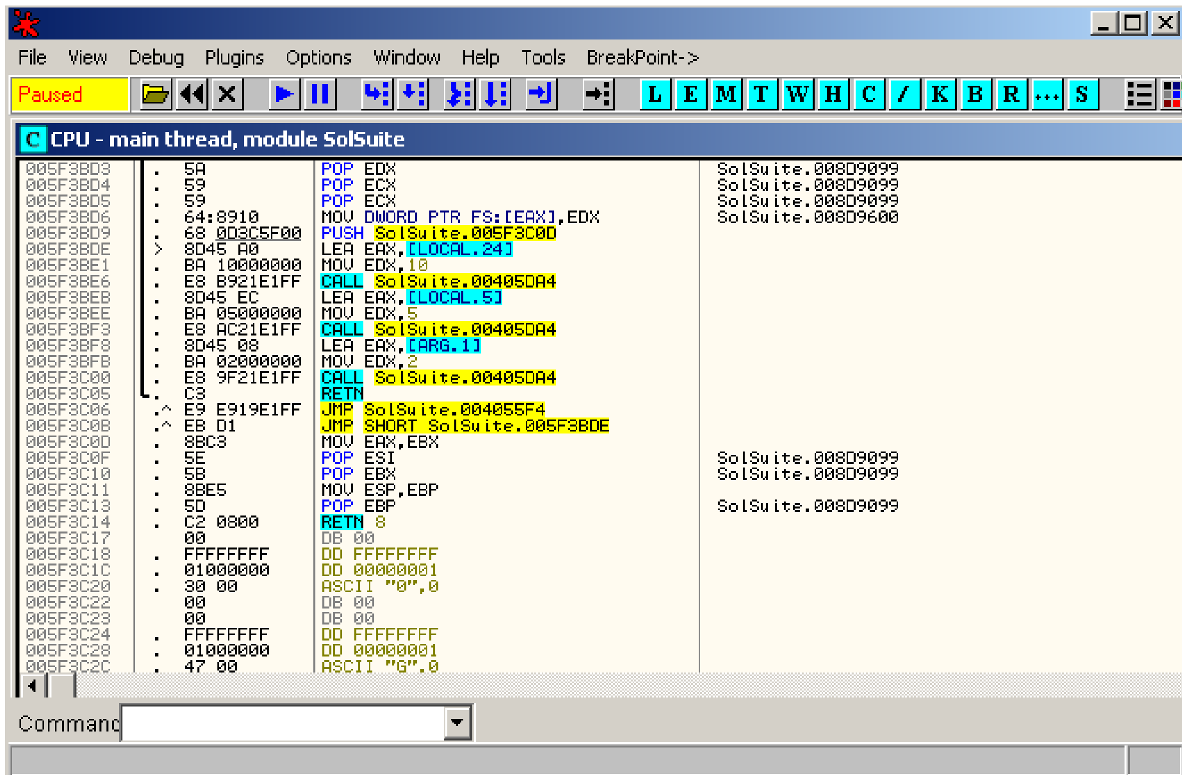
dirección 8D9099. Sabemos también, ya que ejecutamos el código línea por línea pulsando F8, que AL viene determinado por el CALL justo por encima de 8D9094. Pongamos pues un punto de ruptura en ese CALL. Ejecutamos Olly y nos detendremos en el CALL:



Pulsamos F7 para entrar en el CALL. Hay que recordar que debemos volver del CALL con AL = 1 para luego coger el salto:



Como podemos observar se trata de una rutina larga y sin RETN, por lo que bajamos hasta el final de dicha rutina:

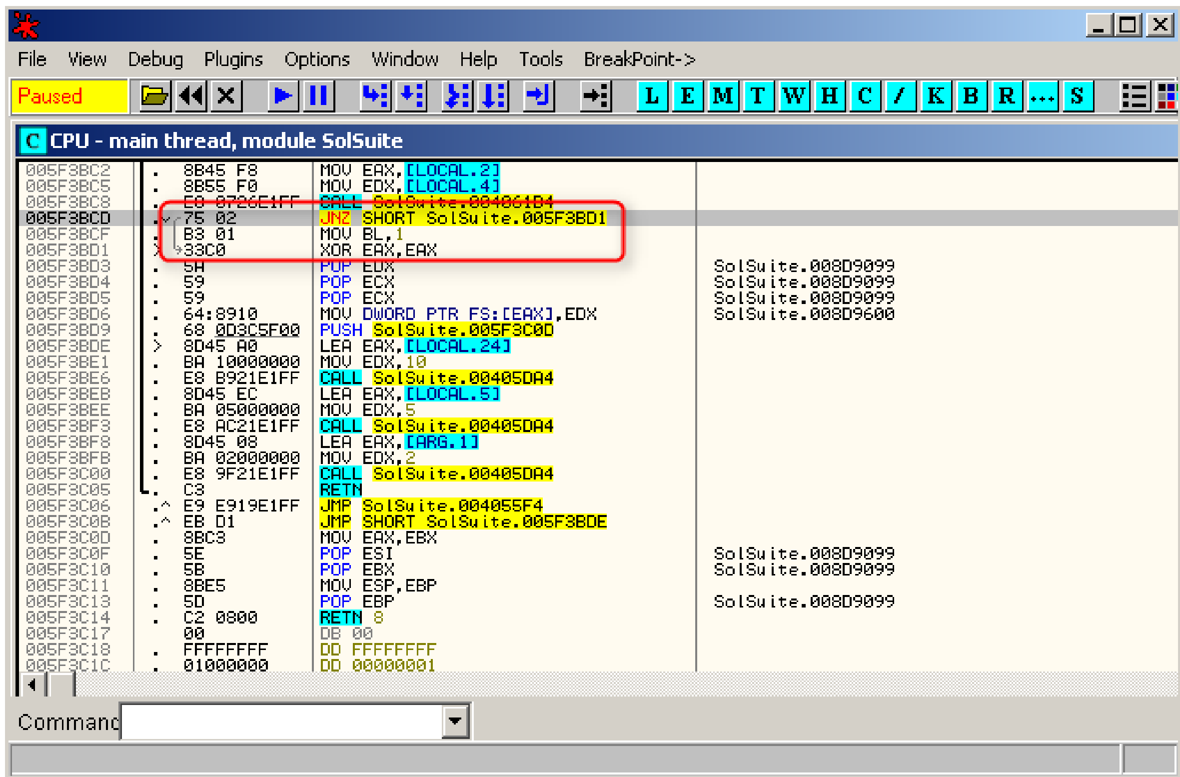


Si nos fijamos bien en el código, en ninguna parte AL se pone igual a uno. Pero hay un sitio donde BL es igual a uno. Lo que Olly nos está diciendo es que la rutina acaba en 5F3C05 y antes de eso BL no se ha movido a AL.

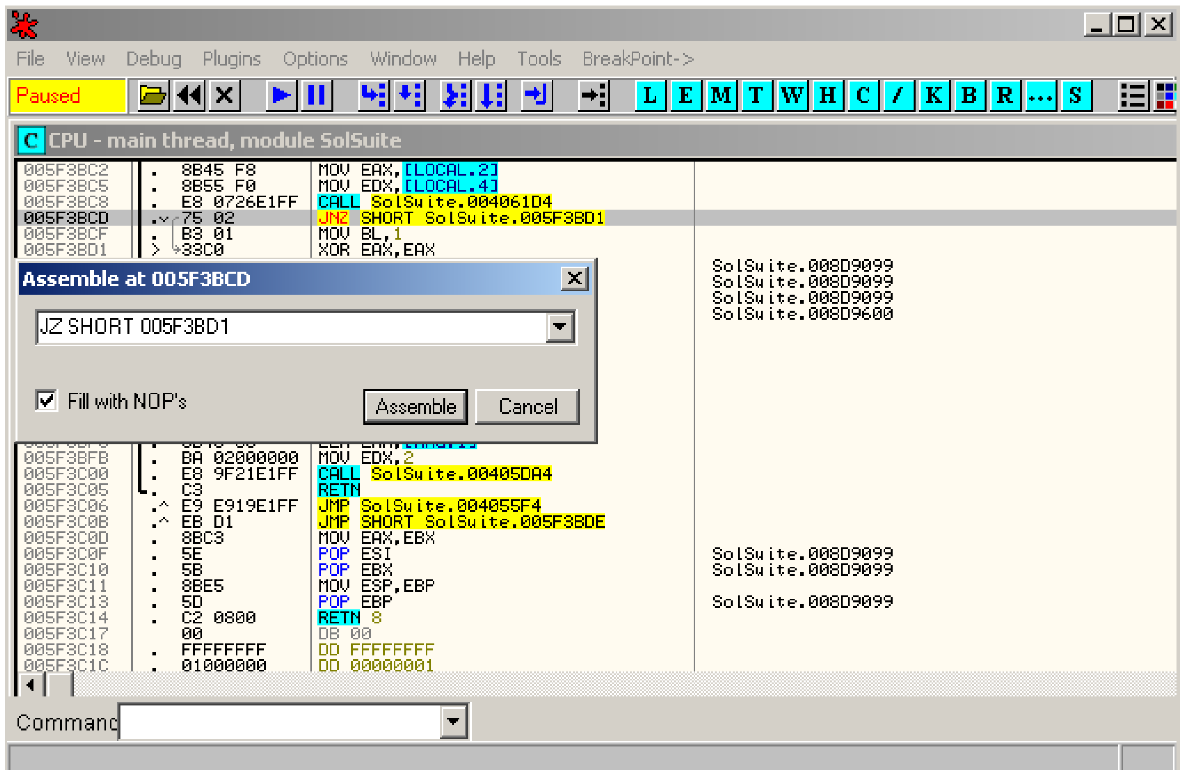
Fijemonos en la dirección 5F3BD9, este PUSH SolSuite.005F3C0D empuja esa dirección en la pila de forma que cuando llegemos a la instrucción RETN nos mandará devuelta a 5F3C0D.

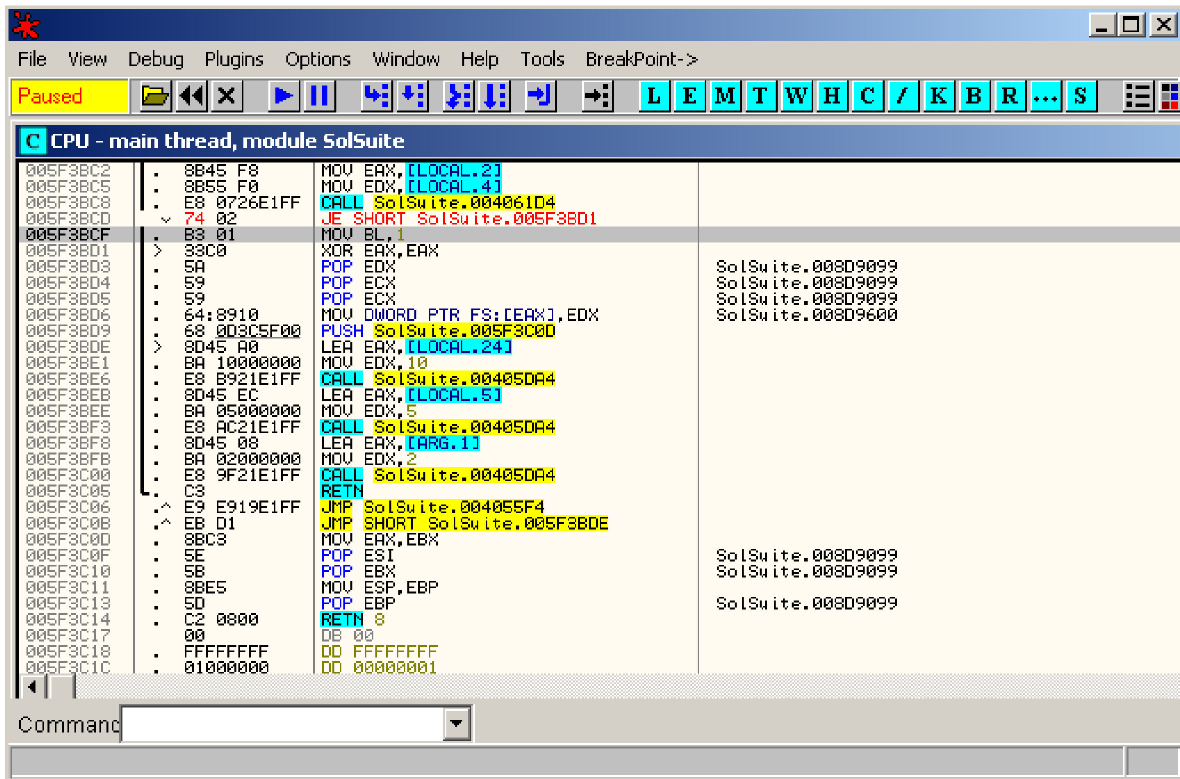
Si nos fijamos en la dirección 5F3C0D vemos la instrucción MOV EAX, EBX (o lo que es lo mismo MOV AL, BL). Si BL = 1 va a ser movido a AL antes de acabar esa rutina.

Sabemos que AL = 0 al salir de la rutina, por lo que debe de tomar el salto en 5F3BCD



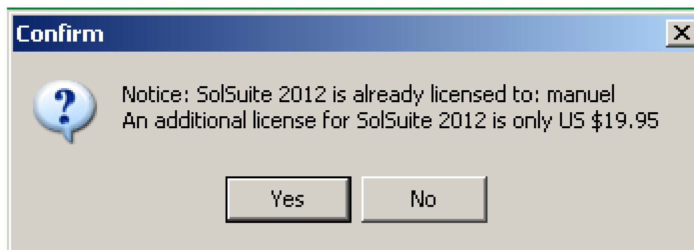
Cambiemos por lo tanto la instrucción JNZ a JZ y guardemos el nuevo ejecutable en el disco.





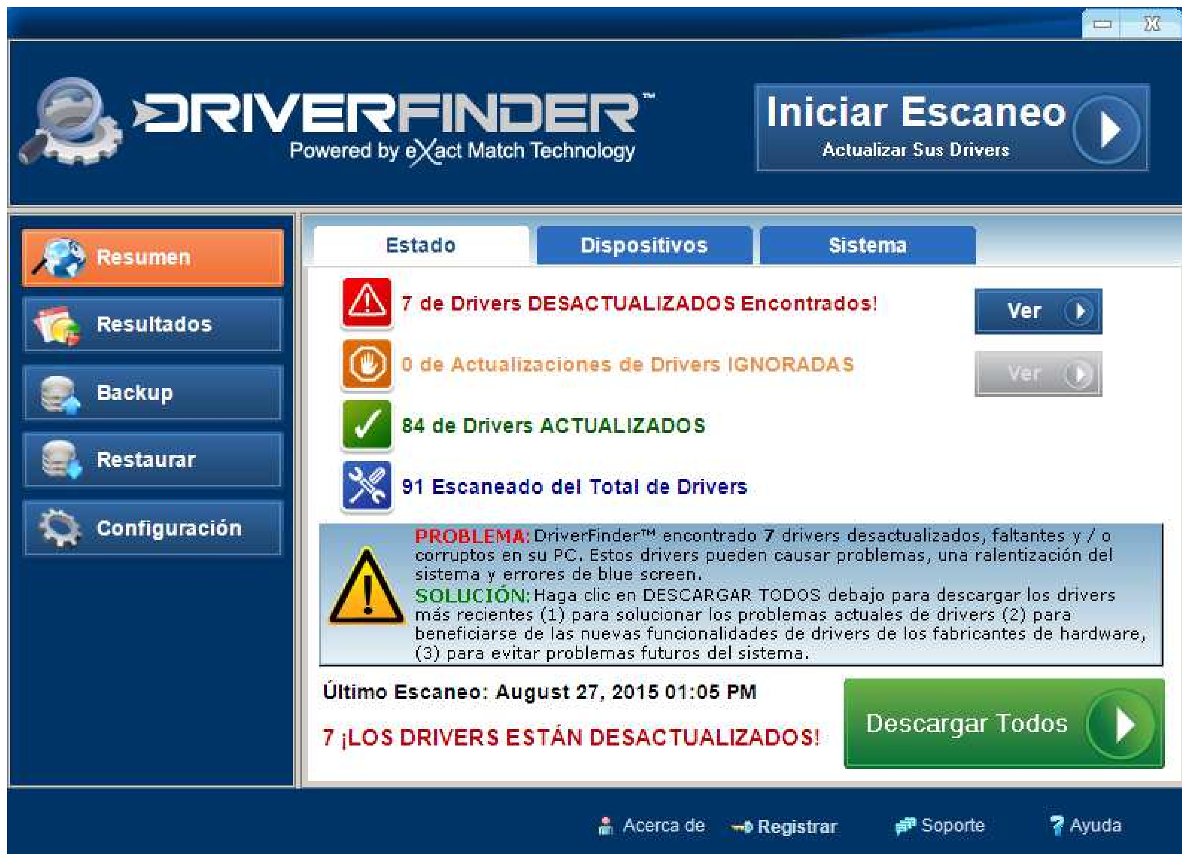
Si ahora ejecutamos el nuevo ejecutable veremos como todo va como la seda.

Si pulsamos sobre el botón "Activate SolSuite" veremos el siguiente mensaje:



8.4 Cracking Driver Finder

Descargamos Driver Finder 2.1 de Internet. Inmediatamente después de instalar la aplicación, esta procede a escanear el equipo en busca de drivers desactualizados.



Hacemos clic en “Registrar” e introducimos un ID y una contraseña:

Activar la licencia

DRIVERFINDER™
Powered by eXact Match Technology

No dejes que tu PC se deterioran. Activar su copia de DriverFinder ahora y actualizar los controladores de dispositivo al instante!

ID de Licencia: 121212121212

Contraseña: manuel

Activar

¿Aún no tiene una licencia?

Registrar

Cerrar

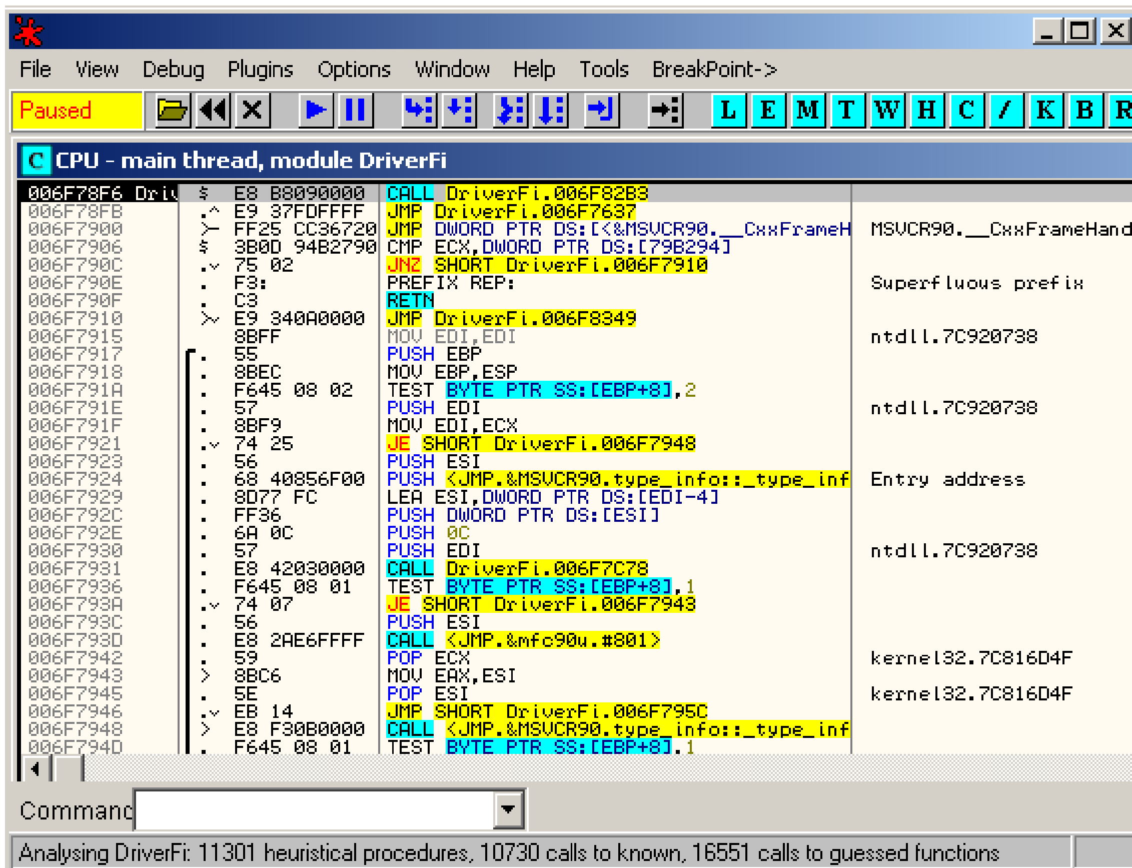
Hacemos clic en "Activar" y nos aparece la siguiente pantalla:

DriverFinder

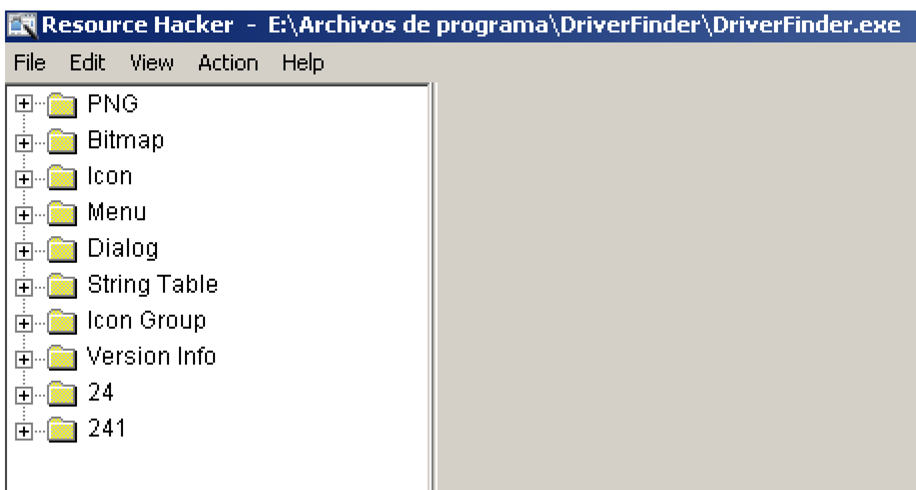
Información de licencia no válida. Por favor revise su ID de Licencia y Contraseña y vuelva a intentarlo, o póngase en contacto con soporte técnico para obtener ayuda.

OK

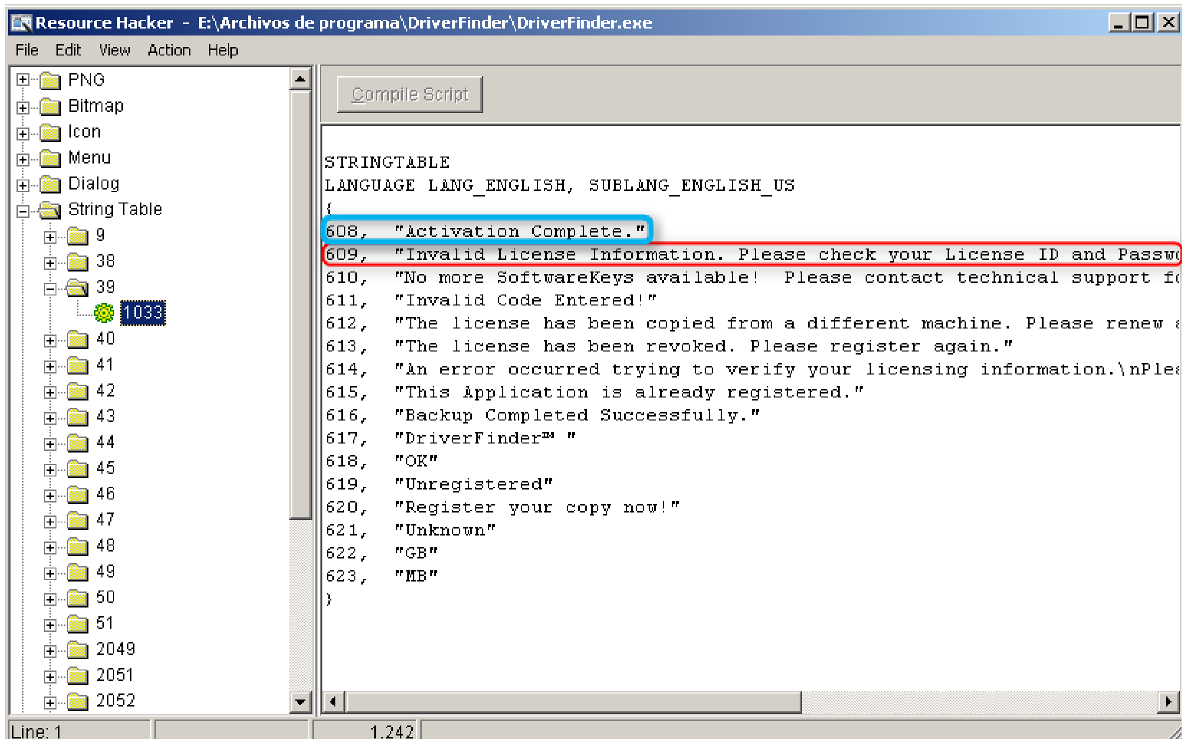
Abrimos Olly y cargamos la aplicación:



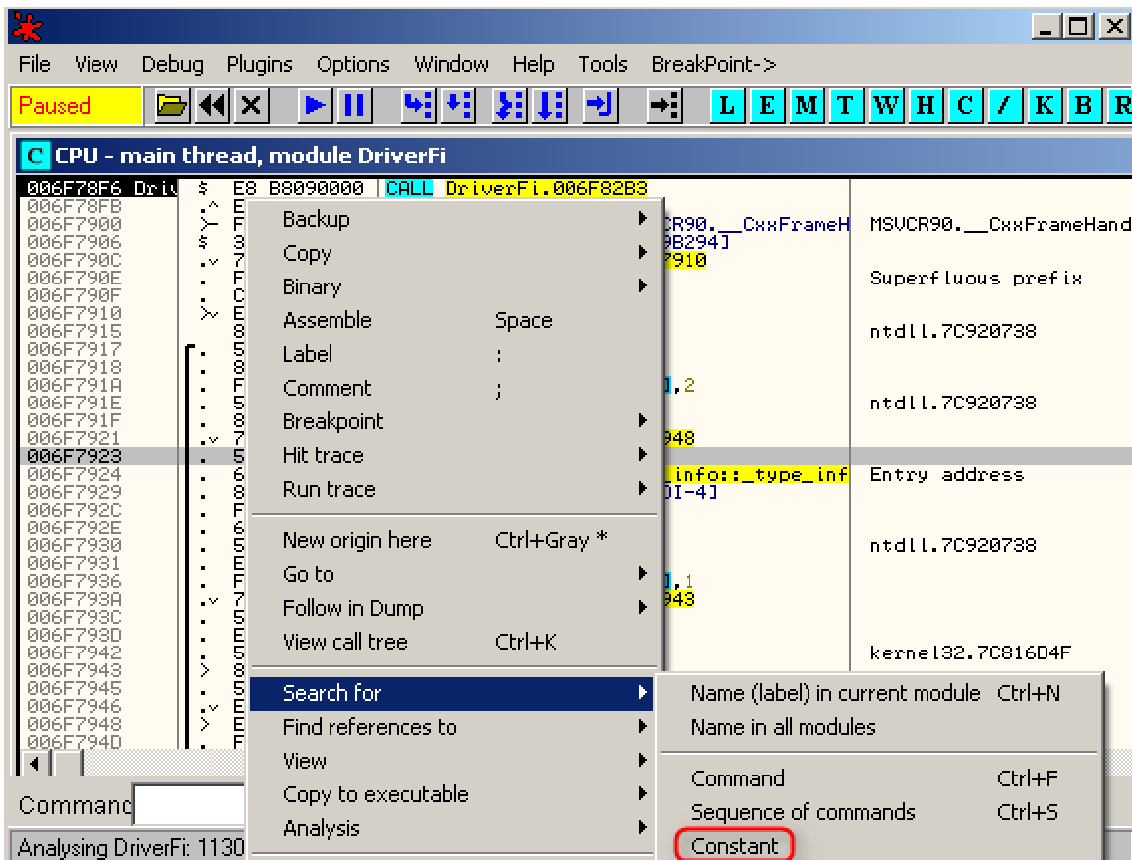
Si ejecutamos la aplicación y buscamos por cadenas de texto no vamos a encontrar al “bad boy”. Por eso vamos a cargar la aplicación en Resource Hacker:



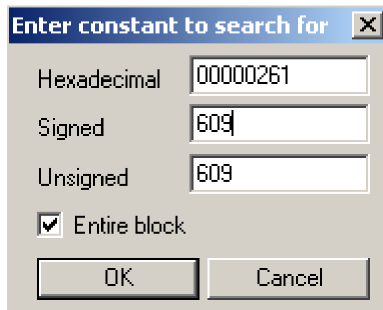
Desplegamos la carpeta “String Table” y seleccionamos el número 39:

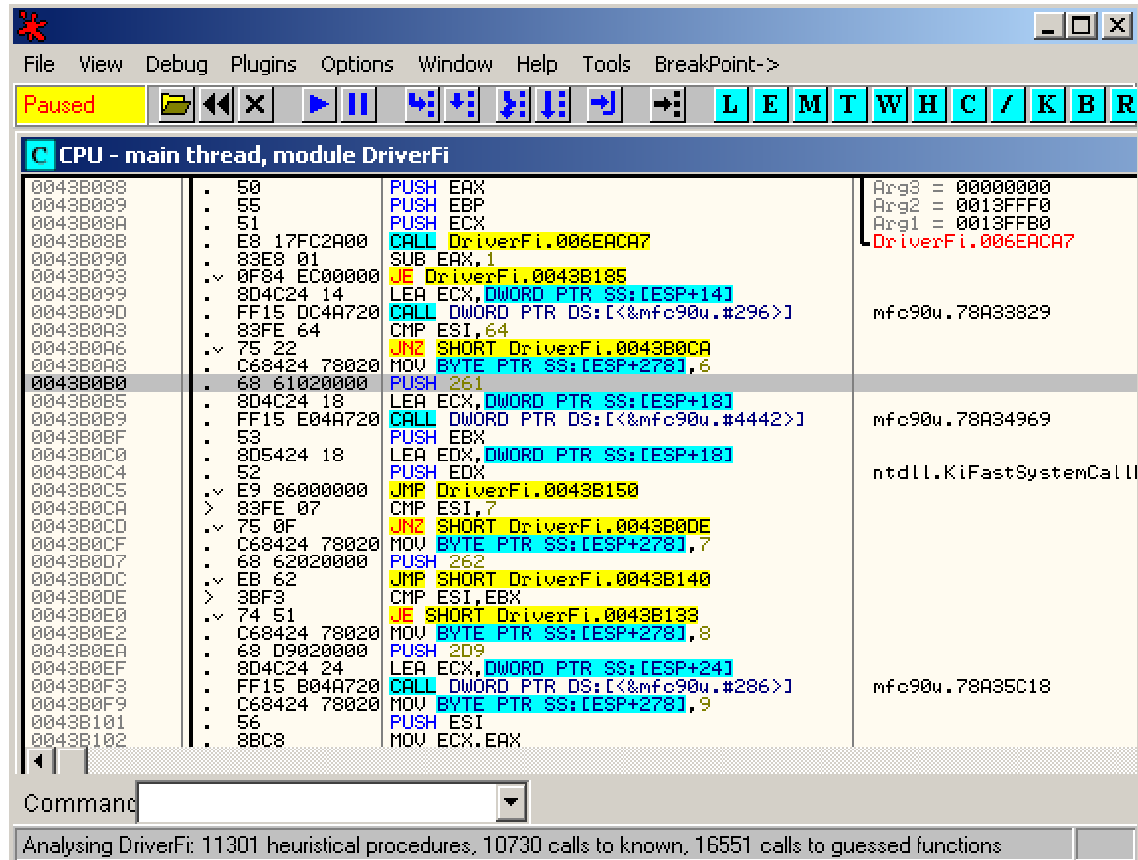


Y vemos que tenemos con el número 209 nuestro “bad boy” y una línea más arriba con el número 608 nuestro “good boy”. Devuelta en Olly hacemos clic con el botón derecho y seleccionamos “Search for” -> “constant”:

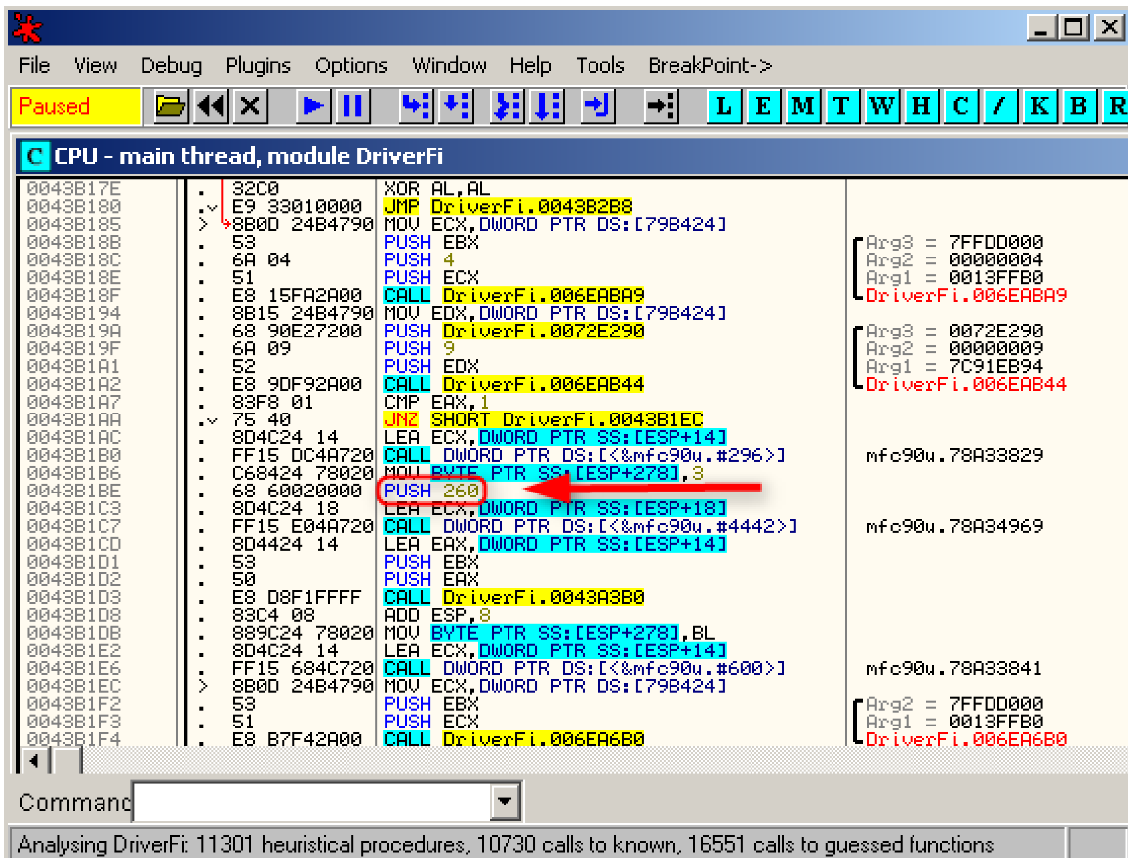


Introducimos el número en el campo “Signed” o “Unsigned” y pulsamos “OK”:

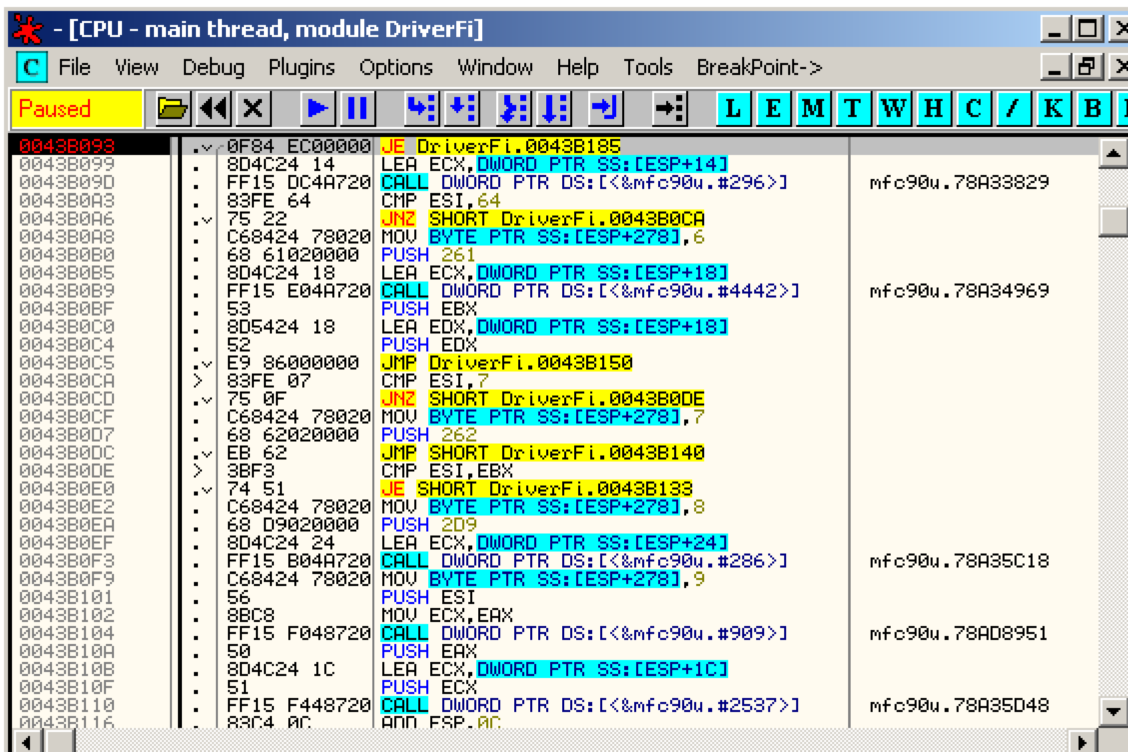




Dos líneas más arriba tenemos una combinación de comparar/saltar. Si nos situamos encima de la instrucción saltar en la dirección 43B0A6 vemos que salta pasando a nuestro “bad boy”, hacia más instrucciones PUSH, todas ellas diferentes formas de “bad boys” y sin llegar a nuestro “good boy”. Ahora bien si marcamos el salto condicional JE en la dirección 43B093 vemos que saltamos pasando todos los “bad boys” que vienen a continuación, llegando a una área donde podemos ver la instrucción PUSH 260 (que en hexadecimal es 608, nuestro “good boy”):

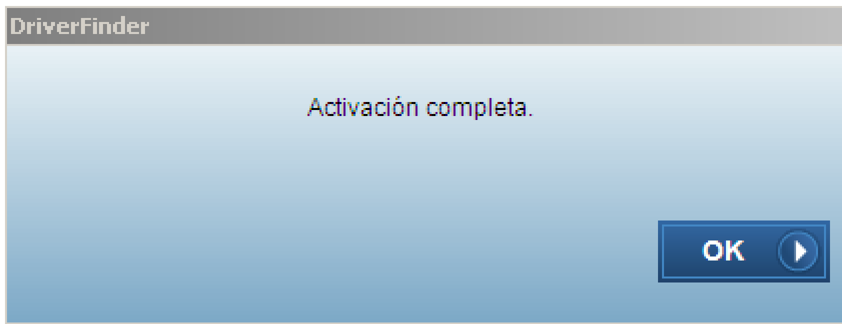


Ponemos un punto de ruptura en el salto condicional, ejecutamos la aplicación, introducimos un ID y contraseña y vemos como Olly se detiene en nuestro punto de ruptura:



Cambiamos el valor de la bandera Z para que tome el salto y pulsamos F8:

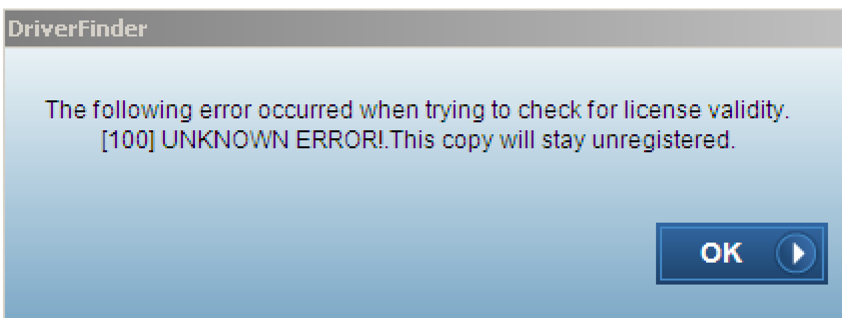
Antes de llegar a la instrucción PUSH 260, vemos que hay otro salto en la dirección 43B1AA. Pulsamos F8 hasta llegar a esa dirección y miramos si Olly va a saltar o no. Como no salta pulsamos F9 para ejecutar la aplicación:



Hemos llegado a nuestro "good boy". Hacemos clic en "OK" y vemos que el botón para registrarse ha desaparecido:

Si hacemos clic en "Acerca de" vemos que tenemos una licencia válida:

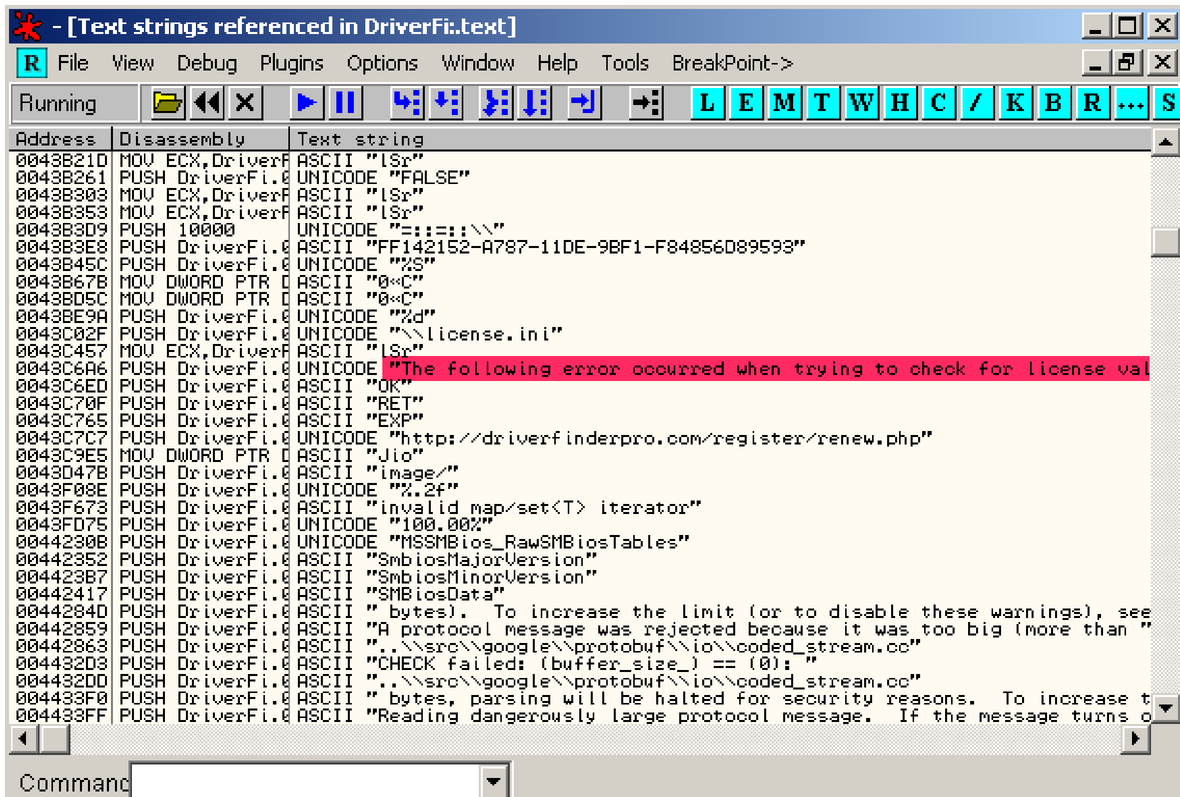
Reiniciamos la aplicación en Olly para comprobar que todo funcione bien.



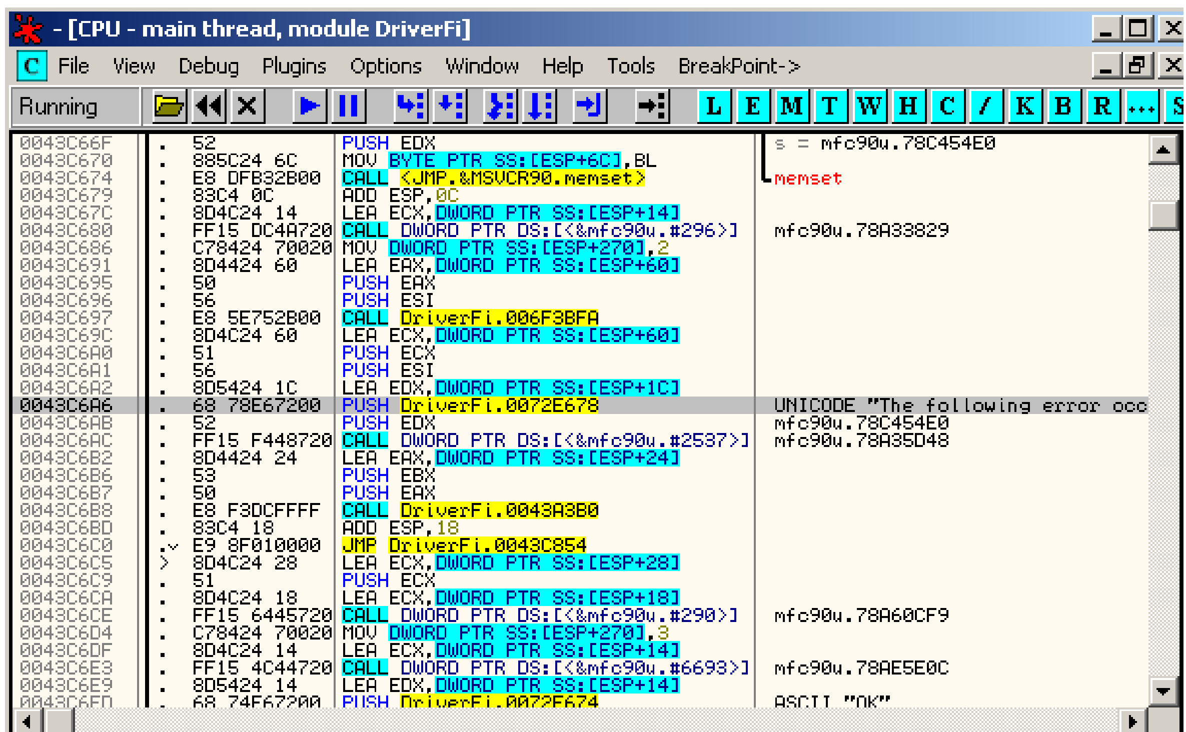
Y vemos que como siempre hay algo que va mal… Hacemos clic en "OK" y todo vuelve a su sitio original:

Tiene que haber otra comprobación que necesita ser parcheada.

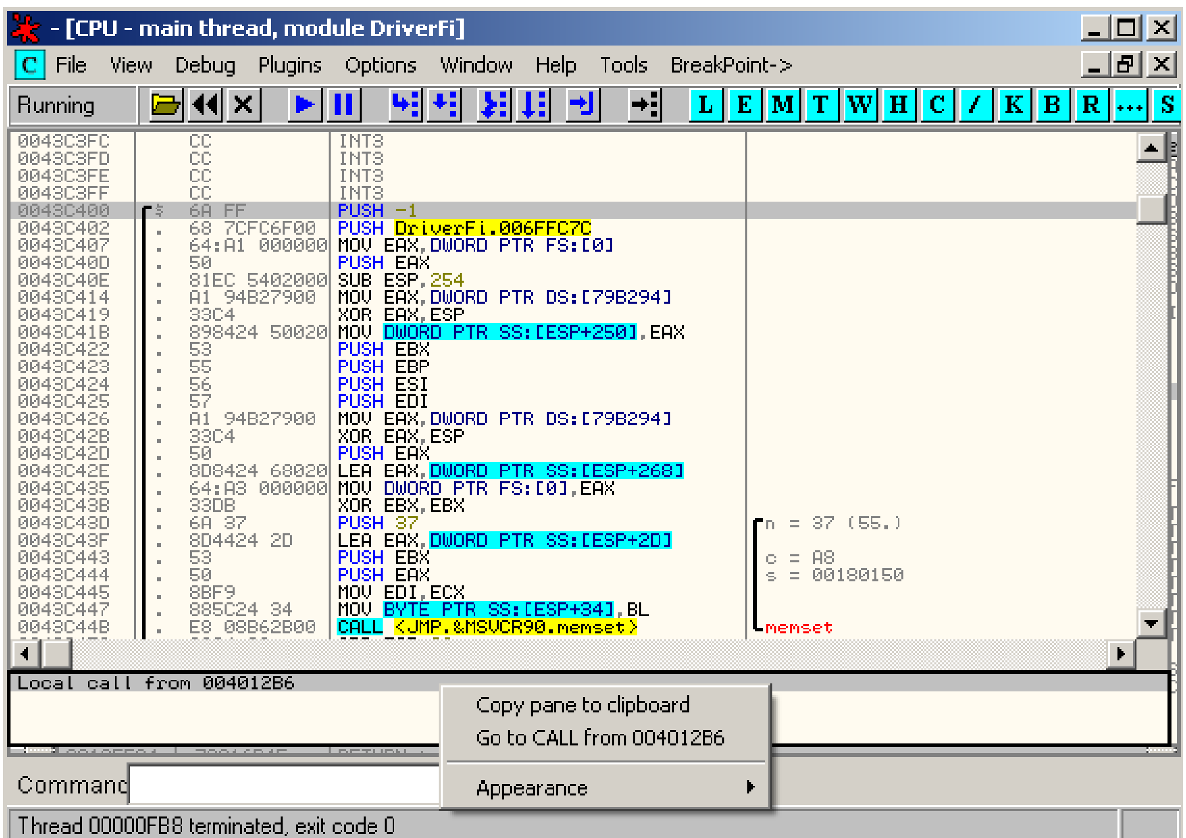
Busquemos esta vez por cadenas de texto en Olly y nos encontramos con esto:



Hacemos doble clic:



Vamos a subir hasta el principio de la rutina para averiguar como evitar esta comprobación:



Vemos que la rutina se llama desde 4012B6. Hacemos clic con el botón derecho y seleccionamos “Go to CALL from 4012B6”:

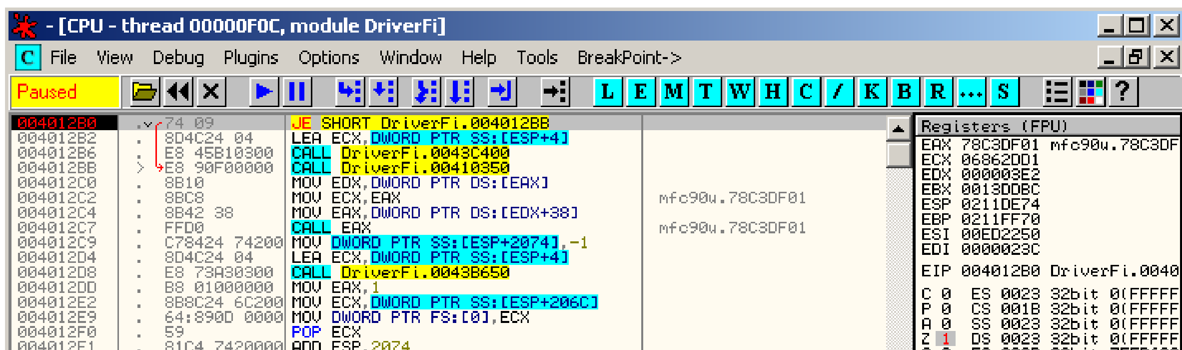
```
- [CPU - main thread, module DriverFi]
File  View  Debug  Plugins  Options  Window  Help  Tools  BreakPoint->
Running
0040127D  .  33C4          XOR EAX,ESP
0040127F  .  50            PUSH EAX
00401280  .  8D8424 6C200  LEA EAX,DWORD PTR SS:[ESP+206C]
00401287  .  64:A3 000000  MOV DWORD PTR FS:[0],EAX
0040128D  .  6A 00         PUSH 0                          Arg1 = 00000000
0040128F  .  8D4C24 08     LEA ECX,DWORD PTR SS:[ESP+8]
00401293  .  E8 78AA0300   CALL DriverFi.0043BD10          DriverFi.0043BD10
00401298  .  C78424 74200  MOV DWORD PTR SS:[ESP+2074],0
004012A3  .  6A 00         PUSH 0                          Arg1 = 00000000
004012A5  .  8D4C24 08     LEA ECX,DWORD PTR SS:[ESP+8]
004012A9  .  E8 D2A00300   CALL DriverFi.0043B380          DriverFi.0043B380
004012AE  .  84C0          TEST AL,AL
004012B0  .v 74 09         JE SHORT DriverFi.004012BB
004012B2  .  8D4C24 04     LEA ECX,DWORD PTR SS:[ESP+4]
004012B6  .  E8 45B10300   CALL DriverFi.0043C400
004012BB  >  E8 90F00000   CALL DriverFi.00410350
004012C0  .  8B10          MOV EDX,DWORD PTR DS:[EAX]
004012C2  .  8BC8          MOV ECX,EAX
004012C4  .  8B42 38       MOV EAX,DWORD PTR DS:[EDX+38]
004012C7  .  FFD0          CALL EAX
004012C9  .  C78424 74200  MOV DWORD PTR SS:[ESP+2074],-1
004012D4  .  8D4C24 04     LEA ECX,DWORD PTR SS:[ESP+4]
004012D8  .  E8 73A30300   CALL DriverFi.0043B650
004012DD  .  B8 01000000   MOV EAX,1
004012E2  .  8B8C24 6C200  MOV ECX,DWORD PTR SS:[ESP+206C]
004012E9  .  64:890D 0000  MOV DWORD PTR FS:[0],ECX
004012F0  .  59            POP ECX                         USER32.77D1919B
004012F1  .  81C4 7420000  ADD ESP,2074
004012F7  .  C3            RETN
```

La pregunta que se nos plantea ahora es como poder evitar este CALL. Vemos que hay un JE dos líneas más arriba que pasa saltando por nuestro CALL. Pongamos un punto de ruptura y ejecutamos la aplicación:

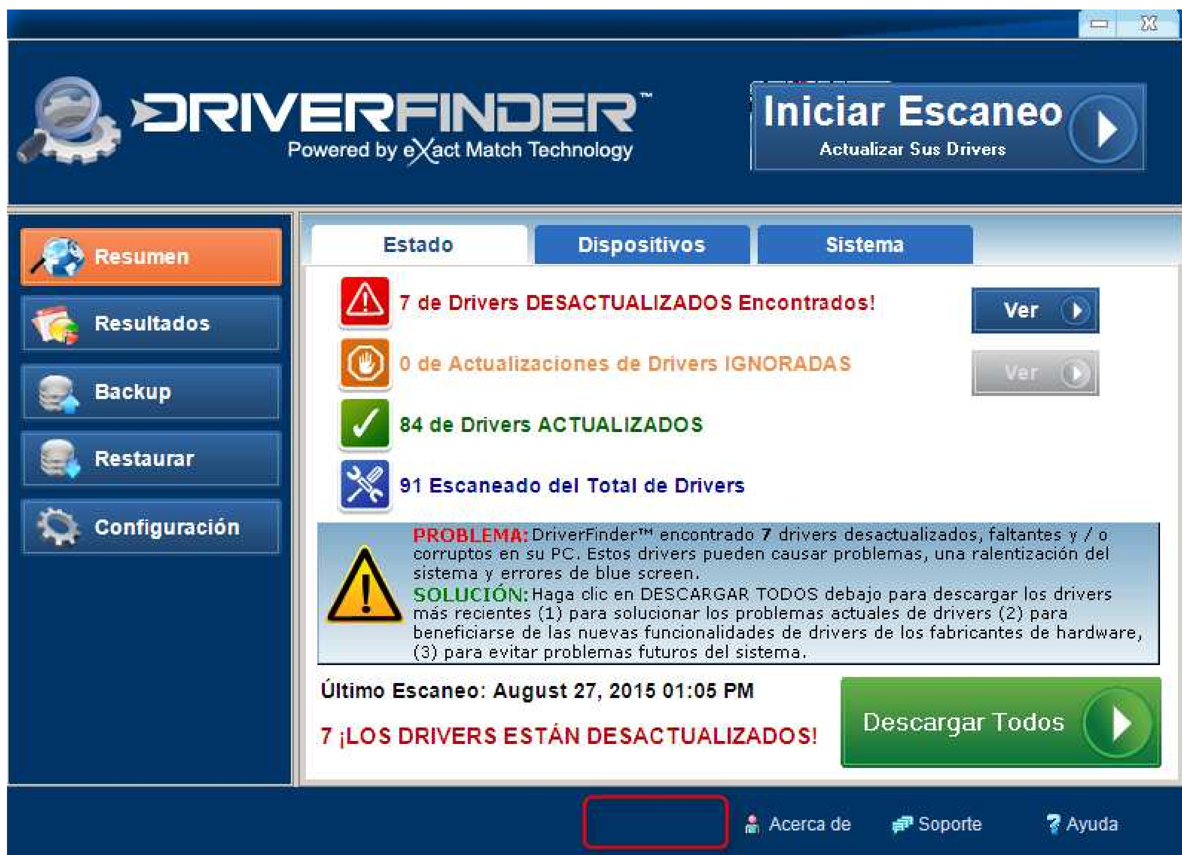
```
- [CPU - main thread, module DriverFi]
File  View  Debug  Plugins  Options  Window  Help  Tools  BreakPoint->
Running
0040127D  .  33C4          XOR EAX,ESP
0040127F  .  50            PUSH EAX
00401280  .  8D8424 6C200  LEA EAX,DWORD PTR SS:[ESP+206C]
00401287  .  64:A3 000000  MOV DWORD PTR FS:[0],EAX
0040128D  .  6A 00         PUSH 0                          Arg1 = 00000000
0040128F  .  8D4C24 08     LEA ECX,DWORD PTR SS:[ESP+8]
00401293  .  E8 78AA0300   CALL DriverFi.0043BD10          DriverFi.0043BD10
00401298  .  C78424 74200  MOV DWORD PTR SS:[ESP+2074],0
004012A3  .  6A 00         PUSH 0                          Arg1 = 00000000
004012A5  .  8D4C24 08     LEA ECX,DWORD PTR SS:[ESP+8]
004012A9  .  E8 D2A00300   CALL DriverFi.0043B380          DriverFi.0043B380
004012AE  .  84C0          TEST AL,AL
004012B0  .v 74 09         JE SHORT DriverFi.004012BB
004012B2  .  8D4C24 04     LEA ECX,DWORD PTR SS:[ESP+4]
004012B6  .  E8 45B10300   CALL DriverFi.0043C400
004012BB  >  E8 90F00000   CALL DriverFi.00410350
004012C0  .  8B10          MOV EDX,DWORD PTR DS:[EAX]
004012C2  .  8BC8          MOV ECX,EAX
004012C4  .  8B42 38       MOV EAX,DWORD PTR DS:[EDX+38]
004012C7  .  FFD0          CALL EAX
004012C9  .  C78424 74200  MOV DWORD PTR SS:[ESP+2074],-1
004012D4  .  8D4C24 04     LEA ECX,DWORD PTR SS:[ESP+4]
004012D8  .  E8 73A30300   CALL DriverFi.0043B650
004012DD  .  B8 01000000   MOV EAX,1
004012E2  .  8B8C24 6C200  MOV ECX,DWORD PTR SS:[ESP+206C]
004012E9  .  64:890D 0000  MOV DWORD PTR FS:[0],ECX
004012F0  .  59            POP ECX                         USER32.77D1919B
004012F1  .  81C4 7420000  ADD ESP,2074
004012F7  .  C3            RETN
```

```
- [CPU - thread 00000F0C, module DriverFi]
File  View  Debug  Plugins  Options  Window  Help  Tools  BreakPoint->
Paused
004012B0  .v 74 09         JE SHORT DriverFi.004012BB
004012B2  .  8D4C24 04     LEA ECX,DWORD PTR SS:[ESP+4]
004012B6  .  E8 45B10300   CALL DriverFi.0043C400
004012BB  >  E8 90F00000   CALL DriverFi.00410350
004012C0  .  8B10          MOV EDX,DWORD PTR DS:[EAX]
004012C2  .  8BC8          MOV ECX,EAX                     mfc90u.78C3DF01
004012C4  .  8B42 38       MOV EAX,DWORD PTR DS:[EDX+38]
004012C7  .  FFD0          CALL EAX                        mfc90u.78C3DF01
```

Vemos que Olly no salta. Cambiemos el valor de la bandera Z para que salte:

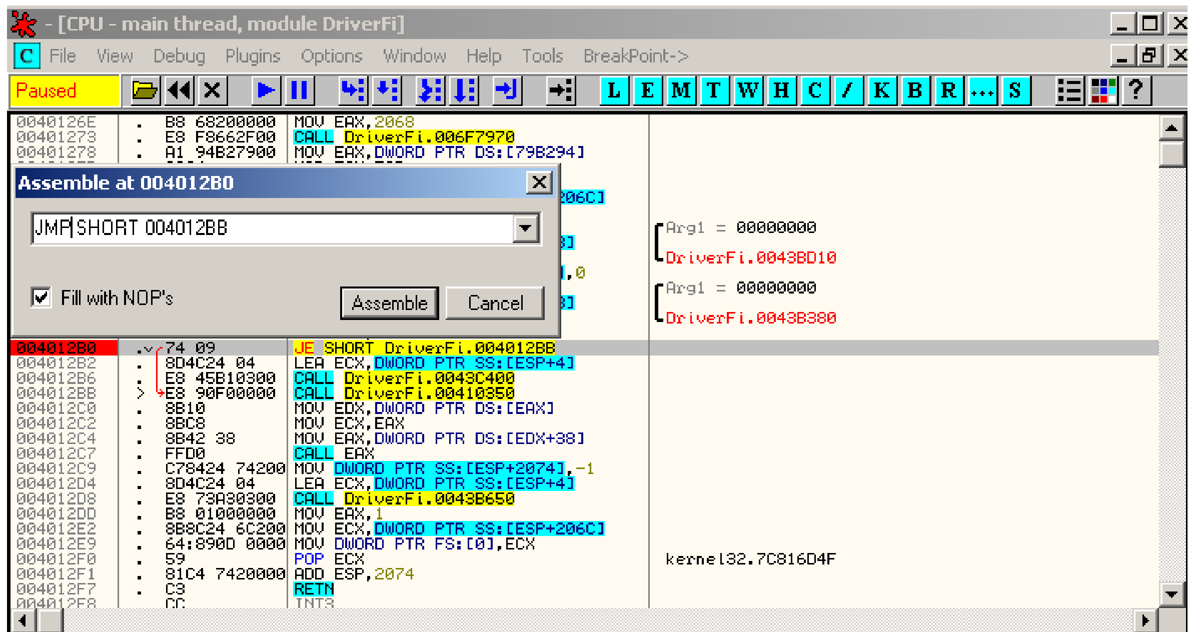


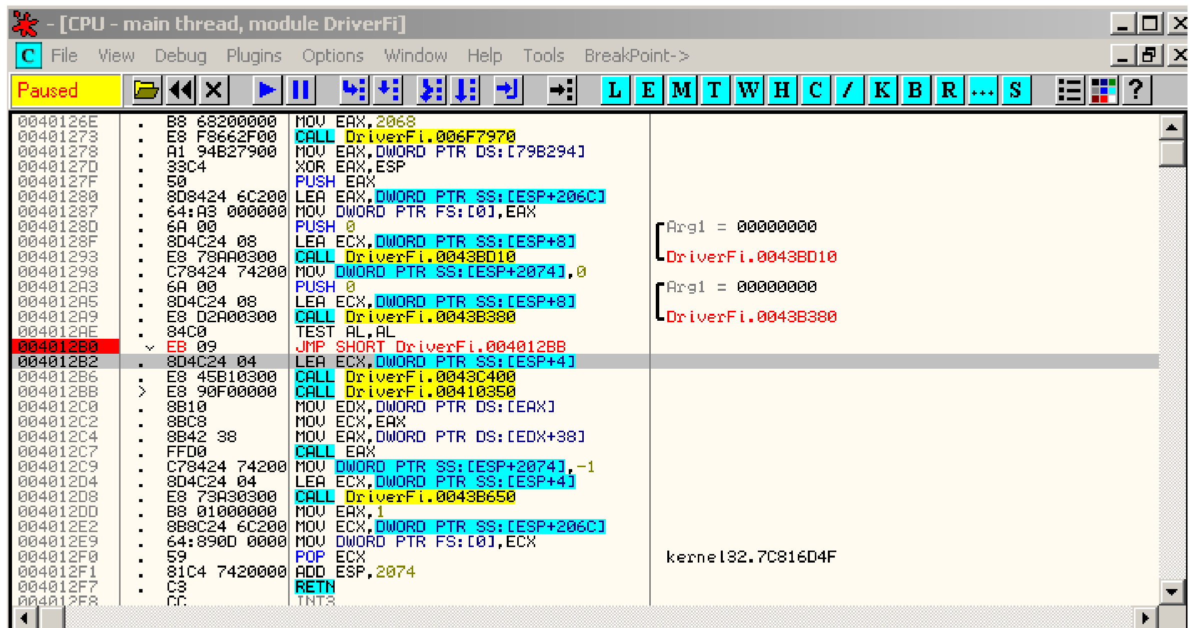
Pulsamos F9:



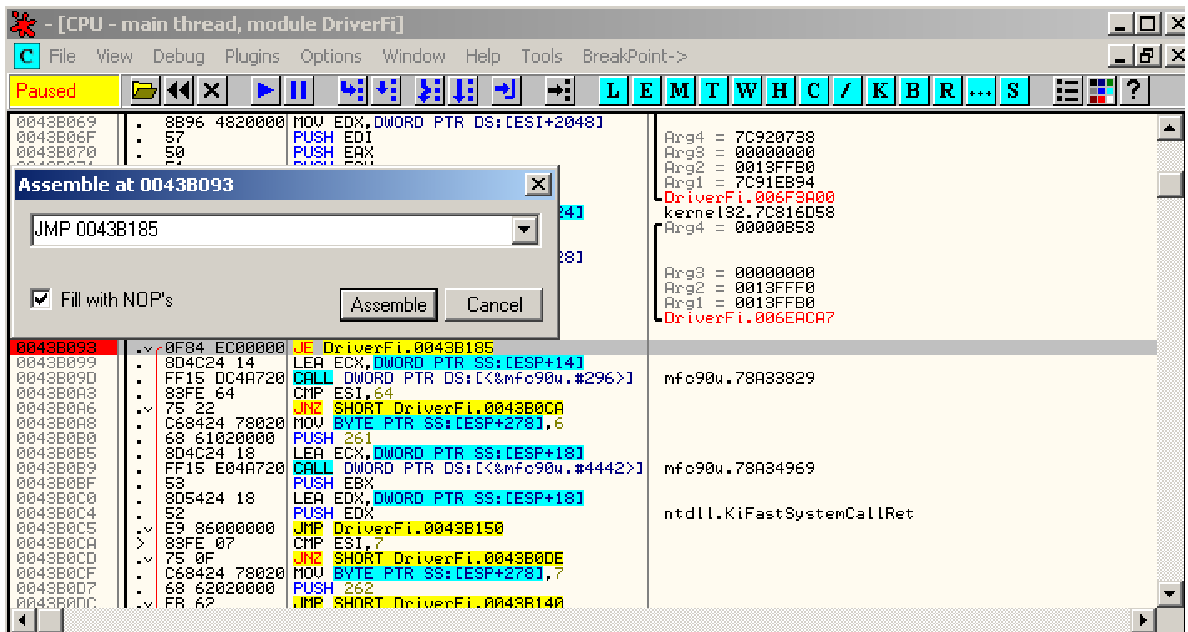
Y volvemos a estar registrados.

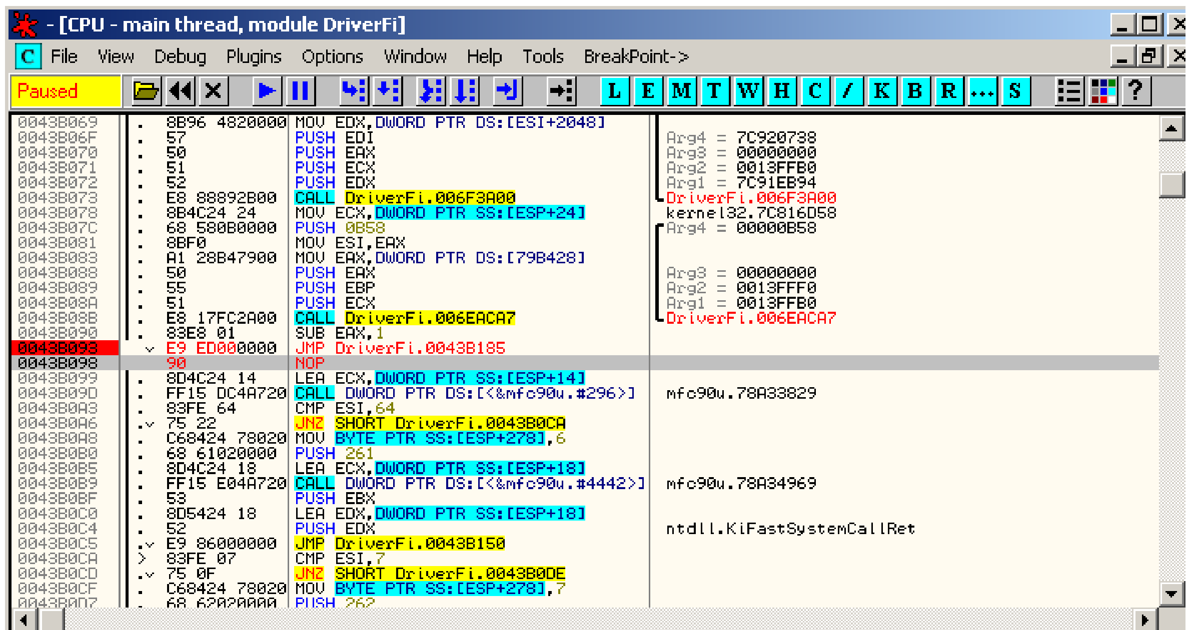
A continuación hacemos que los parches sean permanentes. Empezamos por el último, cambiamos la instrucción JE por JMP:





Hacemos lo mismo con el primer punto de ruptura:

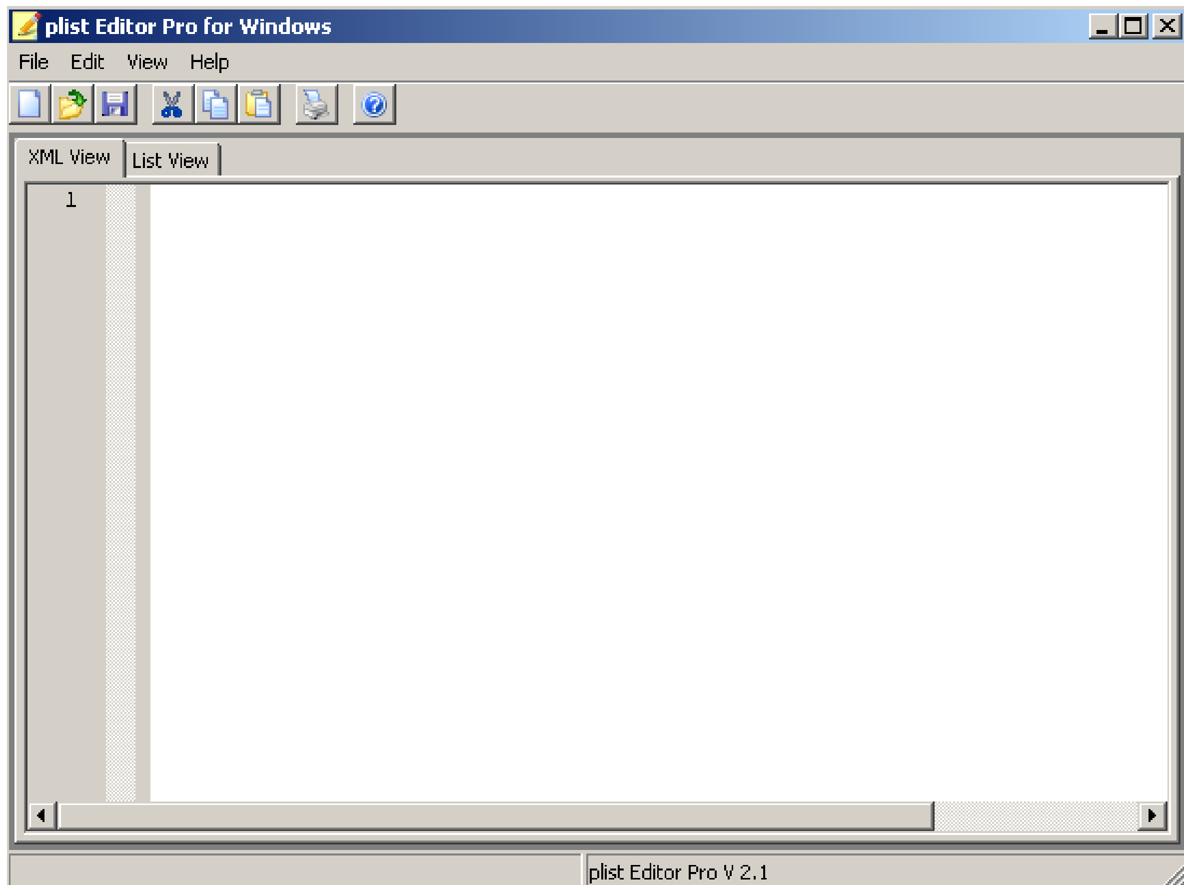




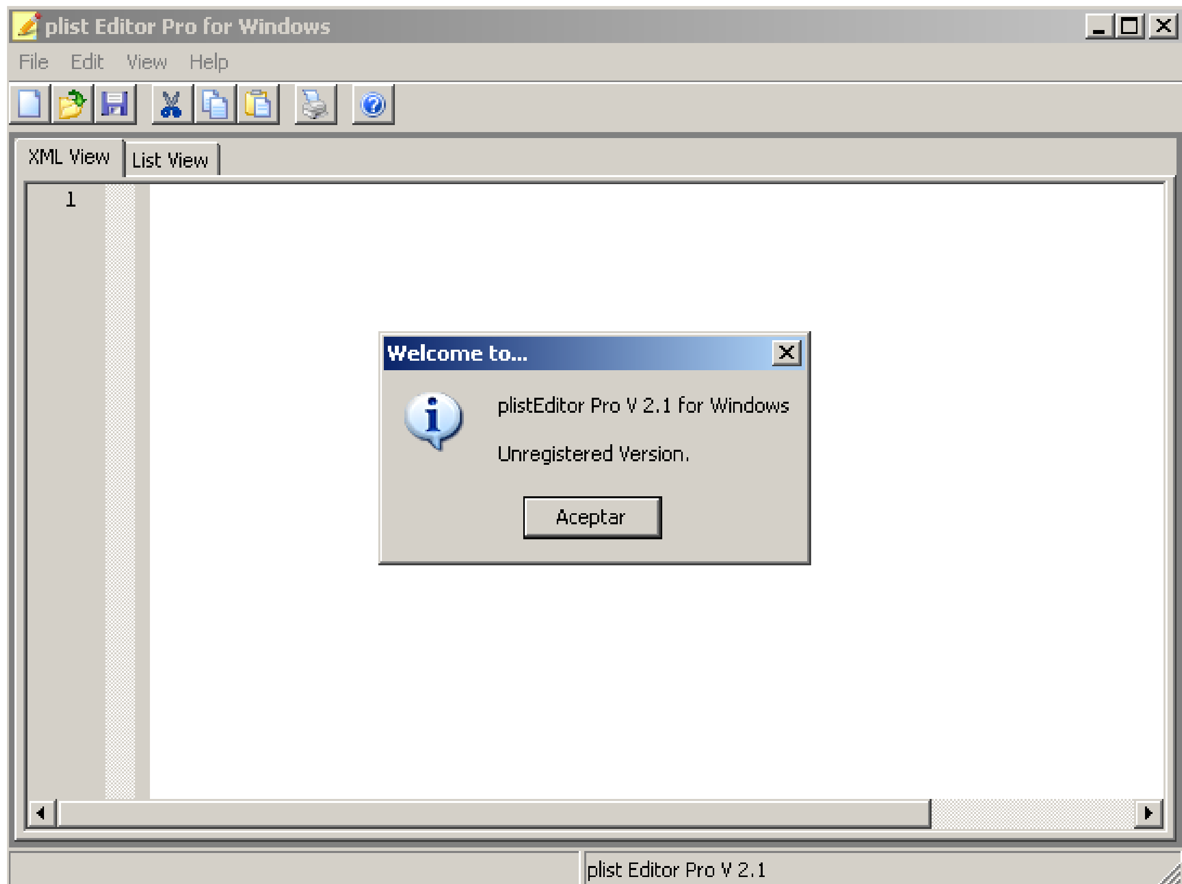
Guardamos los cambios a un ejecutable nuevo y listo.

8.5 Desembalaje rápido de PEcompact

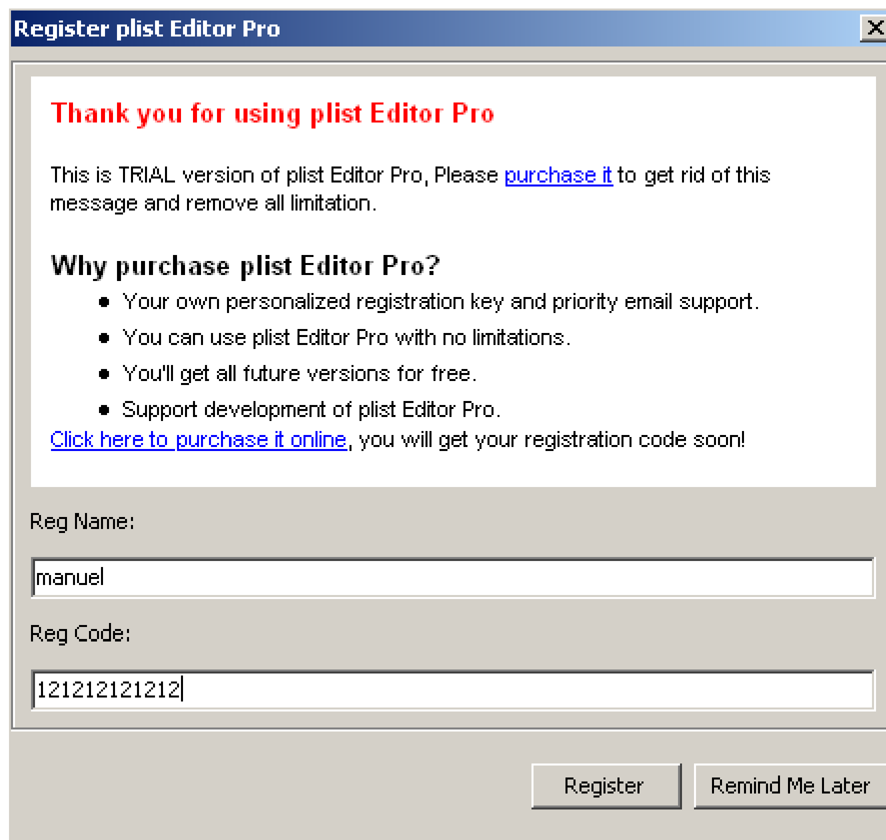
Descargamos PlistEditor Pro 2.0.0 de internet, instalamos la aplicación y la ejecutamos:



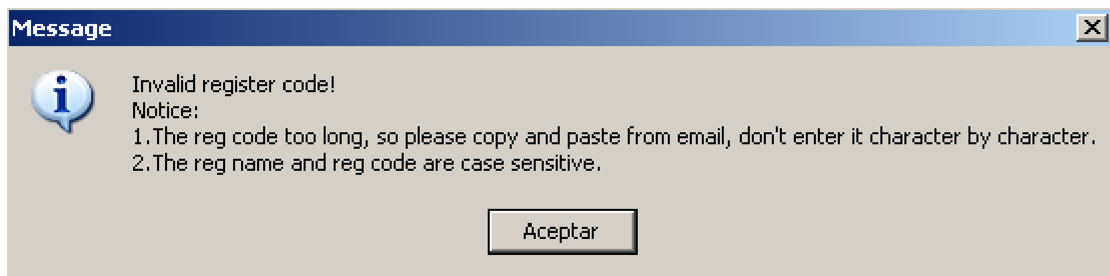
Seleccionamos la pestaña "Help" -> "About"



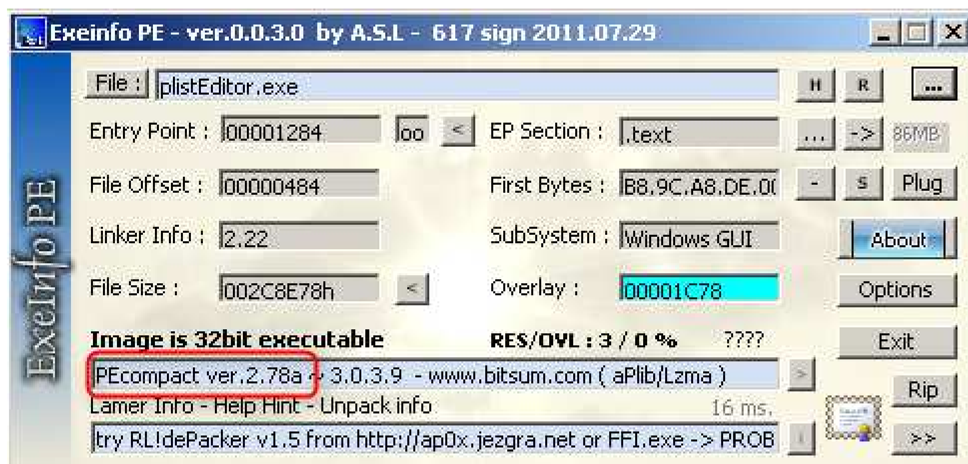
A continuación seleccionamos “Help” -> “Register…” e introducimos nuestros datos:



Pulsamos en “Register” y nos sale el siguiente mensaje:

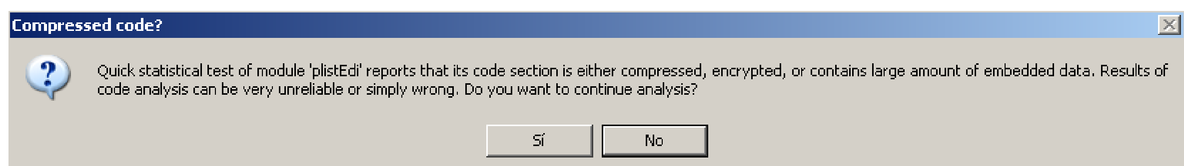


Esto es todo lo que necesitamos por el momento. Vamos a ver si la aplicación está empaquetada. Para ello la cargamos en ExeinfoPE:

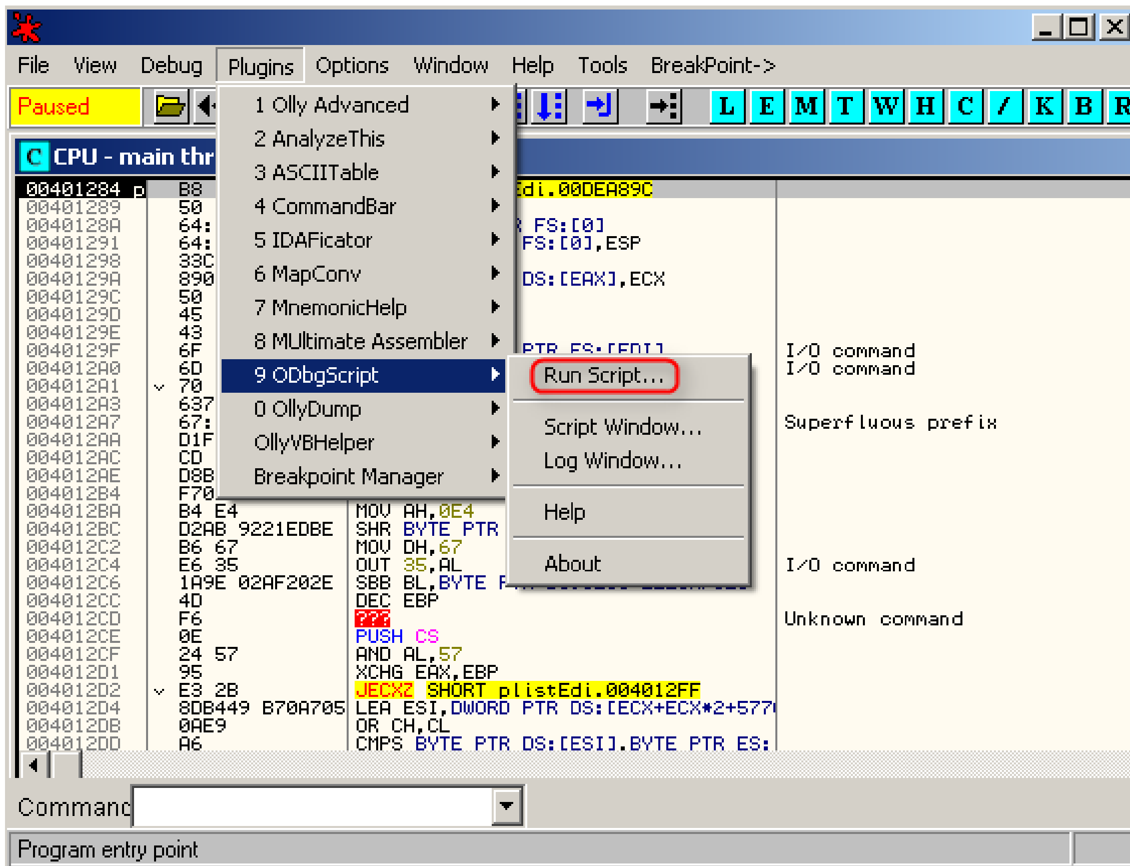


Vemos que está empaquetada con la versión 2.78a hata 3.0.3.9, por lo que tenemos que desempaqueter primero la aplicación.

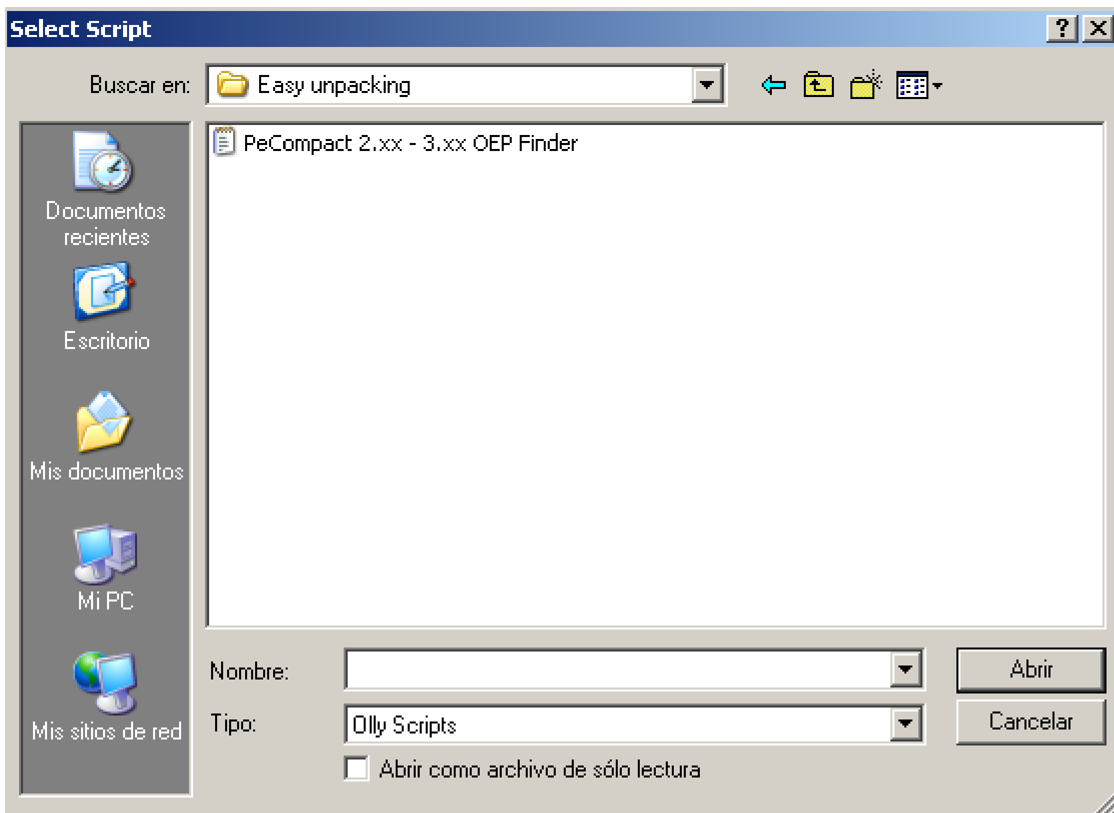
Abrimos Olly y cargamos la aplicación. A los pocos segundos nos sale el siguiente mensaje:

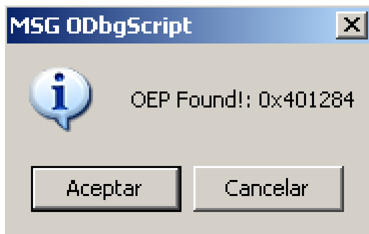


Seleccionamos “No”. Vamos a la pestaña Plugins y seleccionamos “ODbgScript” -> “Run Script…”:

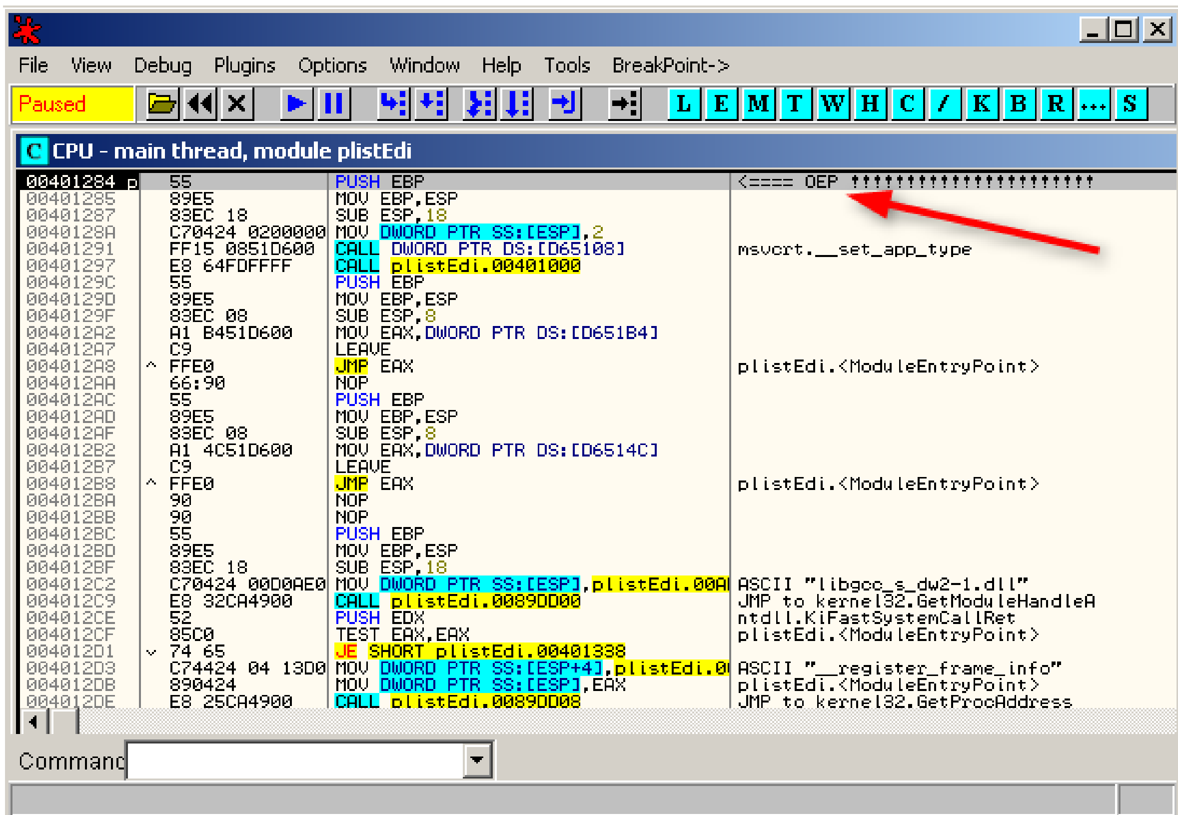


Seleccionamos el script y hacemos clic en “Abrir”:

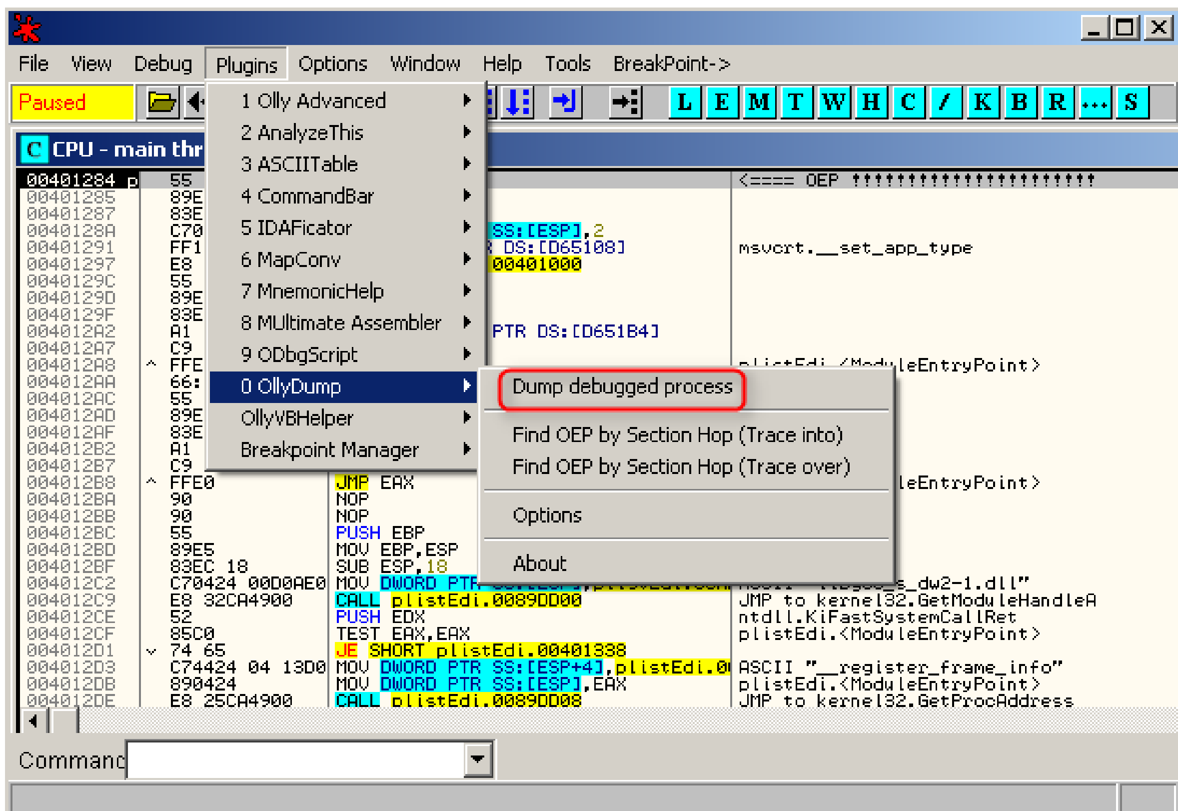




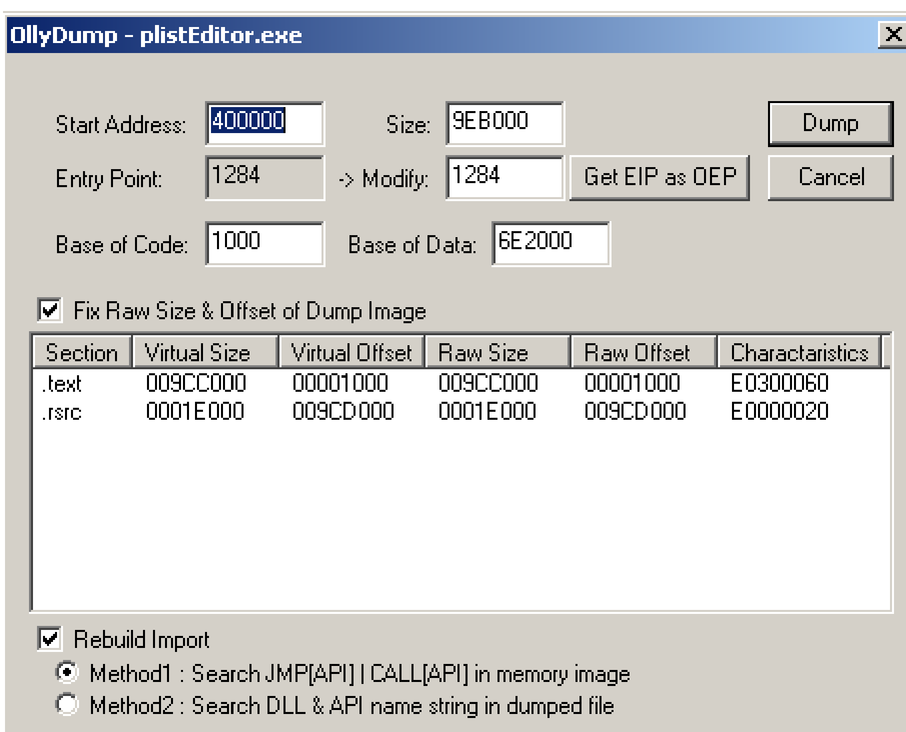
Hacemos clic en ‘Aceptar’ y regresaremos a Olly justo en OEP:



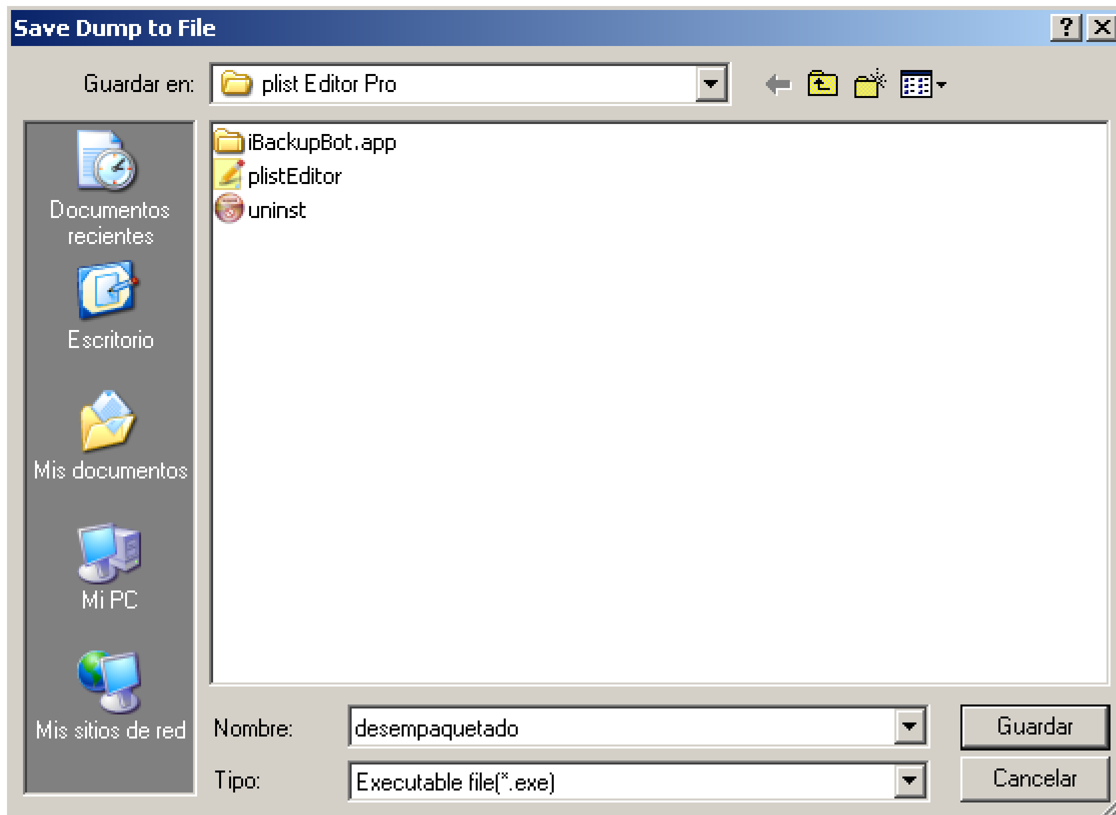
Seleccionamos la pestaña “Plugin” -> “OllyDump” -> “Dump debugged process”:



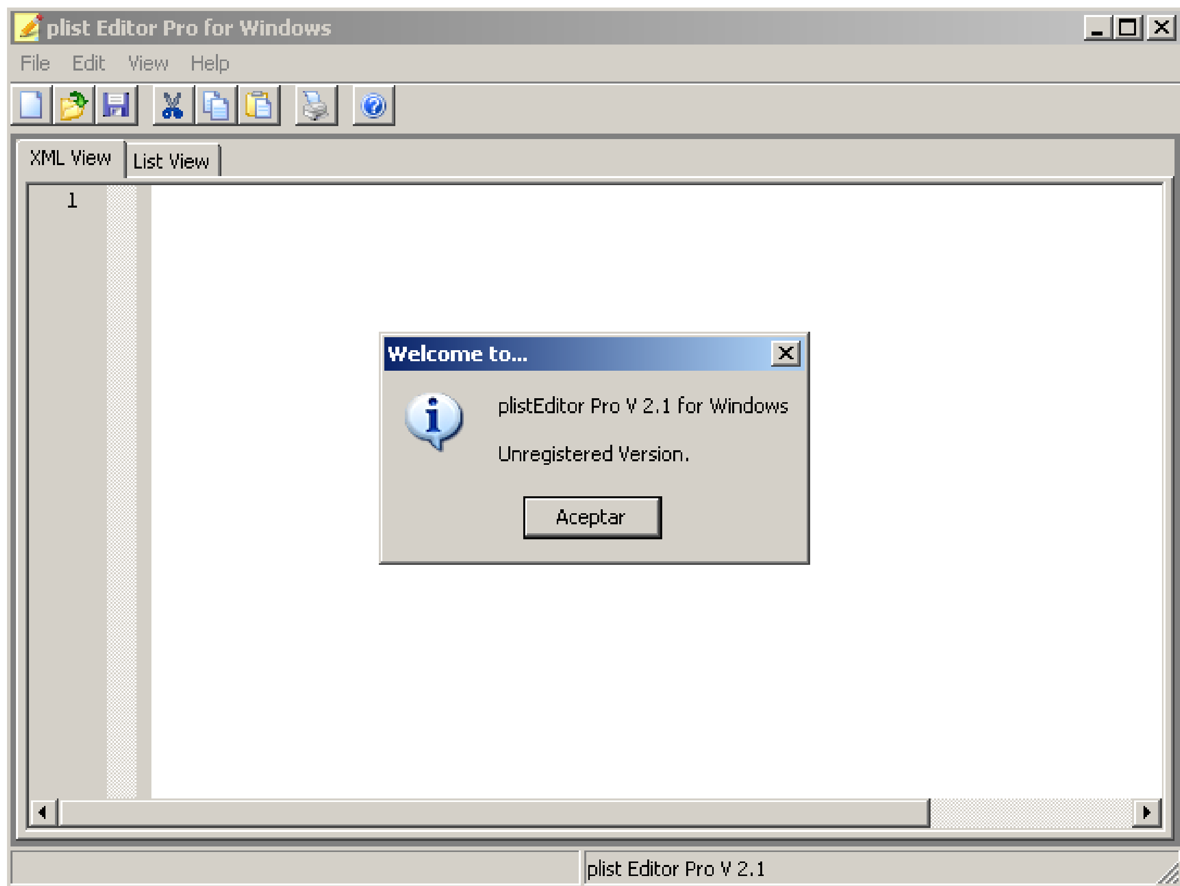
Y se abre la siguiente pantalla:



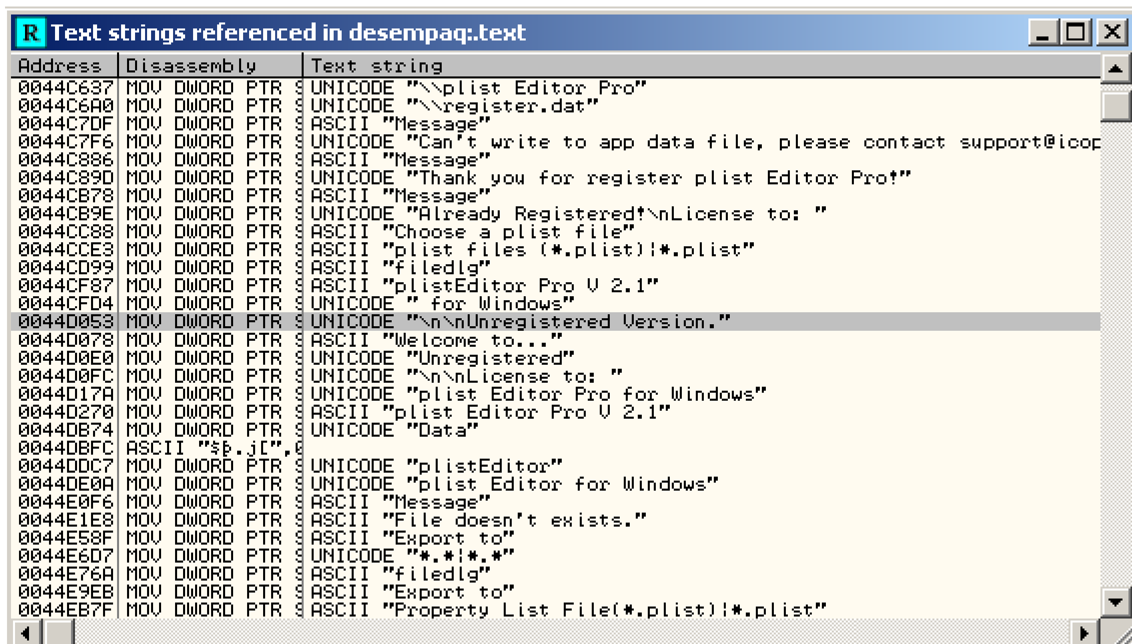
En este caso no tenemos que cambiar nada, solo hacemos clic en “Dump”. Guardamos el nuevo archivo y ya hemos desempaquetado (o desembalado) PEcompact.



Ejecutamos la aplicación desempaquetada en Olly. Vemos que carga correctamente. Pulsamos F9 y sale la pantalla de inicio. Seleccionamos la pestaña “Help” -> “About”:



Buscamos la cadena de texto “Unregistered Version”:



Hacemos doble clic sobre la línea seleccionada:

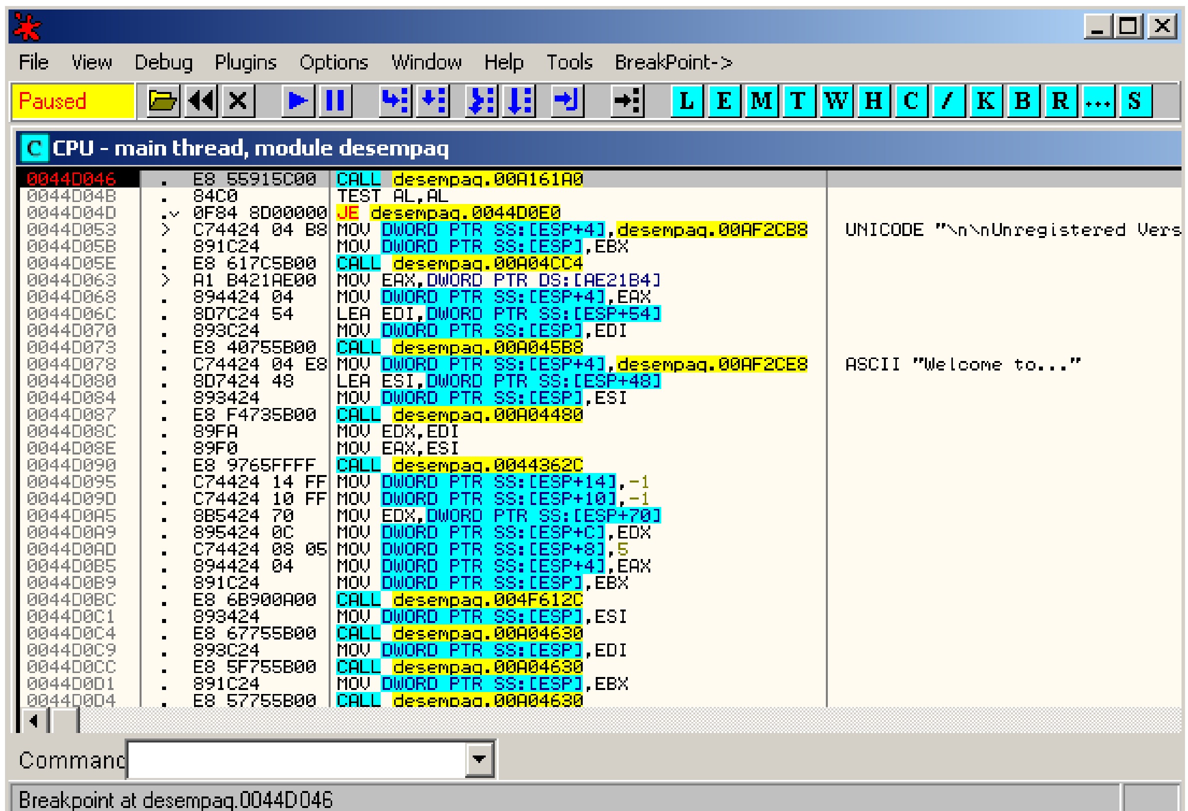
```
File  View  Debug  Plugins  Options  Window  Help  Tools  BreakPoint->
Paused
CPU - main thread, module desempaq
0044D021     90             NOP
0044D022  .  66:90          NOP
0044D024  .  57             PUSH EDI                                   ntdll.7C920738
0044D025  .  56             PUSH ESI
0044D026  .  53             PUSH EBX
0044D027  .  83EC 60        SUB ESP,60
0044D02A  .  8D5C24 3C      LEA EBX,DWORD PTR SS:[ESP+3C]
0044D02E  .  C74424 04 01   MOV DWORD PTR SS:[ESP+4],1
0044D036  .  891C24         MOV DWORD PTR SS:[ESP],EBX
0044D039  .  E8 2AFFFFFF    CALL desempaq.0044CF68
0044D03E  .  50             PUSH EAX
0044D03F  .  C70424 F8D4D   MOV DWORD PTR SS:[ESP],desempaq.00D
0044D046  .  E8 55915C00    CALL desempaq.00A161A0
0044D04B  .  84C0           TEST AL,AL
0044D04D  .v 0F84 8D000000  JE desempaq.0044D0E0
0044D053  >  C74424 04 B8   MOV DWORD PTR SS:[ESP+4],desempaq.0   UNICODE "\n\nUnregistered Version."
0044D05B  .  891C24         MOV DWORD PTR SS:[ESP],EBX
0044D05E  .  E8 617C5B00    CALL desempaq.00A04CC4
0044D063  >  A1 B421AE00    MOV EAX,DWORD PTR DS:[AE21B4]
0044D068  .  894424 04      MOV DWORD PTR SS:[ESP+4],EAX
0044D06C  .  8D7C24 54      LEA EDI,DWORD PTR SS:[ESP+54]
0044D070  .  893C24         MOV DWORD PTR SS:[ESP],EDI              ntdll.7C920738
0044D073  .  E8 40755B00    CALL desempaq.00A045B8
0044D078  .  C74424 04 E8   MOV DWORD PTR SS:[ESP+4],desempaq.0   ASCII "Welcome to..."
0044D080  .  8D7424 48      LEA ESI,DWORD PTR SS:[ESP+48]
0044D084  .  893424         MOV DWORD PTR SS:[ESP],ESI
0044D087  .  E8 F4735B00    CALL desempaq.00A04480
0044D08C  .  89FA           MOV EDX,EDI                             ntdll.7C920738
0044D08E  .  89F0           MOV EAX,ESI
0044D090  .  E8 9765FFFF    CALL desempaq.0044362C
0044D095  .  C74424 14 FF   MOV DWORD PTR SS:[ESP+14],-1
0044D09D  .  C74424 10 FF   MOV DWORD PTR SS:[ESP+10],-1
Command
Program entry point
```

Vemos una combinación de comparar/saltar justo antes del “bad boy”. Si nos situamos encima del salto en la dirección 44D04D vemos que nos lleva al “good boy”.

Si ponemos un punto de ruptura antes y después del CALL en la dirección 44D046 veremos que el valor de AL depende de este CALL.

Podríamos parchear solo el salto, pero esto solo eliminaría el cuadro de dialogo de “About”. Por el contrario si entramos en el CALL esto eliminará todas las comprobaciones que son usadas en el CALL.

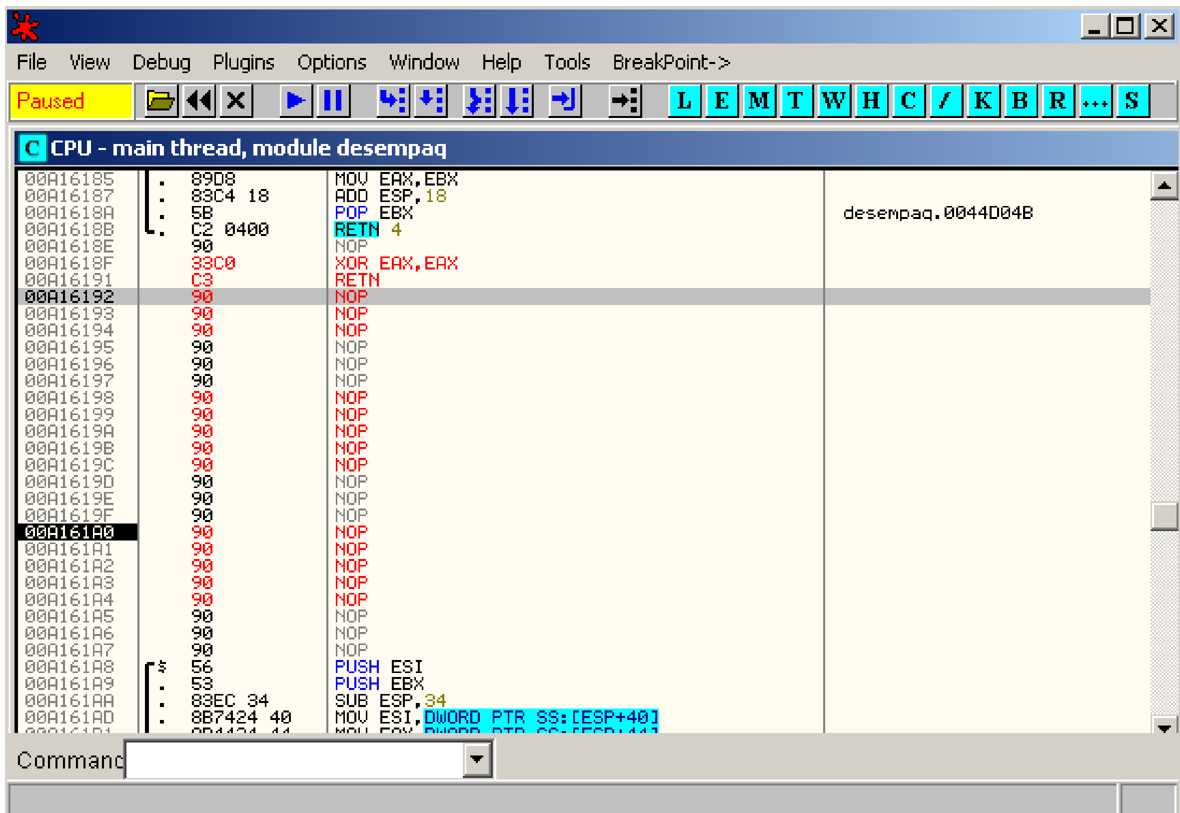
Seleccionamos por lo tanto la línea del CALL y ponemos un punto de ruptura. Después de seleccionar “Help” -> “About” Olly se detiene en 44D046:



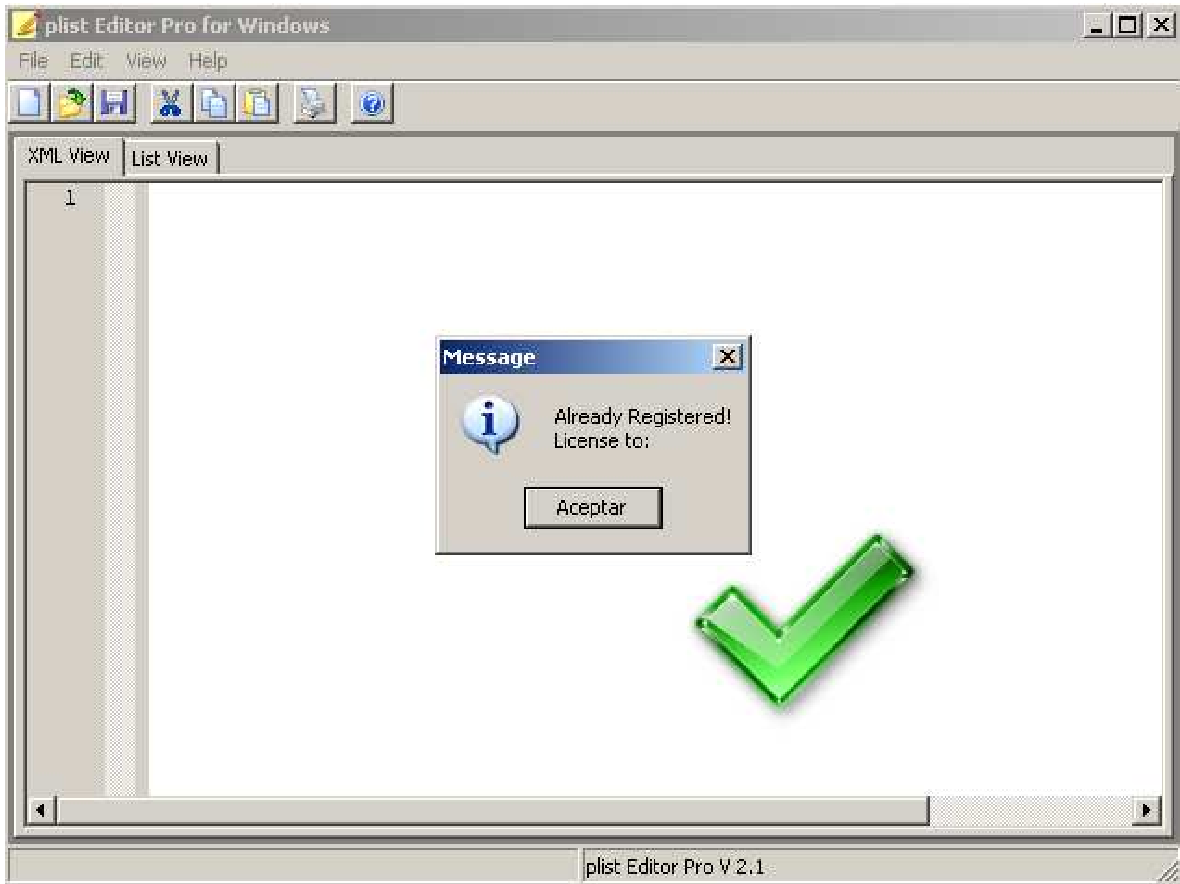
Antes de entrar en el CALL recordemos que queremos que AL (EAX) sea igual a cero cuando regrese del CALL ya que de esta forma tomaremos el salto en 44D04D.

Pulsamos F7:

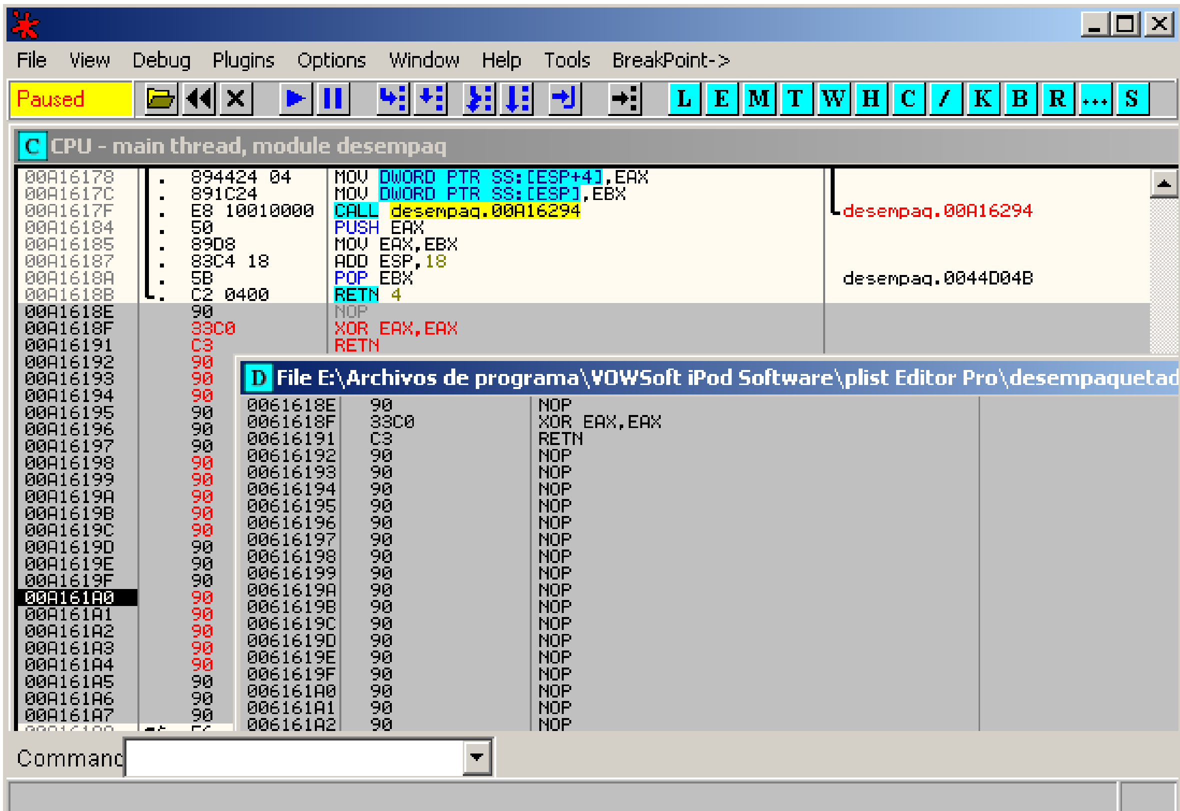
Vemos que esta rutina es llamada de siete lugares diferentes. Como tomamos como hipotesis de que esta rutina solo comprueba si estamos registrados o no, vamo a nopear las cuatro siguientes líneas y en su lugar escribiremos nuestro propio código. Como queremos que AL sea igual a cero escribiremos la siguiente instrucción XOR EAX, EAX, y después RETN.



Pulsamos F9, y seleccionamos “Help” -> “Register”:

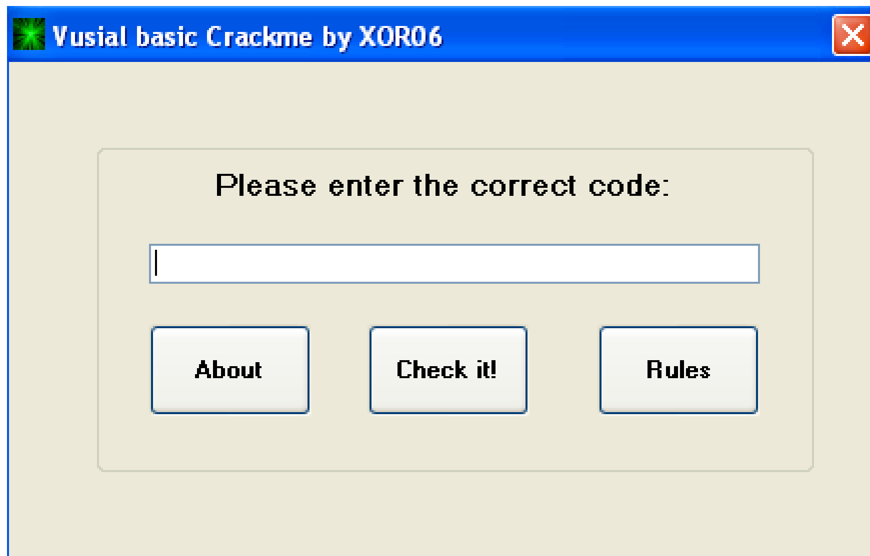


A la hora de guardar el nuevo ejecutable seleccionaremos solo “copy selection” ya que en este caso se trata de una aplicación con código auto modificable y lo único que queremos parchear es el código que nosotros hemos modificado y no aquel código que el programa modifica automaticamente:



8.6 Crackme .net

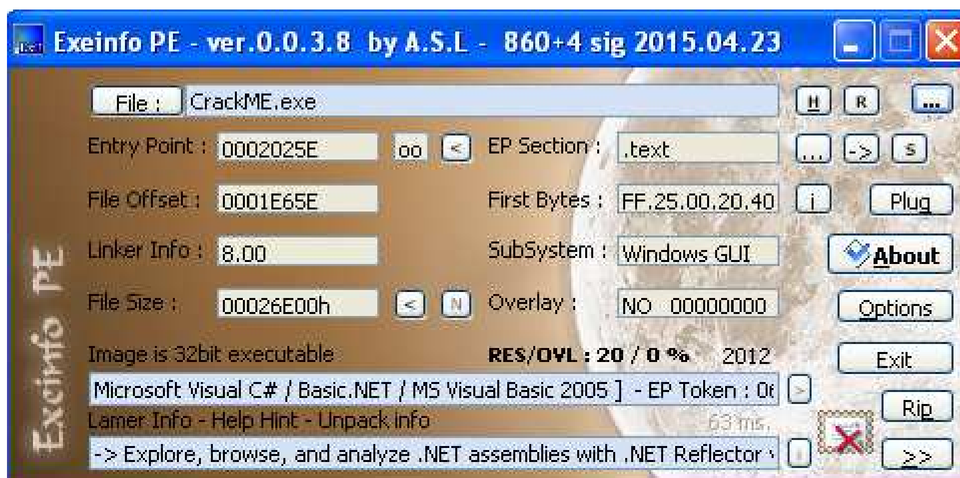
Ejecutamos la aplicación CrackMe para una primera toma de contacto:



Vemos que hay tres botones. Si hacemos clic en "Check it" nos sale el "bad boy":

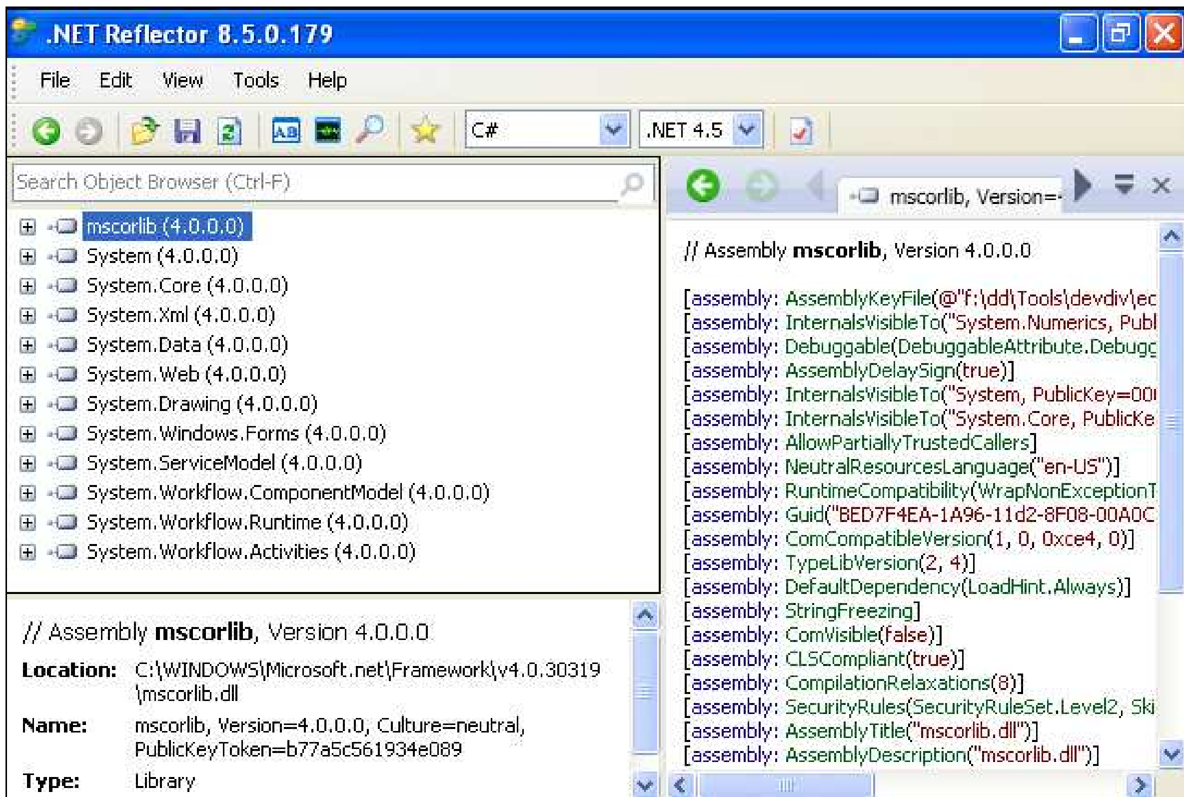
Si intentamso cargar el programa en Olly veremos rapidamente que esto no nos lleva a ningún sitio. De la misma forma que los programas escritos en Delphi se pueden cargar en DeDe para obtener información, los programas .NET tambíen cuentan con programas para ayudar a parchearlos.

Cargemos el CrackMe en ExeinfoPE para averiguar de que tipo de programa estamos hablando:

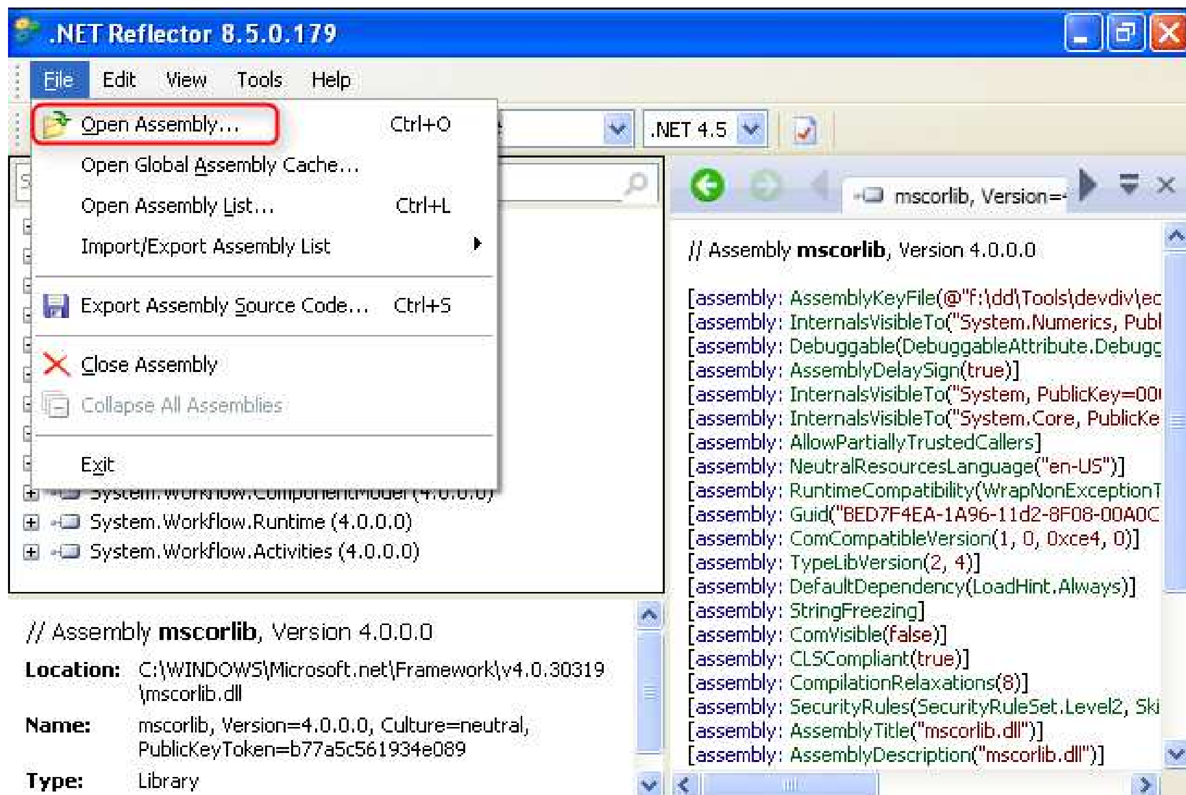


Vemos que la aplicación está escrita en Visual C# / Basic.NET. Cuando veamos un programa escrito en *.NET debemos intentar cargarlo siempre en “Redgate’s Reflector”.

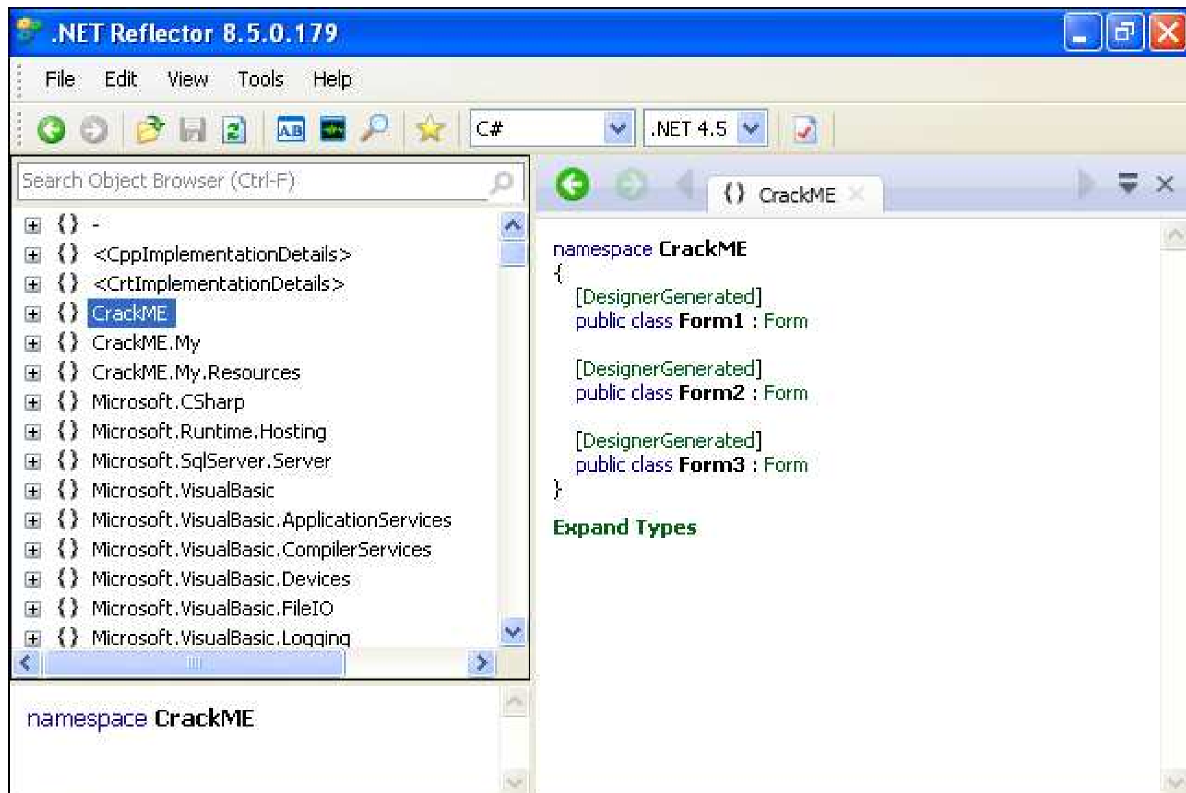
Si ejecutamos Reflector veremos la siguiente pantalla inicial:



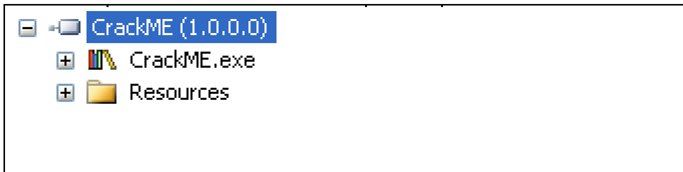
Cargamos nuestro CrackMe haciendo clic en “File” -> “Open Assembly…”



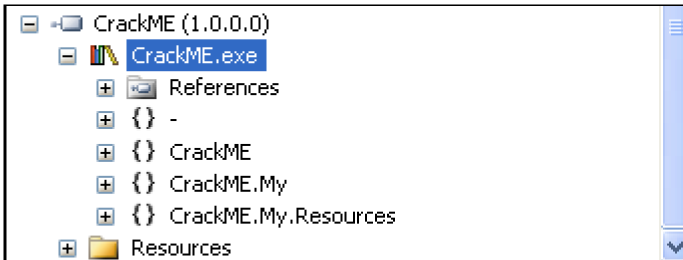
Seleccionamos nuestro CrackMe:



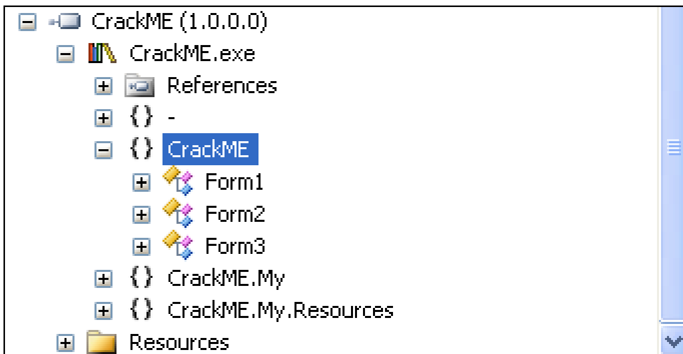
Y desplegamos la pestaña:



A continuación desplegamos la pestaña CrackMe.exe:

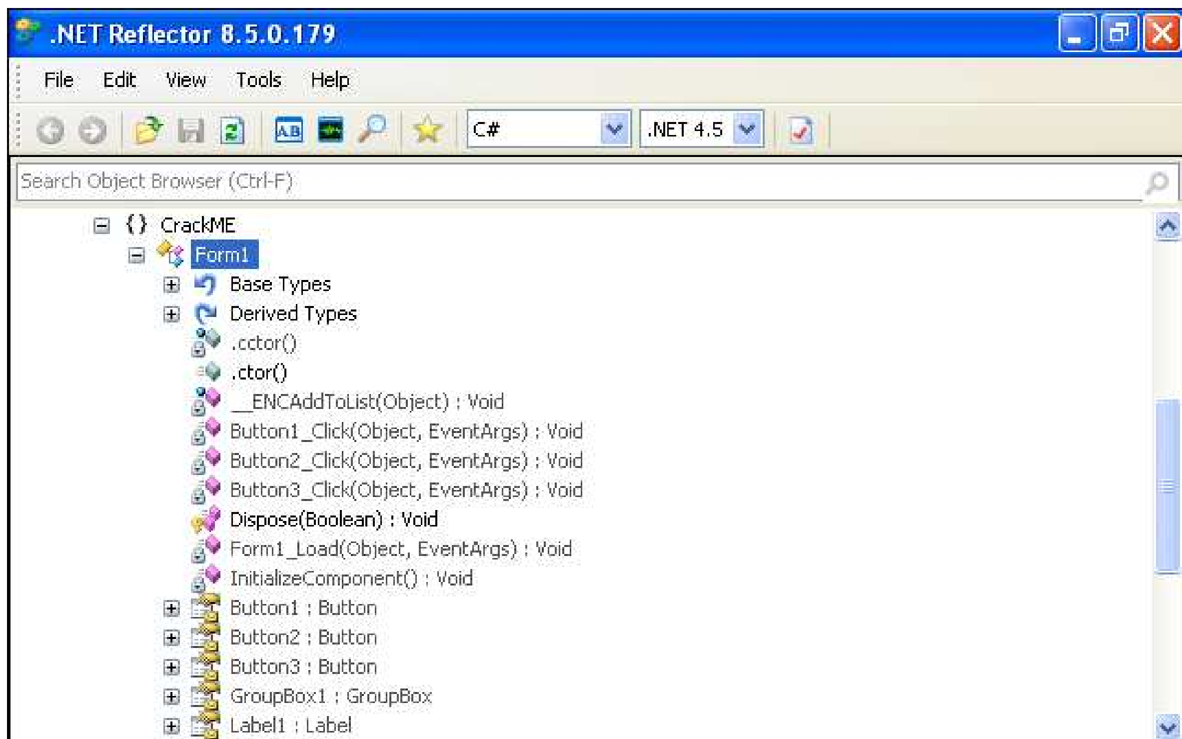


Y en esta sección encontraremos toda la información que necesitamos. Seleccionamos CrackMe (sin extensiones) y volvemos a desplegarlo:

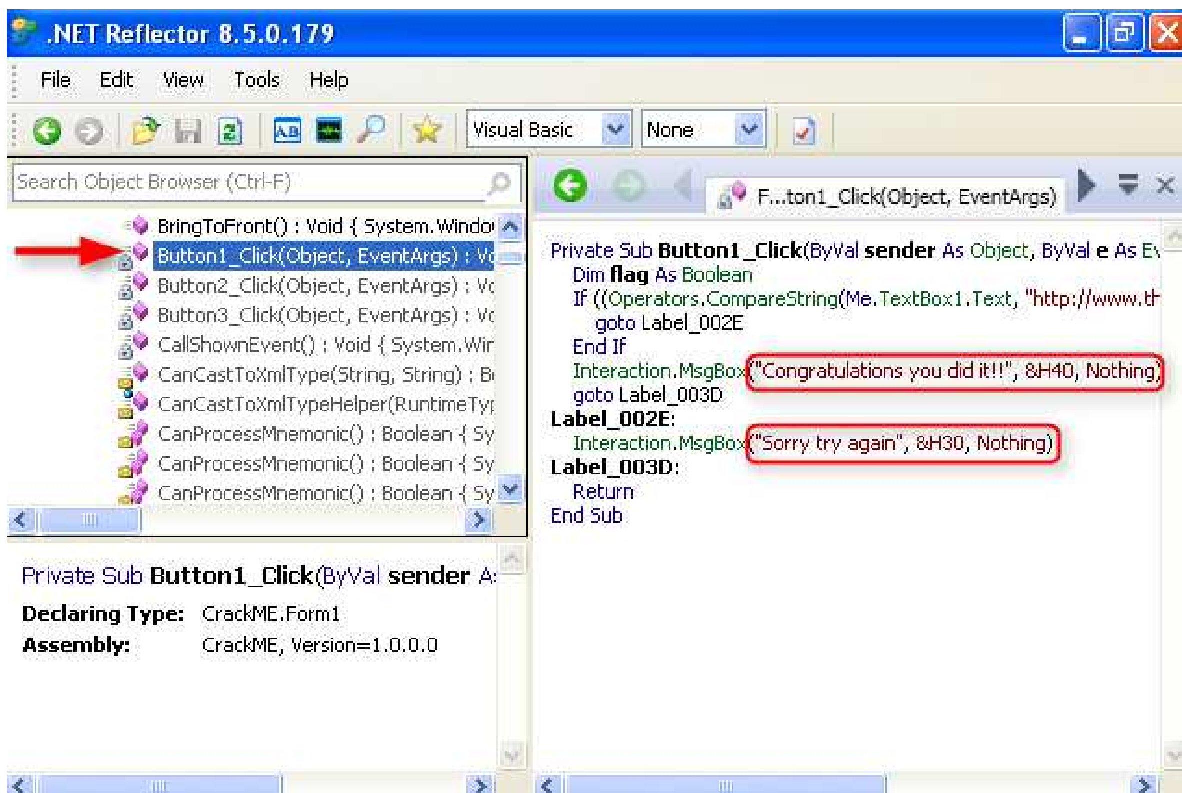


Podemos ver todos los formularios que utiliza el programa. En Visual Basic las ventanas se llaman Form, como en Delphi. Sabemos que la aplicación tiene tres botones. Busquemos pues el botón “Check it”, que es el que comprueba si el serial es correcto o no.

Desplegamos Form1, y vemos que tiene tres botones y un cuadro de texto:



Si desplegamos Form2 y Form3 veremos que solo hay un botón por lo que Form1 parece mostrarnos el camino a seguir. Si seleccionamos el evento Button1_Click, veremos en la parte derecha tanto el "bad boy" como el "good boy".



Echemos un vistazo a la primera parte del código:

```
Private Sub Button1_Click(ByVal sender As Object, ByVal e As EventArgs) Dim flag As Boolean If ((Operators.CompareString(Me.TextBox1.Text, "xxxxxxxxx", 0) Is 0) Is Nothing) Then goto Label_002E
```

El código compara el serial introducido en el CrackMe con el contenido de TextBox1, que en nuestro caso es “xxxxxxxxx”. Si ambos contenidos coinciden entonces veremos el “good boy” en caso contrario, o si no se introduce nada, entonces veremos al “bad boy”.

8.7 Multi-parche

En este ejercicio vamos a crackear una aplicación que en lugar de comprobar el serial, lo que hace es una comprobación del servidor; hay un periodo de prueba de 30 días con su correspondiente nag.

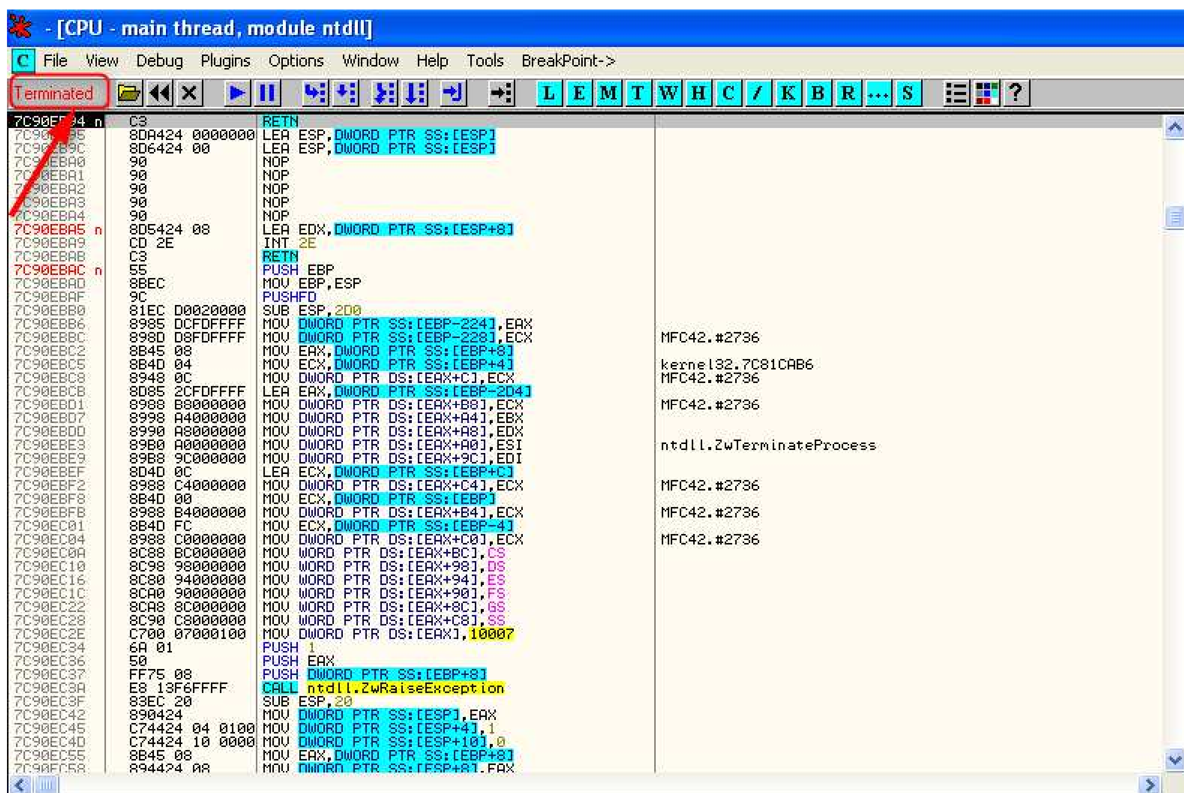
Descargamos Honestech Video Editor 8.0 Trial de internet, y hacemos doble clic sobre el ejecutable:

Podemos escoger entre dos versiones. Nosotros vamos a seleccionar el "Advanced Mode":

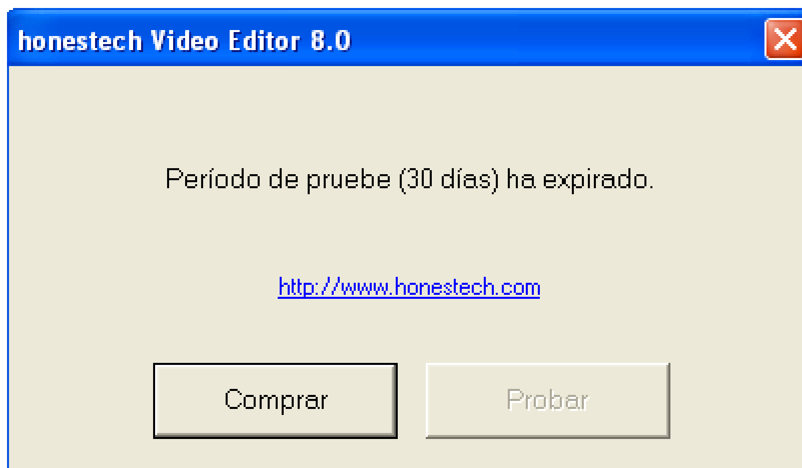
Vamos adelantar la fecha de nuestro equipo 30 días y reiniciamos la aplicación:

Ahora el periodo de prueba ha expirado y la opción de “Probar” ha desaparecido. Lo primero que vamos a parchear es el periodo de prueba de 30 días:

Cargamos la aplicación en Olly, pulsamos F9 y cuando sale la pantalla inicial seleccionamos “Advanced Mode”. Vemos aparecer la pantalla anterior por unos segundos, luego desaparece y Olly muestra el siguiente resultado:

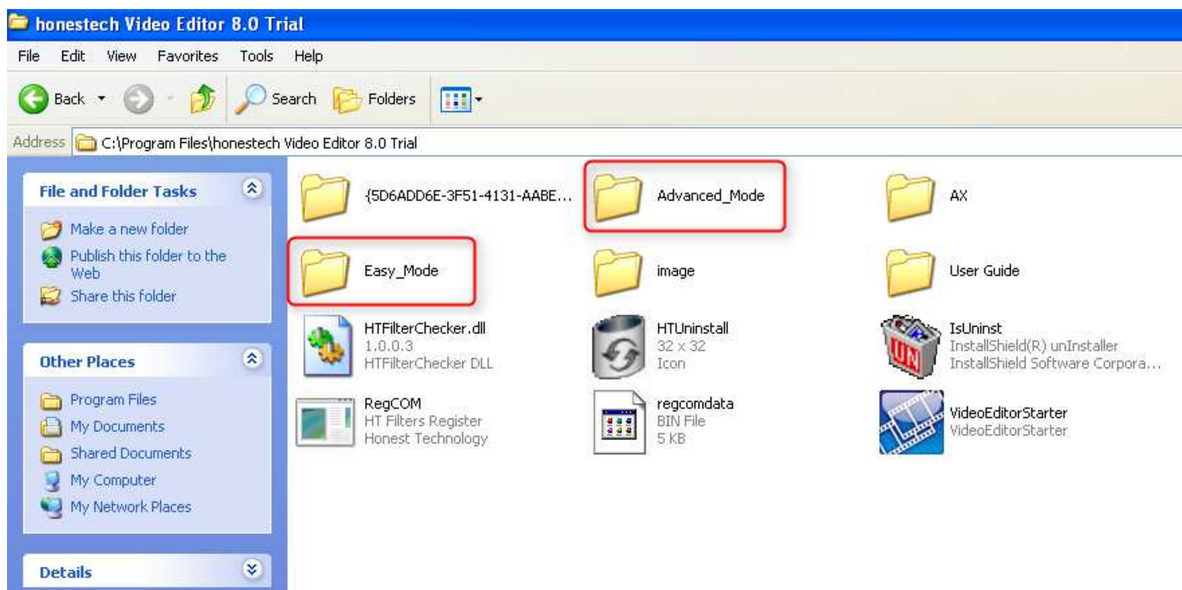


Analizemos esta situación: Cerramos Olly y vemos que la aplicación sigue ejecutandose!!

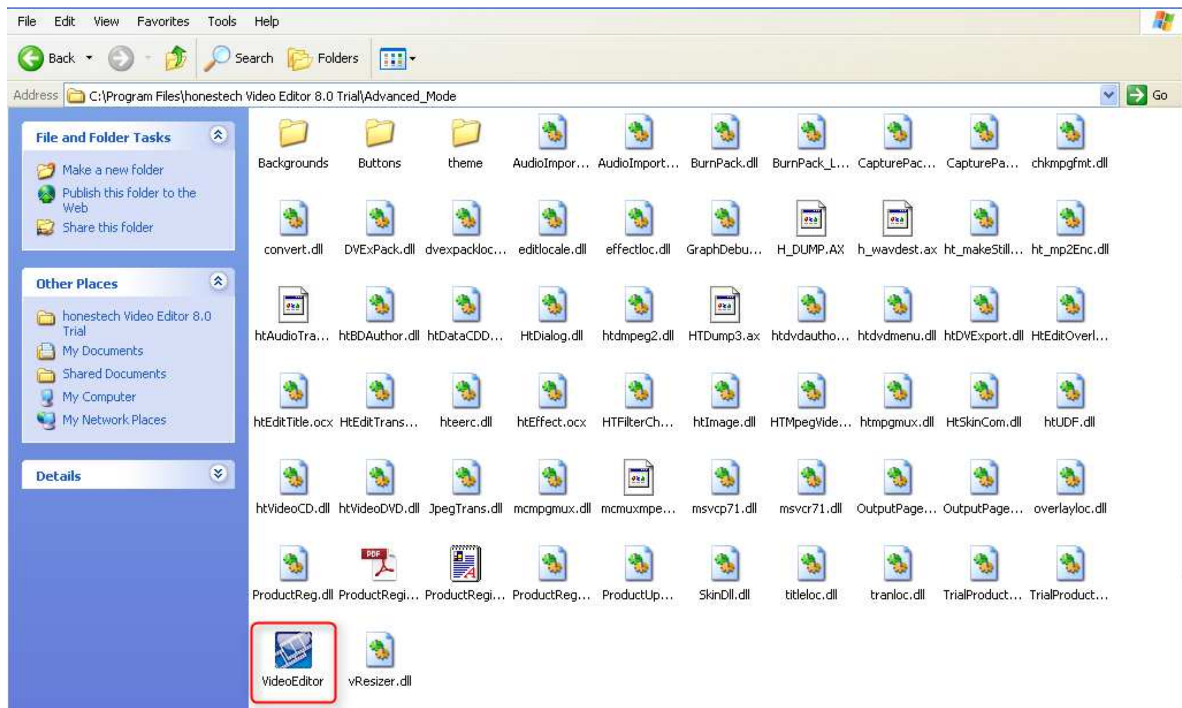


Esto nos indica que lo que hemos cargado en Olly no es la aplicación en sí. De hecho cuando hemos seleccionado la opción “Advanced” lo que hizimos fue ejecutar un cargador que al cabo de unos segundos finalizó.

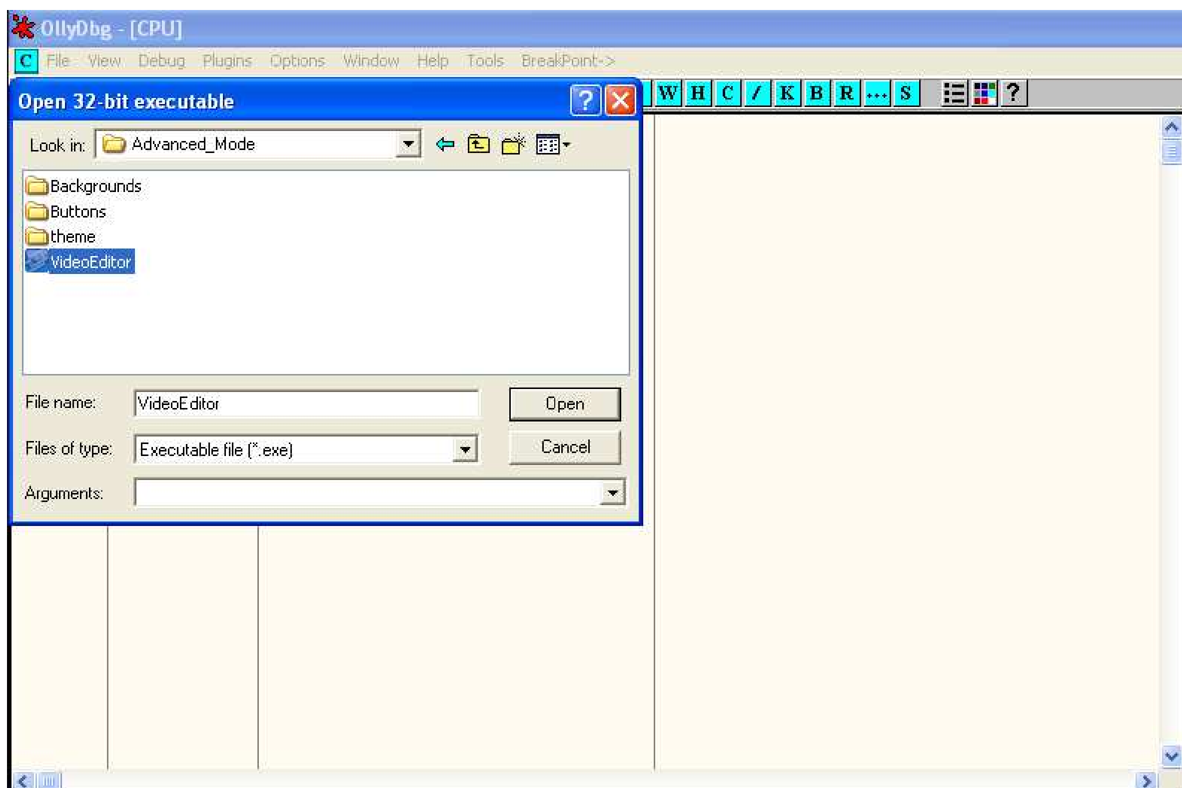
Echemos un vistazo al directorio del programa:



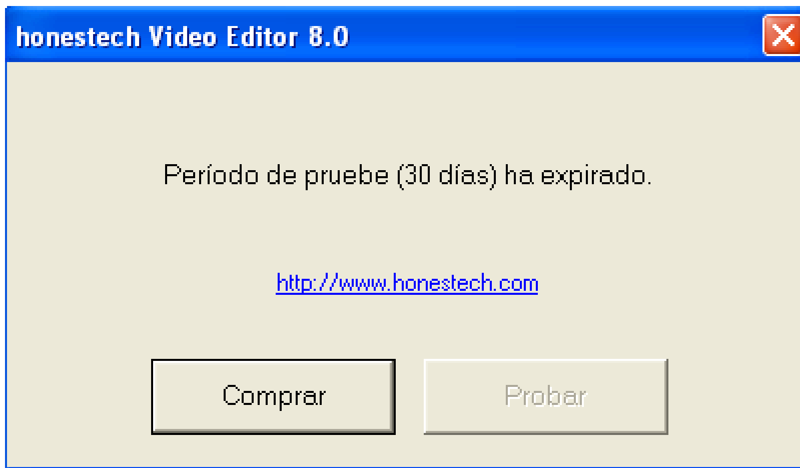
Como podemos ver hay una carpeta “Advanced_Mode” y otra “Easy_Mode”. Miremos lo que hay en la carpeta “Advanced_Mode”:



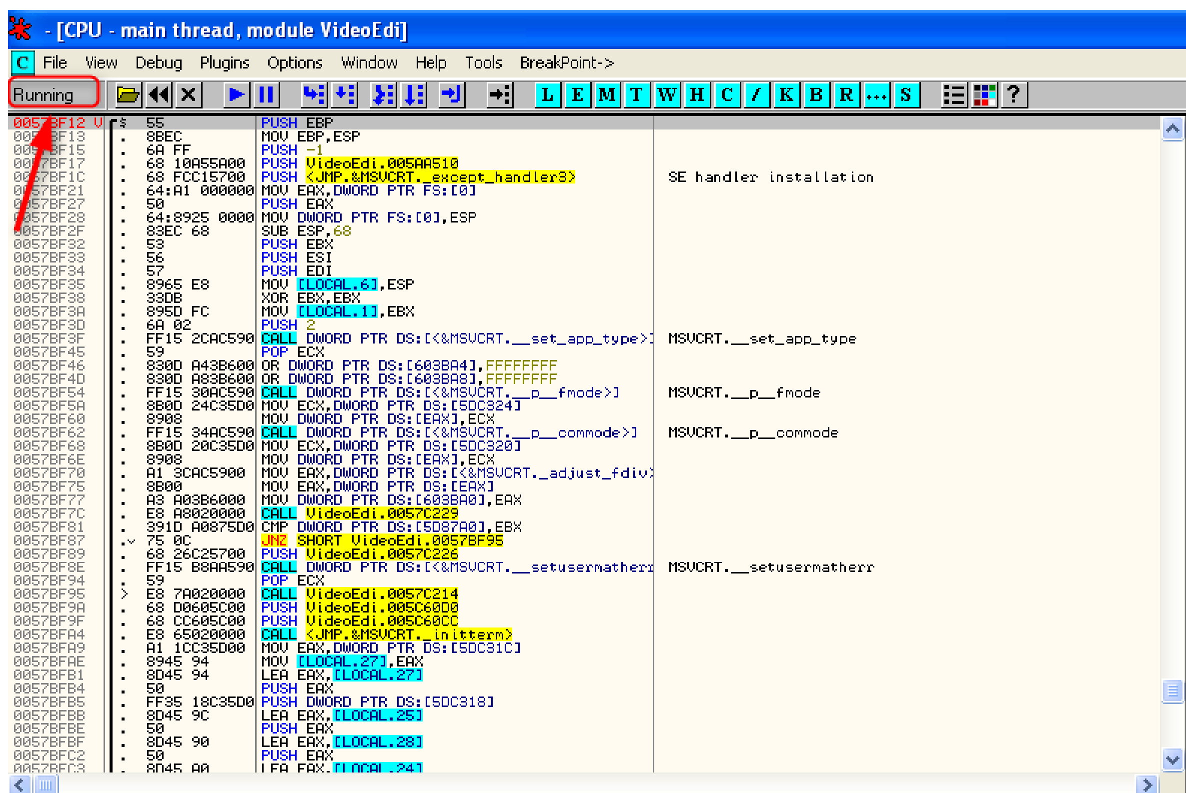
Vemos el ejecutable del "Advanced_Mode", lo que viene confirmar nuestras sospechas. Cargemos este ejecutable en Olly:



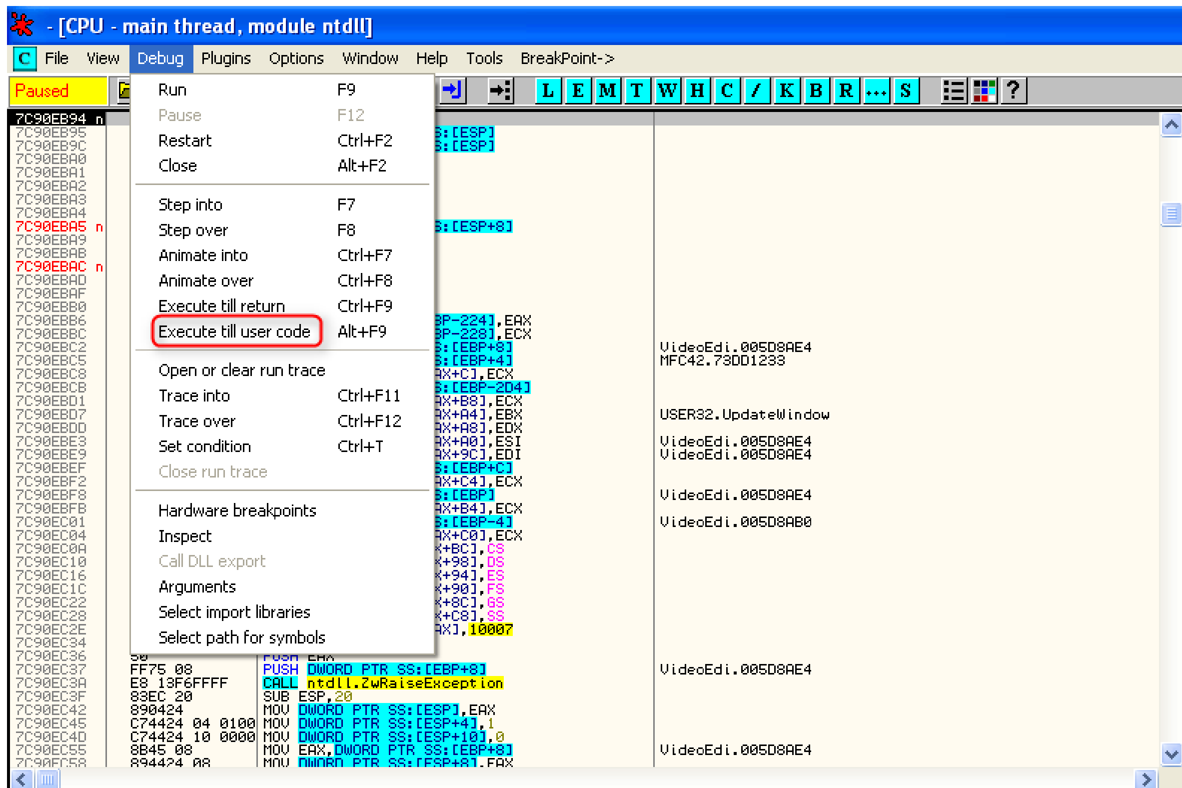
Pulsamos F9:



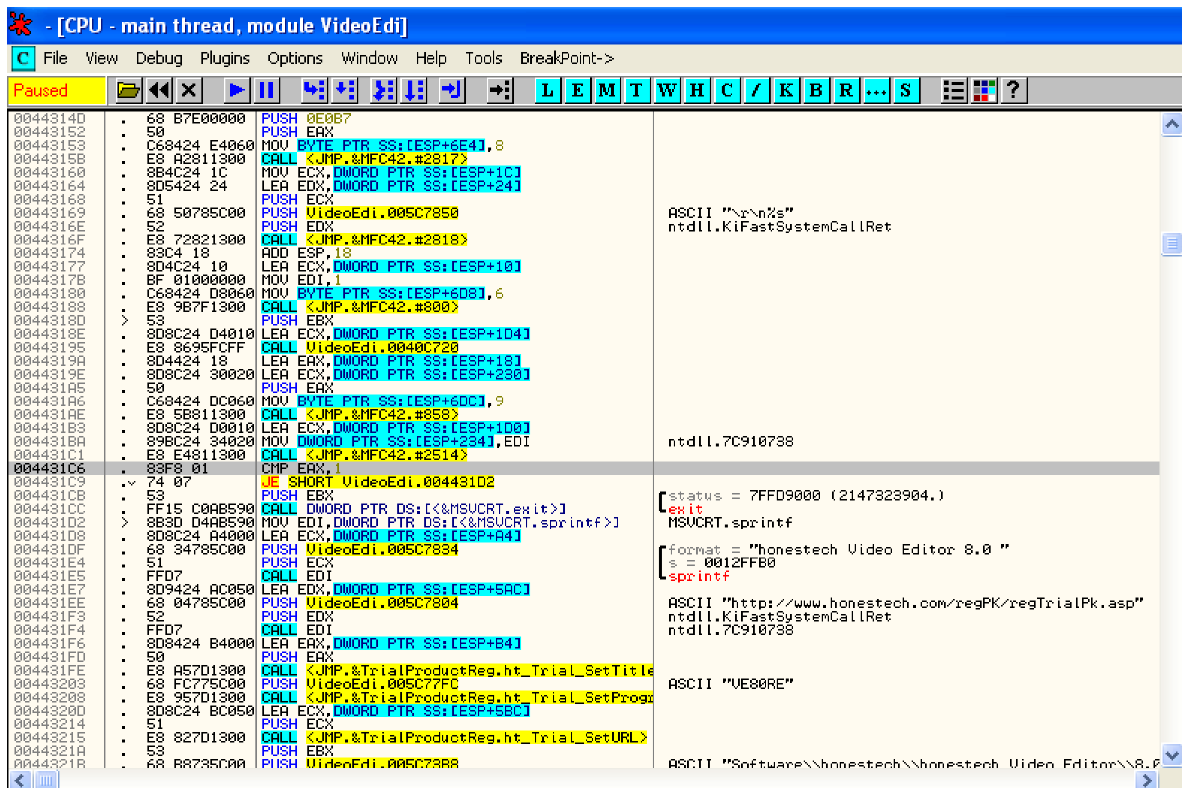
Y Olly sigue ejecutandose:



Procedemos a parchear el periodo de prueba. Detenemos Olly y seleccionamos "Execute till user code".

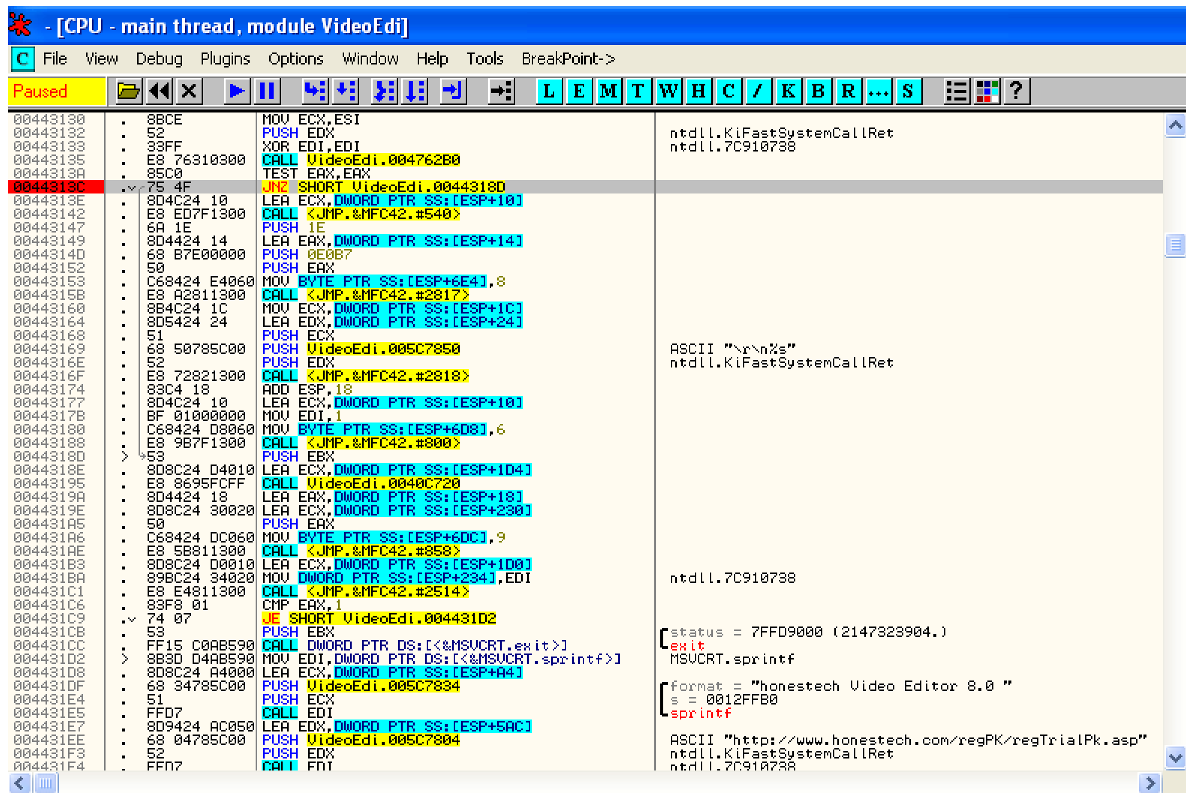


Hacemos clic en el único botón activo: “Comprar” y paramos en la siguiente línea de código:

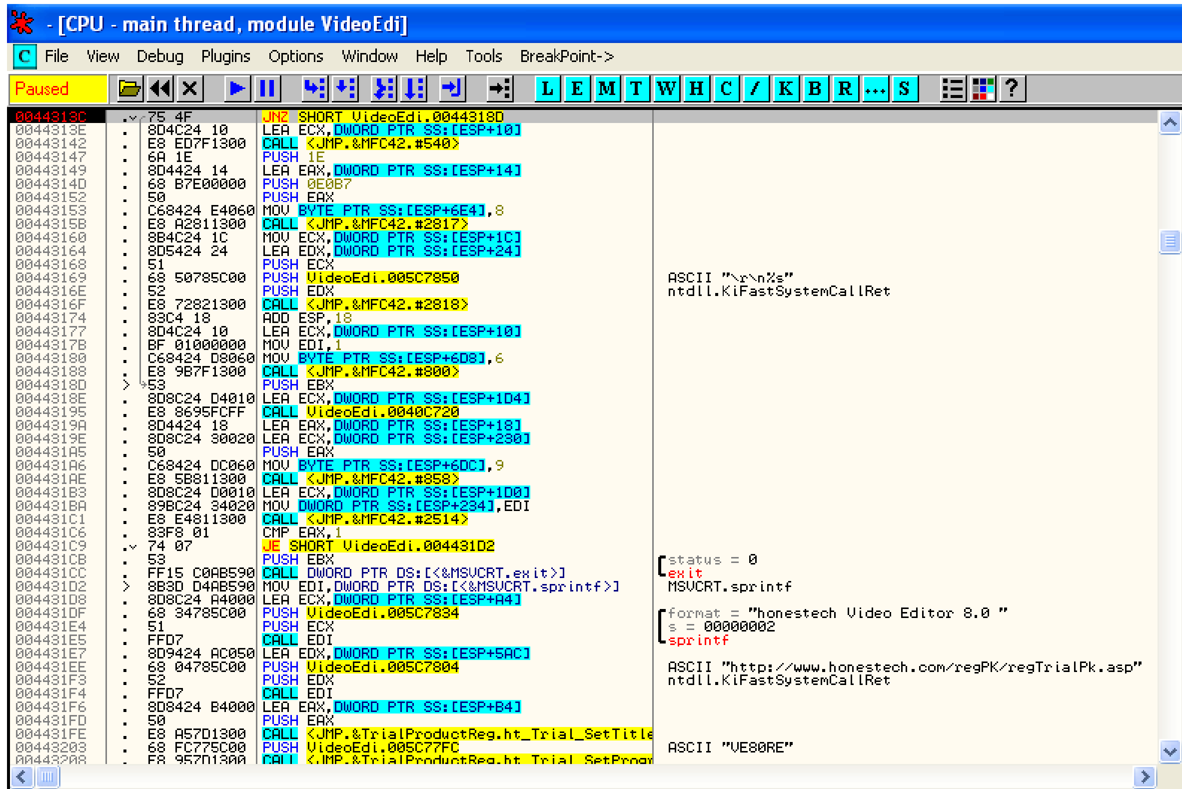


El CALL en la dirección 4431C1 es el que llama a nuestro nag. El CALL en la dirección 4431AE llama a MFC42. Descartamos este CALL porque nos interesa permanecer en

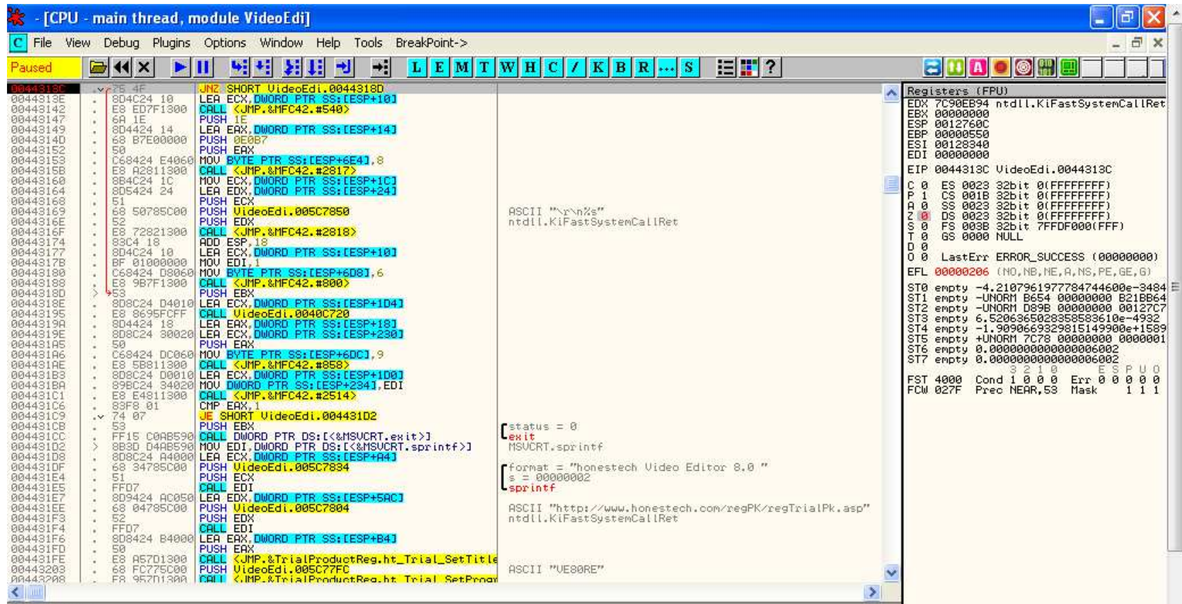
la rutina actual. Luego tenemos un CALL en la dirección 443195 hacia 40C720, pero si entramos aquí veremos que no hay ningún salto condicional y ningún CALL que profundize más en el código. Por esta razón paramos en la dirección 44313C donde nos encontramos con una instrucción JNZ. Ponemos un punto de ruptura y reiniciamos Olly:



Olly se detiene en el punto de ruptura, y como podemos ver no va tomar el salto:

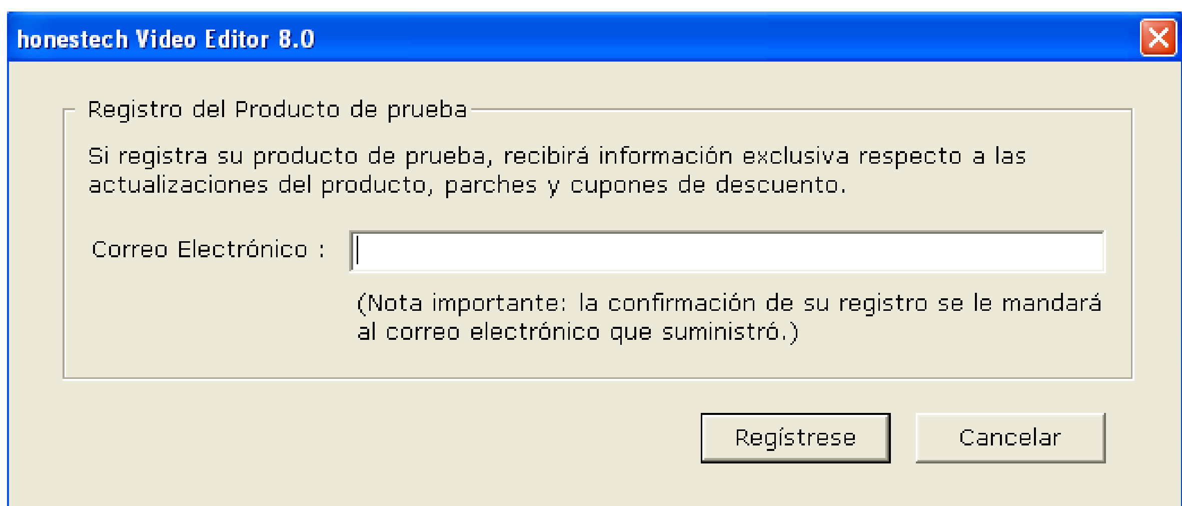


Cambiamos el valor de la bandera Z para que salte por encima del nag, y pulsamos F9:



Como podemos comprobar ahora sí es posible hacer clic sobre el botón “Probar”.

La respuesta que obtenemos es la siguiente:

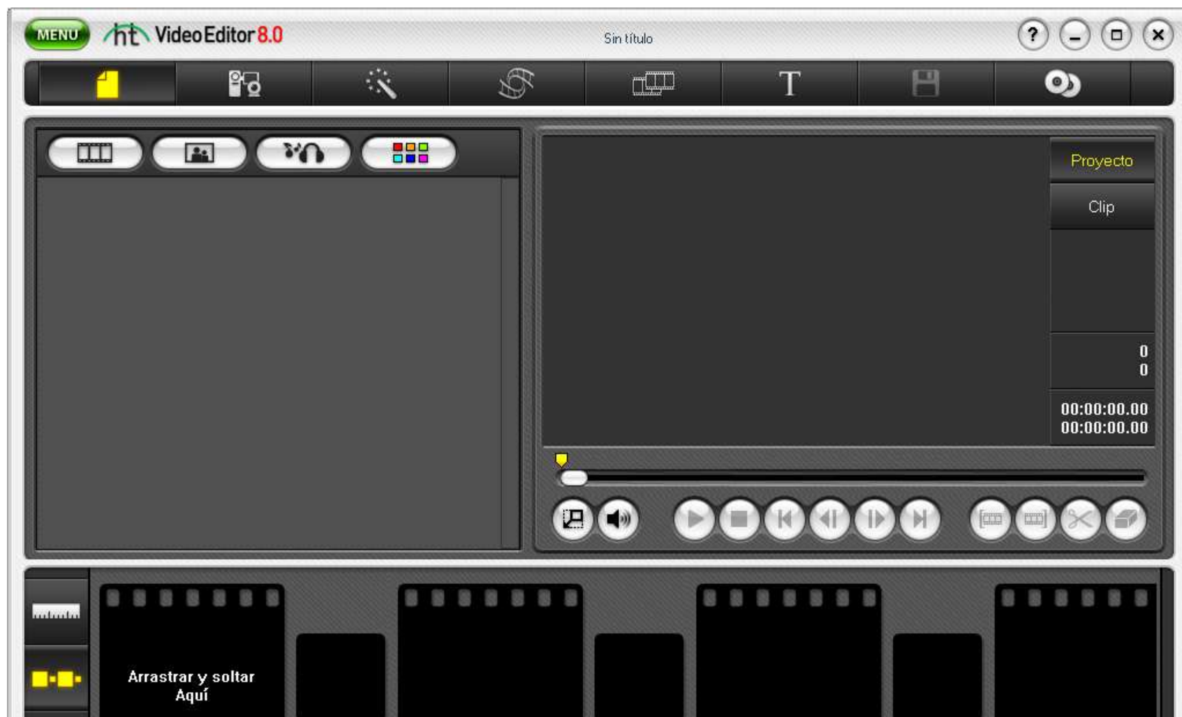


Como no queremos que nuestro correo electrónico quede registrado en una base de datos, lo primero que haremos es desconectarnos de internet. A continuación ejecutamos Fiddler, introducimos un correo electrónico erroneo y hacemos clic sobre “Regístrese”.

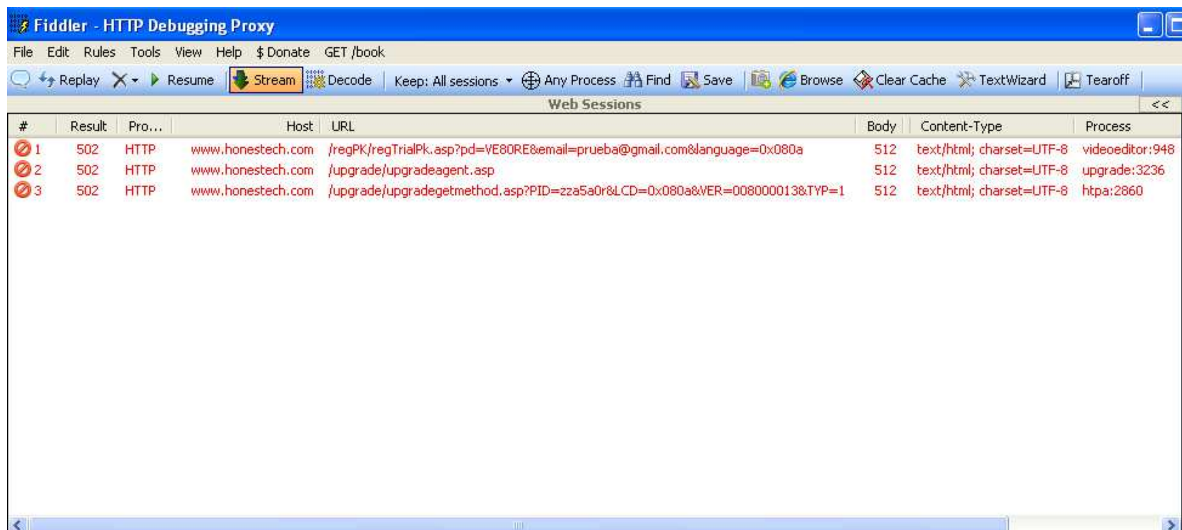
Nota: ***Fiddler*** *es un Servidor proxy para depurar código HTTP escrito por Eric Lawrence, ex administrador de programas en el equipo de desarrollo de Internet Explorer en Microsoft.1Fiddler captura tráfico HTTP y HTTPS y lo registra para que el usuario pueda revisar (esta última mediante la implementación de intercepción man-in-the-middle utilizando certificados autofirmados).5*

Fiddler también se puede utilizar para modificar (en inglés "fiddle with") el tráfico HTTP para solucionar problemas mientras se estén enviando o recibiendo.3 Por omisión, el tráfico de la pila WinINET HTTP (S) de Microsoft se dirige automáticamente al proxy en tiempo de ejecución, pero cualquier navegador o en aplicación web (y la mayoría de los dispositivos móviles) pueden configurarse para enrutar el tráfico a través de Fiddler.

https://es.wikipedia.org/wiki/Fiddler_%28software%29

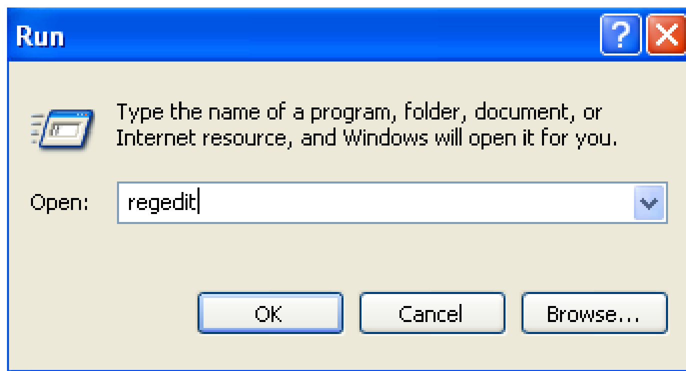


Vemos que el programa se ha cargado correctamente ya que no necesita de “feedback” por parte del servidor, solo envía nuestra información al servidor.

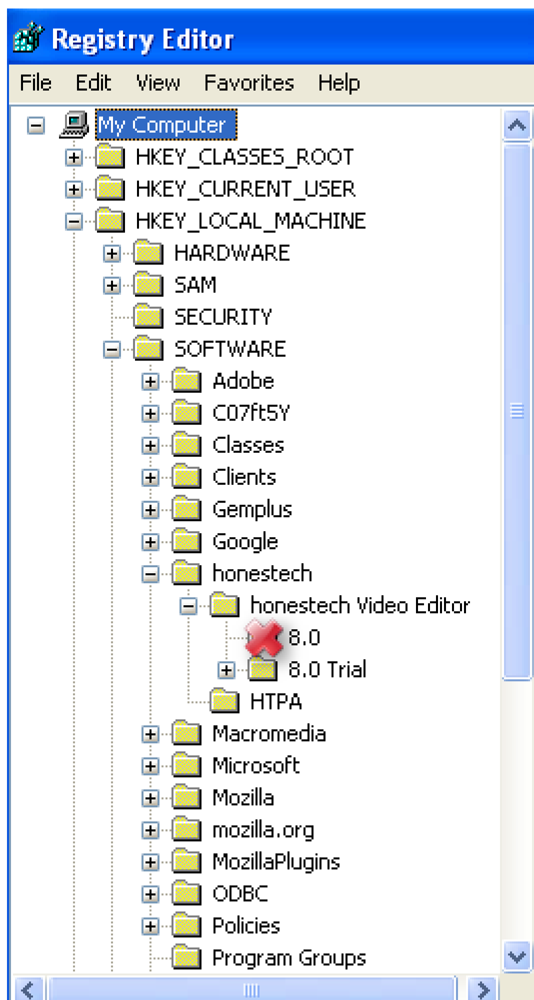


Si miramos en Fiddler podemos ver que la primera línea corresponde al envío de nuestro falso correo electronico al servidor. Las dos últimas lineas parecen ser actualizaciones de Fiddler.

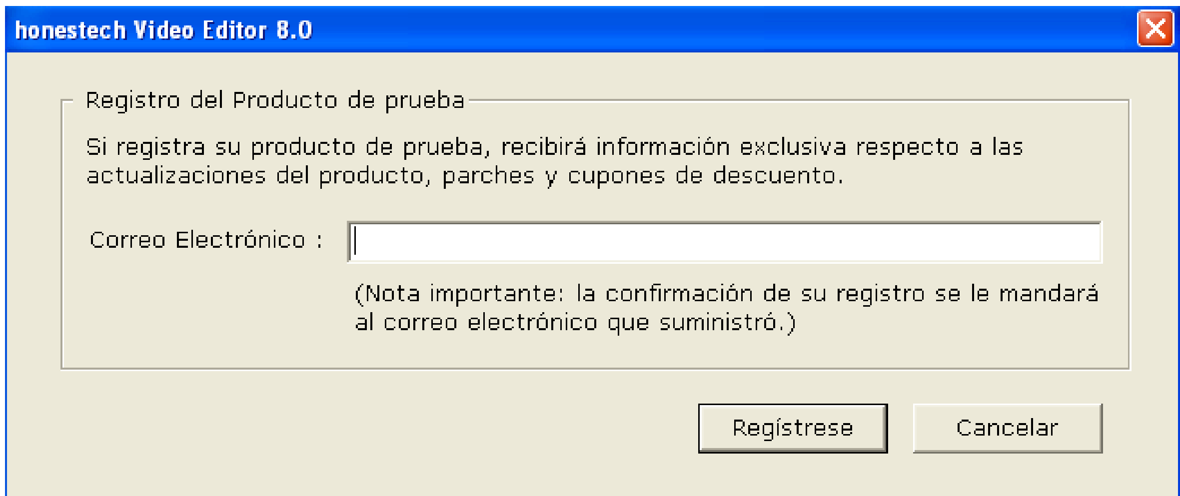
Vamos pues a eliminar la primer línea. Como no sabemos a priori si el programa necesita algún tipo de feedback por parte del servidor, necesitamos eliminar la clave de registro. A parte de esto tenemos que estar seguros que el programa no guarde nuestra información en algún sitio para enviarla una vez conectado el equipo a internet. Para borrar nuestro registro de correo electronico, ejecutamos regedit en nuestro equipo:



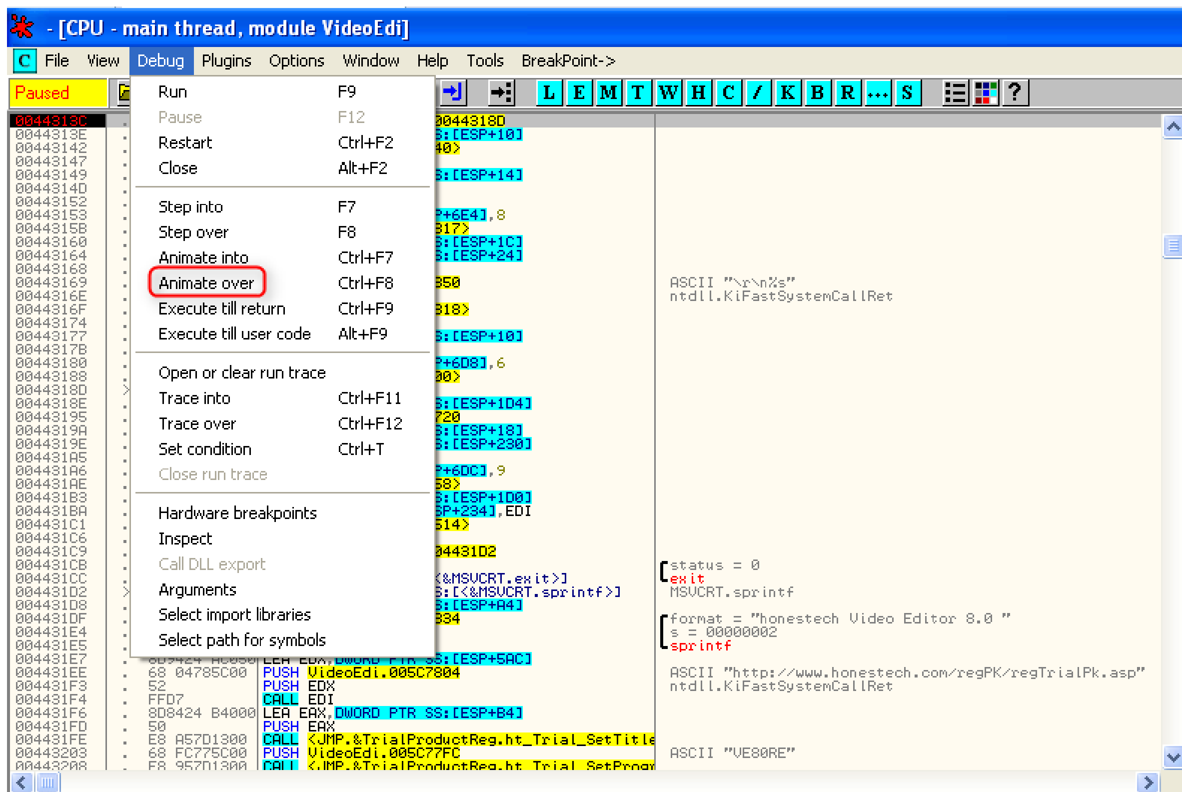
Y navegamos hacia "HKEY_LOCAL_MACHINE\SOFTWARE\honestech\honestech Video Editor", y borramos la carpeta "8.0":



Ahora la proxima vez que hagamos clic en "Probar" aparezerá de nuevo nuestro nag. Reiniciamos pues Olly, pulsamos F9, cambiamos el valor de la bandera Z, hacemos clic en "Probar" y volevemos a la pantalla de registrar:

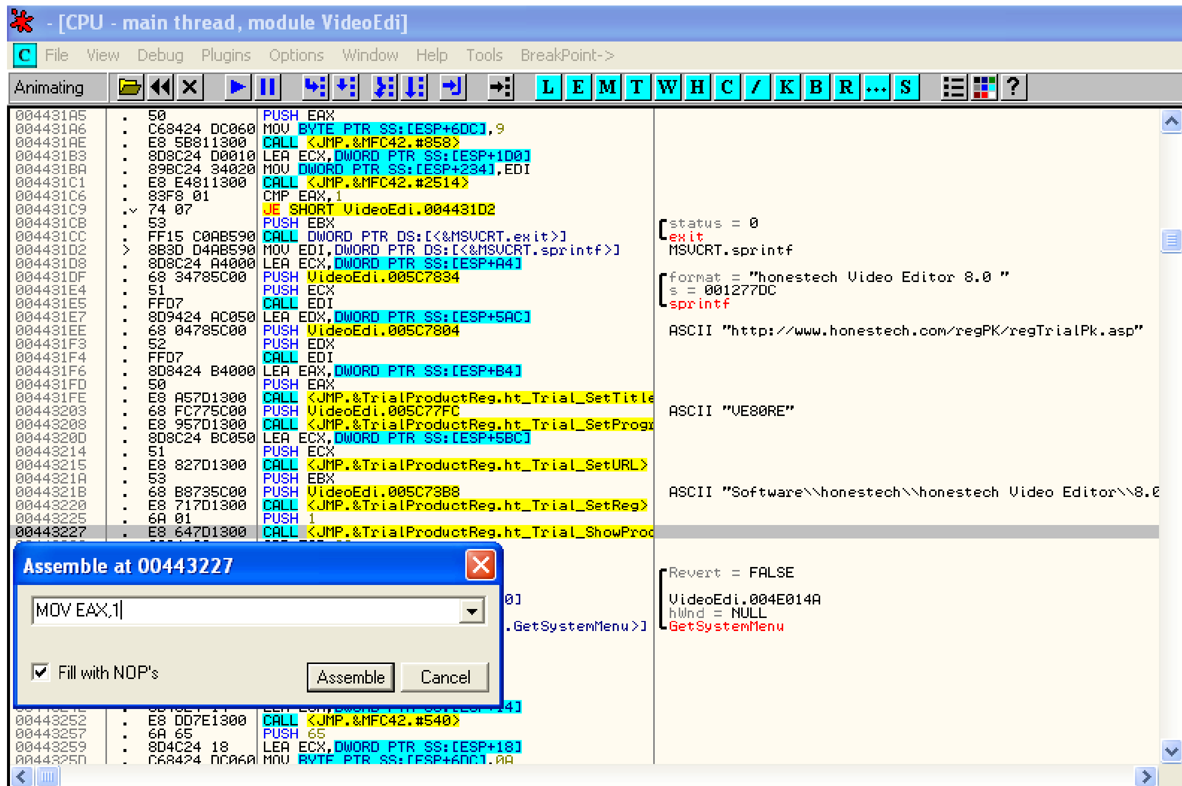


Utilizaremos la herramienta de "animate over" para parchear la aplicación.

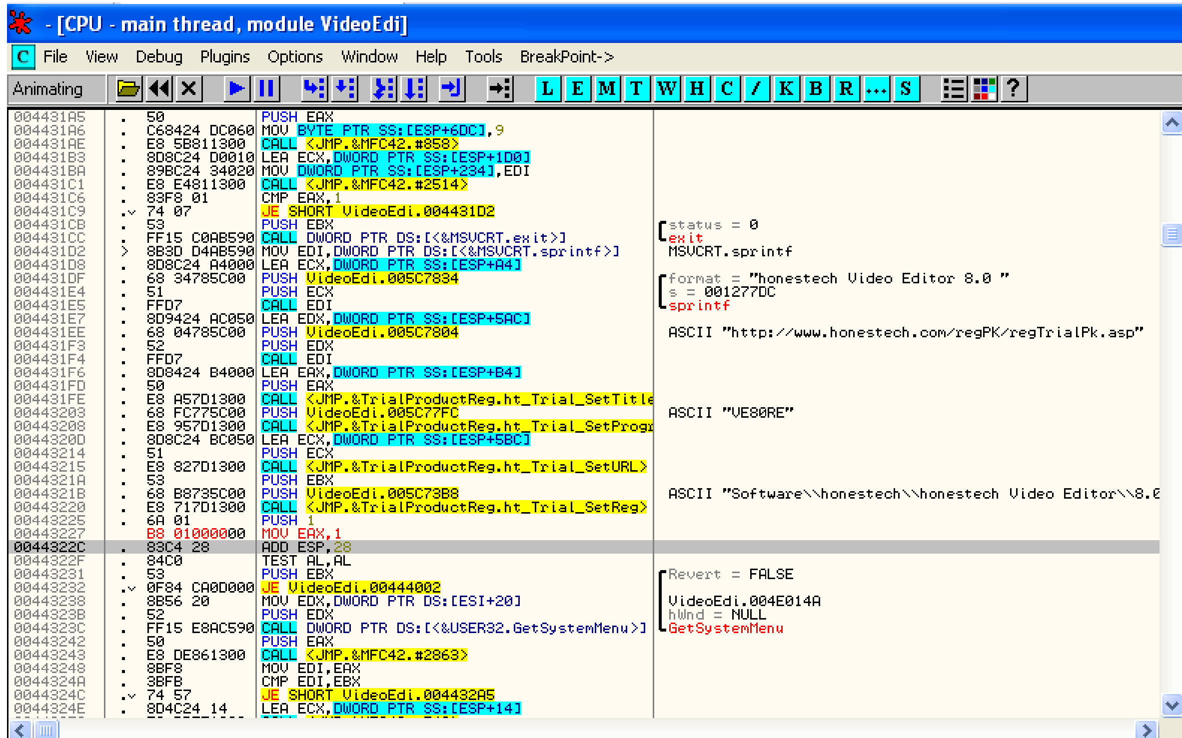


Después de pulsar sobre "Animate over", Olly se detiene en el CALL de la dirección 4431C1. Este CALL es el que nos muestra la ventana de registro y si nos fijamos en Fiddler, sin habernos conectado a internet.

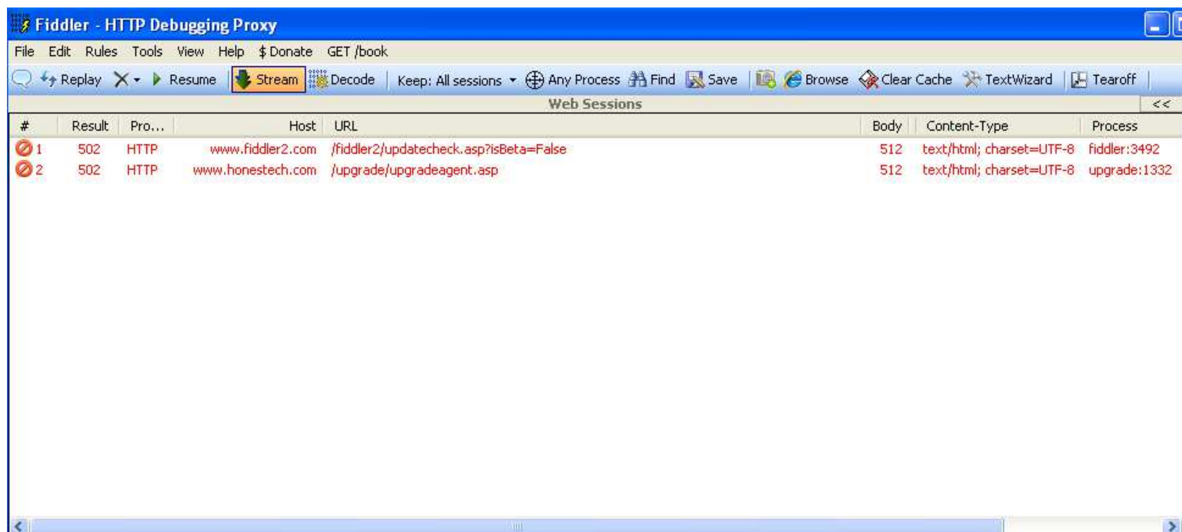
Si queremos cambiar la instrucción CALL por NOP veremos que la aplicación caerá ya que necesita obtener algún tipo de respuesta por parte del usuario en el cuadro de dialogo. Esa respuesta, como siempre, se almacena en EAX para ser comprobada a través de TEST EAX,EAX en la dirección 44322F. Si el usuario hace clic en "Registrar" EAX valdrá 1 y si el usuario hace clic en "Cancel" valdrá 0. Así que para deshacernos de este dialogo debemos nopear la instrucción CALL en la dirección 443227 y poner EAX igual a 1. Siempre que intentemos eliminar más de un botón en un cuadro de dialogo debemos especificar el valor de EAX.



El CALL era de 8 bytes y el MOV EAX,1 es de 8 bytes.

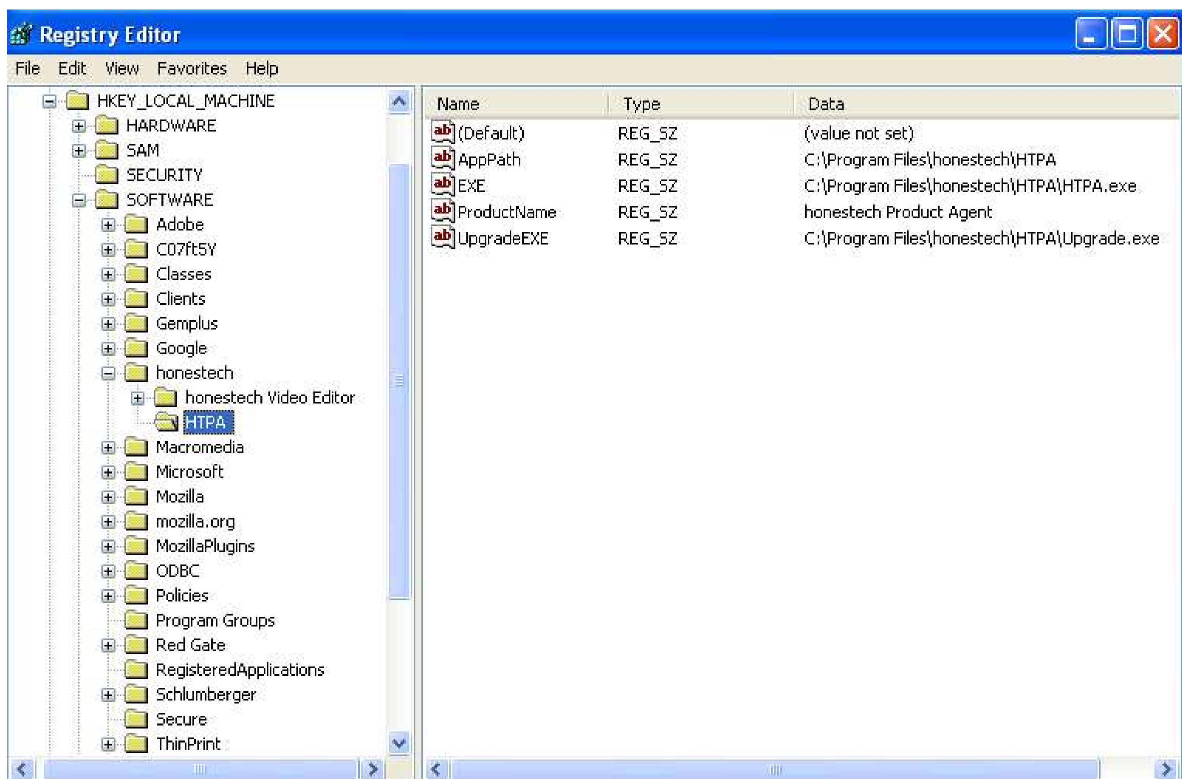


Reiniciamos Olly, aplicamos el parche y cambiamos la bandera en el punto de ruptura. Vemos que la aplicación se ejecuta normalmente y en Fiddler desapareció la primera línea.



Vemos que en la segunda línea aparece el nombre del dominio y al lado upgradeagent. Como no queremos dejar ninguna pista que pueda se reastreado volvemos a mirar en los registros a ver si averiguamos algo con respecto a esta actualización.

Para ello vamos a HKEY_LOCAL_MACHINE\SOFTWARE\honestech\HTPA



Aquí vemos la ruta hacia el upgradeagent (HTPA.exe) y el Upgrade.exe. Eliminamos ambas rutas ya que si no se pueden encontrar no se podrán iniciar.

Borramos todo lo que tenemos en Fiddler, reiniciamos Olly, aplicamos el parche, cambiamos la bandera y si nos fijamos en Fiddler ya no hay ningún intento para conectarse a internet.

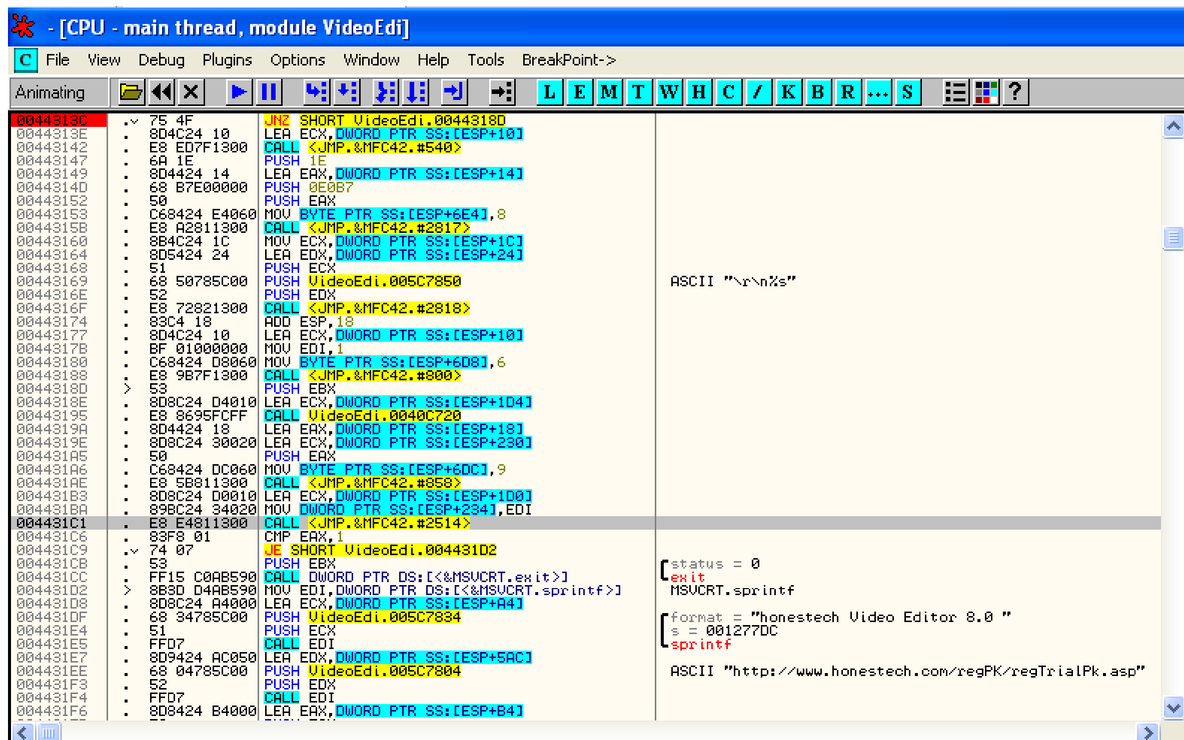
Resumimos lo que hizimos hasta ahora:

- Hemos parcheado el periodo de prueba de 30 días.
- Hemos parcheado cualquier intento de conectarse al servidor.
- Eliminamos el nag que envíaba nuestro correo electrónico.

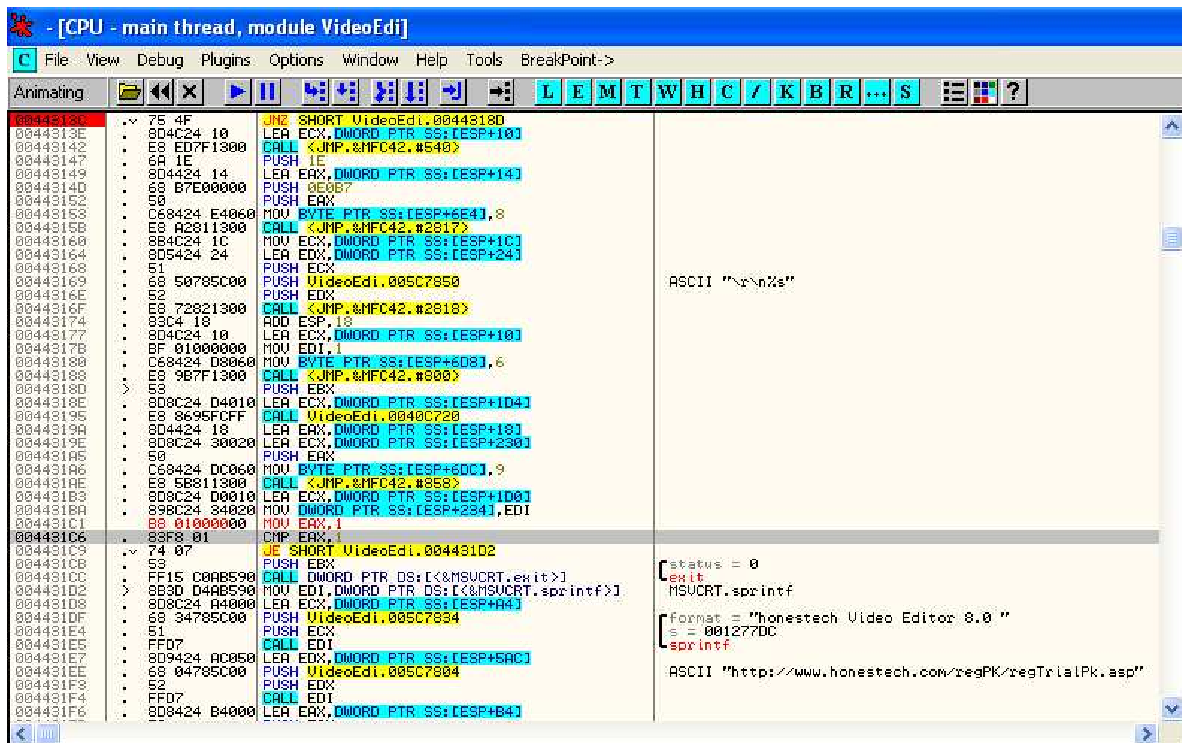
Cosas por hacer:

- Si vamos a la ventana de “Acerca de” sigue pondiendo “versión de prueba”.
- Prescindir del hecho de tener que hacer clic en “Probar” para acceder al programa.

Empezaremos por solucionar el punto 2. Reiniciamos Olly y ejecutamos la aplicación hasta el punto de ruptura, pulsamos “animate over” hasta que veamos el nag y Olly se detiene en 4431C1:

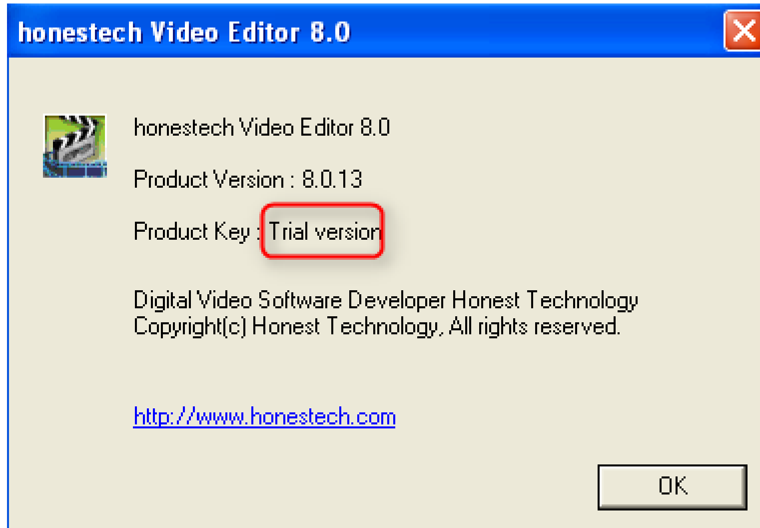


Haremos lo mismo que antes; sustituimos la instrucción CALL por MOV EAX,1.



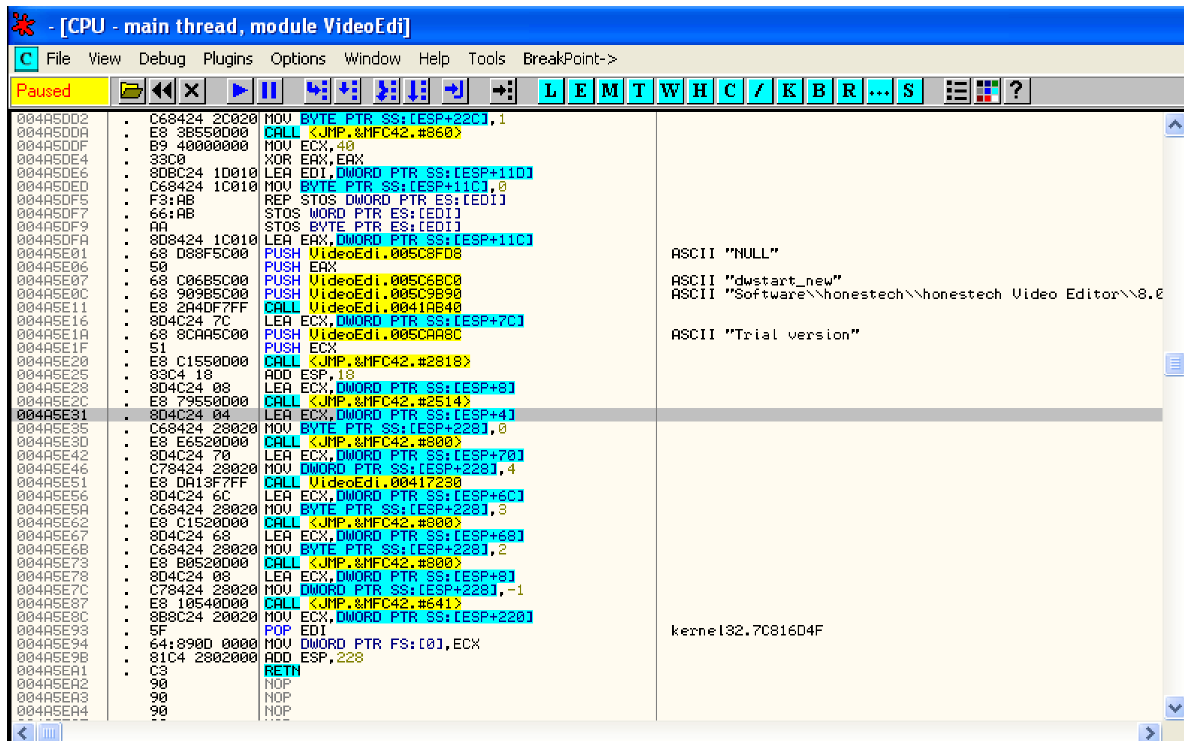
Este parche lo que hará es indicar al programa que el usuario hizo clic en “Probar”.

Con respecto al primer punto; reiniciamos Olly, aplicamos los dos parches y hacemos clic en “Acerca de… ”

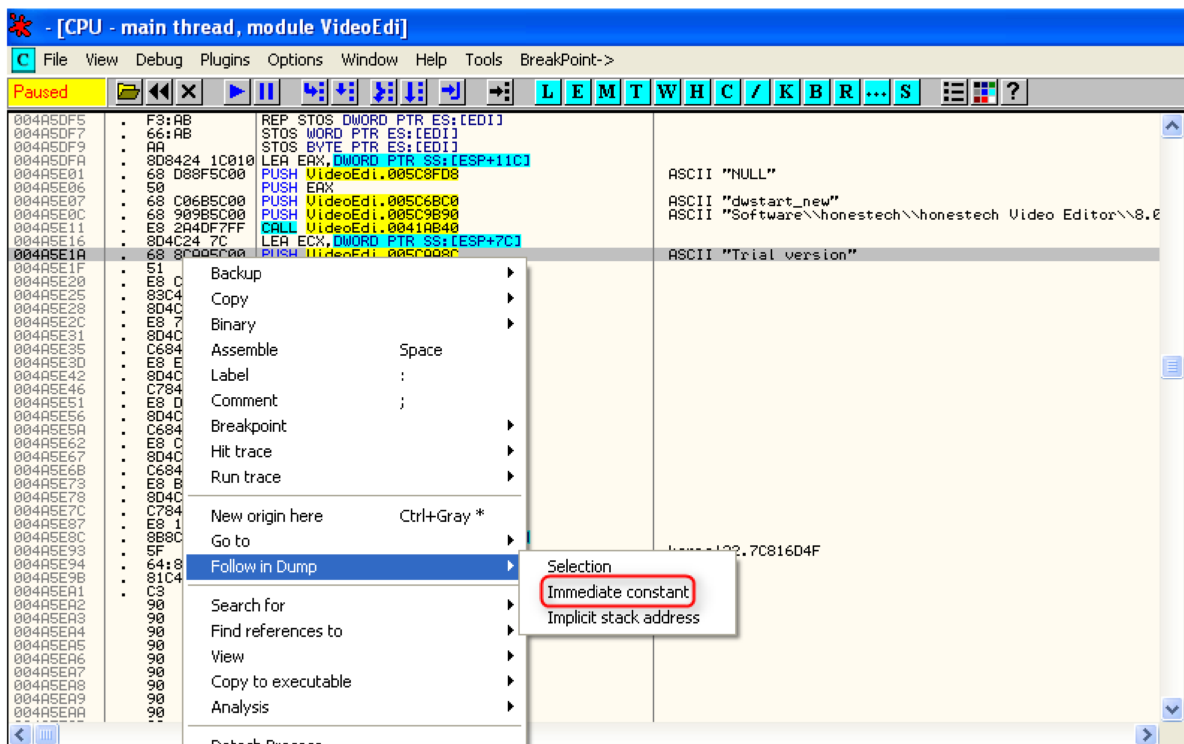


Realmente, el hecho de que ponga aquí “Trial version” no va afectar al funcionamiento del programa. No obstante veremos como cambiarlo.

Una vez cargado el programa, pausamos Olly y pulsamos sobre “Execute till user code”



En la dirección 4A5E1A podemos ver la cadena de texto que estamos buscando “Trial version”. Hacemos clic con el botón derecho y seleccionamos “Follow in dump” -> “Immediate constant”:



Editamos el binario y escribimos un serial cualquiera:

Hacemos clic en OK y ejecutamos el programa:

ANEXO I

SISTEMAS DE NUMERACIÓN

Un sistema de numeración es el conjunto de símbolos y reglas que se utilizan para la representación de datos numéricos y cantidades. Se caracteriza por su base que es el número de símbolos distintos que utiliza, y además es el coeficiente que determina cual es el valor de cada símbolo dependiendo de la posición que ocupe.

Los sistemas de numeración actuales son sistemas posicionales en los que el valor relativo que representa cada símbolo o cifra de una determinada cantidad depende de su valor absoluto y de la posición relativa que ocupa dicha cifra con respecto a la coma decimal.

I.I. Decimal

Se le llama decimal o en base diez porque cada posición de las cifras que forman un número, representa el múltiplo de esa cifra por una potencia de diez dependiente de su posición en dicho número:

10^4	10^3	10^2	10^1	10^0		
|	|	|	|	|		
4	5	1	8	3 =	4 x 10000	= 40000
					5 x 1000	= 5000
					1 x 100	= 100
					8 x 10	= 80
					3 x 1	= 3
						45183

I.II. Binario

Desde el diseño de los primeros ordenadores se emplearon circuitos biestables con la posibilidad de reconocer sólo dos estados: con tensión y sin tensión, es decir: 1 y 0. Para representar cualquier número binario sólo son necesarios dos caracteres: el "1" y el "0". El valor de un número binario se obtiene igual que el de uno decimal, sólo que se emplea la base dos en vez de diez:

2^5	2^4	2^3	2^2	2^1	2^0	
|	|	|	|	|	|	
1	1	1	0	0	1 =	1 x 32 (2^5) = 32
						1 x 16 (2^4) = 16
						1 x 8 (2^3) = 8
						0 x 4 (2^2) = 0
						0 x 2 (2^1) = 0
						1 x 1 (2^0) = 1
						57

I.III. Hexadecimal

La representación de un número binario requiere una gran cantidad de cifras. Para escribir 65536 (2^{16}) se necesitan 17 cifras: un uno y dieciséis ceros detrás. Por esta razón se buscó un sistema que fuera más o menos manejable por seres humanos y que fuera también fácilmente convertible a binario. La solución está en el sistema hexadecimal. El primer problema con que nos encontramos es el de que nos faltan cifras. En el sistema decimal tenemos las cifras del 0 al 9, y en binario, el 0 y el 1, pero en un sistema hexadecimal debemos utilizar otros caracteres para las cifras del "10" al "15" ya que en hexadecimal el 10 equivale al 16 decimal. Estos caracteres son las letras mayúsculas de la A a la F. Veamos ahora el valor en decimal de un número hexadecimal:

16^3	16^2	16^1	16^0	
|	|	|	|	
7	A	2	E =	7 x 4096(16^3) = 28672
				A(10)x 256(16^2) = 2560
				2 x 16(16^1) = 32
				E(14)x 1(16^0) = 14
				31278

La conversión entre números en binario y hexadecimal es relativamente sencilla. En efecto, podemos convertir este número hexadecimal en binario sin necesidad de calculadora y ni siquiera de lápiz y papel:

```
            8421 8421 8421 8421

            |||| |||| |||| ||||

     7A2E = 0111 1010 0010 1110

         (4+2+1)(8+2)(2) (8+4+2)
```

o a la inversa:

```
            8421 8421 8421 8421

            |||| |||| |||| ||||

            1111 0011 1001 1100 = F39C

      (8+4+2+1)(2+1)(8+1)(8+4)
```

Cada grupo de cuatro cifras en binario equivale a una cifra hexadecimal. Así: 0111=7 (4+2+1), 1010=A (8+2) etc. Esto es debido a que una cifra hexadecimal puede representar un valor decimal comprendido entre 0 (0) y 15 (F) exactamente igual que un grupo de cuatro cifras en binario: 0 (0000) y 15 (1111).

I.IV. Octal

Este es un sistema que prácticamente no se utiliza pero que no está de más saber que existe. La idea es la misma que la del sistema hexadecimal, pero en grupos de 3 bits: valor entre 000 (0) y 111 (7). Lógicamente, el 8 decimal equivale al 10 octal.

I.V. Conversión

Las conversiones más destacadas son entre números en decimal y hexadecimal. Como el ordenador trabaja sólo con números hexadecimales (esto no es del todo cierto, ya que el ordenador trabaja con números binarios, pero los desensambladores los pasan a hexadecimales), la conversión se hace de la siguiente manera:

```
7A2E:A = C37  resto =   8
 C37:A = 138  resto =   7
  38:A =  1F  resto =   2
  1F:A =   3  resto =   1
   3:A =   0  resto =   3
                      _____
                      31278
```

Si se divide un número por diez, el resto de esta división será siempre menor que diez, o sea una cifra que podrá representarse en formato decimal. La conversión de decimal a hexadecimal queda como sigue:

31278:16 = 1954	resto= 14	(E)
1954:16 = 122	resto= 2	2
122:16 = 7	resto= 10	(A)
7:16 = 0	resto= 7	7
		7A2E

Sin embargo, la máquina lo hace de otra manera:

31278			
0+3	=	3 x A =	1E
1E+ 1	=	1F x A =	136
136+ 2	=	138 x A =	C30
C30+ 7	=	C37 x A =	7A26
7A26+ 8	=		7A2E

El paso de decimal a binario y viceversa, se hace: en el primer caso, multiplicando por potencias de dos, y en el segundo, dividiendo sucesivamente por dos.

I.VI. Identificación

Hasta ahora se emplearon números binarios, decimales y hexadecimales sin especificar a qué sistema pertenecían, lo cual, en algún caso, puede dar lugar a confusiones. Para ello se suelen emplear sufijos al final del número, que determinan a que sistema pertenece dicho número. Estos sufijos son: "d" para decimal, "h" para hexadecimal, "b" para binario (aunque no suele ser necesario indicarlo) y "o" ó "q" para el octal (dato puramente anecdótico).

31278d   7A2Eh   00101110b

De todos modos, a veces es más claro no poner los sufijos que ponerlos, ya que en muchos casos no es necesario y puede contribuir a dificultar algo que ya de por si es confuso.

Otra cosa a tener en cuenta es que los números hexadecimales que empiezan con una letra, se suelen representar poniendo un cero delante. Esto es necesario cuando se trata de un texto que se va a compilar, porque es la manera de que el compilador sepa al empezar a leerlo que aquello es un número; luego, a partir de la letra del final, sabrá en que formato está representado.

ANEXO II

SISTEMAS DE REPRESENTACIÓN

II.I. Números negativos

Con un número binario de ocho dígitos se pueden representar los números del 0 al 255 en decimal. A partir de aquí, si el primer dígito es un cero, significa que el número es positivo, y negativo en caso contrario. Esto significa que con ese número binario de ocho dígitos, en lugar de representar los números del 0 (00000000) al 255 (11111111), se representarán ahora los números del 0 (00000000) al 127 (01111111) y del -1 (11111111) al -128 (10000000). Pero ¿por qué son esos y no otros los números negativos? Pues porque para pasar de un número positivo a uno negativo o viceversa, se invierten todos los dígitos que lo forman y al resultado se le suma uno. Aquí hay unos ejemplos:

00000001 (1) -> 11111110 + 1 -> 11111111 (-1)

00111111 (63) -> 11000000 + 1 -> 11000001 (-63)

01111111 (127) -> 10000000 + 1 -> 10000001 (-127)

Este sistema tiene una ventaja y es la de que si efectuamos la suma entre un número positivo y uno negativo, el resultado es correcto, o sea que es igual a la resta de ambos números sin tener en cuenta el signo:

01111111 (127)	01000000 (64)	10000001 (-127)
+ 11000001 (-63)	+ 11000001 (-63)	+ 11111111 (-1)
*1*01000000 (64)	*1*00000001 (1)	*1*10000000 (-128)

El uno de la izquierda (en cursiva) se desborda por la izquierda ya que en este ejemplo se trabaja con 8 bits y no hay sitio para el noveno, por lo que no se tiene en cuenta.

Visto así, quizá la cosa no parece tan complicada, pero las cosas siempre tienden a torcerse y el caso de los números negativos no es una excepción. En este ejemplo, se trabajó con ocho bits, lo que viene a representar por ejemplo el registro AL, que más que un registro, es la parte inferior (8 bits=1 byte) del registro EAX (32 bits=4 bytes). Si se trata con 32 bits, o sea con un registro completo, los números positivos estarán comprendidos entre 00000001 y 7FFFFFFF y los negativos entre 80000000h y FFFFFFFF. Y es que todo depende del entorno, por lo que un número puede ser positivo o negativo según la referencia en el que se sitúa. Por ejemplo, si AX (16 bits=2 bytes) es igual a FFFF, significa que es igual a -1, pero si EAX es igual a 0000FFFF, esto implica que es igual a 65535. De la misma manera, si se suma FF a AL, se está sumando -1 (o dicho de otra manera, restándole uno), pero si se suma esta cantidad a EAX, se está sumando 255.

II.II. Coma (o punto) Flotante

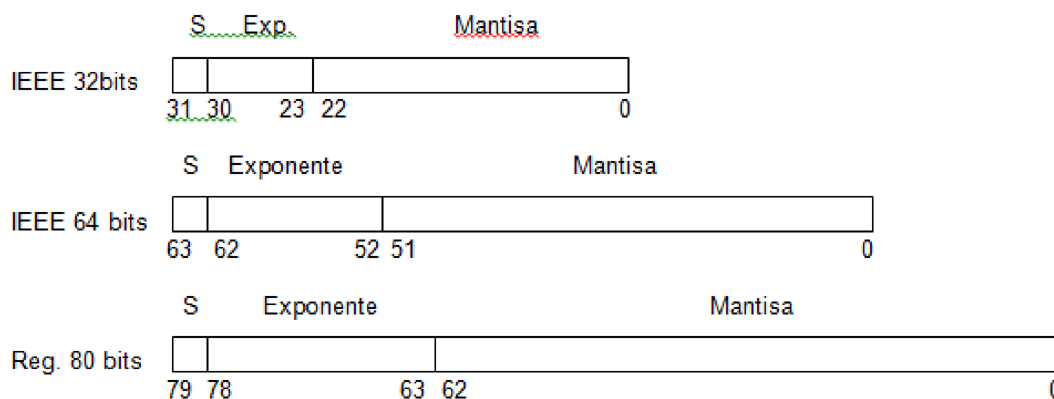
De la misma manera que no se puede poner un guion para representar un número negativo en binario o hexadecimal, tampoco se puede poner una coma para separar la parte entera de la decimal. Para solucionar esto se emplea la representación en coma flotante, mucho más versátil que la representación en coma fija.

Hasta la llegada del procesador 80486, las operaciones de coma flotante las efectuaba el coprocesador matemático. Para ello contaba con unos registros y unas instrucciones propias. A partir de la aparición del 80486, estas funciones han quedado incorporadas a la CPU.

De momento interesa saber que se trata de 8 registros de 80 bits, numerados del ST o ST(0) al ST(7) y cuya composición se detalla a continuación.

Existen tres formatos de números reales en coma flotante: de 32, 64 y 80 bits. Los dos primeros son formatos standard IEEE (Institute of Electrical and Electronics Engineers) para simple y doble precisión, y el tercero es el formato de los registros de coma flotante.

La representación de estos números está formada por tres elementos: signo, exponente y mantisa. El signo está indicado por un bit. El exponente por 8, 11 y 16 bits según el formato y la mantisa por 23, 52 y 63 bits; según el gráfico siguiente:



A continuación se verán unos ejemplos de cómo pasar números reales a formato en coma flotante de 32 bits (El procedimiento para los formatos más largos es similar).

Ejemplo nº 1: 105,28

En primer lugar, hay que pasar los números de la parte entera y decimal a binario. Ya se explicó la forma de pasar a binario la parte entera:

105:2 = 52 resto= 1 1

52:2 = 26 resto= 0 0

26:2 = 13 resto= 0 0

13:2 = 6 resto= 1 1

6:2 = 3 resto= 0 0

3:2 = 1 resto= 1 1

1:2 = 0 resto= 1 1

1101001

Ya tenemos 7 números, ahora sólo faltan 17 para completar los 24 números que se necesitan para la mantisa (Son necesarios 24 porque se elimina el primer número).

La parte decimal se pasa a binario multiplicando la parte decimal por 2 y con la parte entera del resultado se va formando el número binario:

0,28x2 = 0,56   entero=0   0

0,56x2 = 1,12   entero=1   1

0,12x2 = 0,24   entero=0   0

0,24x2 = 0,48   entero=0   0

0,48x2 = 0,96   entero=0   0

0,96x2 = 1,92   entero=1   1

0,92x2 = 1,84   entero=1   1

0,84x2 = 1,68   entero=1   1

0,68x2 = 1,36   entero=1   1

0,36x2 = 0,72   entero=0   0

0,72x2 = 1,44   entero=1   1...

y así hasta 17 ...   01000111101011100

Componemos el binario con decimales:

1101001 , 01000111101011100

Ahora corremos la coma hacia la izquierda hasta que no quede parte entera, en este caso, siete lugares:

0,110100101000111101011100

Eliminamos la coma y el primer uno del número y ya tenemos los 23 bits de la mantisa:

10100101000111101011100

Ahora vamos a por el exponente. Hemos corrido siete lugares a la izquierda, esta cifra la sumaremos a 126 (número constante) y pasaremos a binario el resultado:

7+126 = 133d = 10000101b

Ya tenemos también el exponente. El signo es positivo por lo tanto el bit que correspondiente al signo estará puesto a cero. Ya tenemos el número completo, es el:

0 10000101 10100101000111101011100

Ahora vamos a pasar ese número a hexadecimal y comprobaremos cual es el valor almacenado en el registro (Con un desensamblador).

0100 0010 1101 0010 1000 1111 0101 1100

4 2 D 2 8 F 5 C

C7035C8FD242 mov dword ptr [ebx], 42D28F5C

D903 fld dword ptr [ebx]

Si ejecutamos este código, y ponemos un bpx a la dirección de la instrucción fld, veremos cómo después de ejecutarse dicha instrucción, aparece en la ventana de los registros de coma flotante el valor 105,279998779296875 en el registro ST(0). El error viene dado por el nivel de precisión que hemos empleado. Si hubiéramos trabajado con 64 u 80 bits, el resultado hubiera sido más ajustado.

Ejemplo nº 2: Pasar un número en coma flotante a decimal.

3B8B4395=0 01110111 00010110100001110010101

Vemos por el bit de signo que se trata de un número positivo.

El exponente es 01110111=119. Le restamos 126 y nos da -7.

Añadimos un uno a la izquierda de la mantisa y le ponemos una coma delante. A continuación pasamos esta mantisa a decimal (Lo más sencillo es pasarla con lápiz y papel a hexadecimal y luego con la calculadora científica de Window$, a decimal) y nos da como resultado:

0 , 1000 1011 0100 0011 1001 0101 = 8B4395h = 9126805d

Ahora movemos la coma a la derecha, los lugares necesarios para que nos quede un número entero (en este caso, 24) y restamos el número de desplazamientos al exponente que teníamos (-7 -24 = -31). Así que el número resultante es:

$$9126805 \times 2^{-31} = 0{,}0042499997653067111968994140625$$

Resultado obtenido mediante la calculadora científica de Window$.

II.III. Formato BCD

Se trata simplemente de una manera de almacenar números decimales. Cada dígito decimal se pasa a binario y se guarda en un nibble (cuatro bits). Estos números binarios estarán comprendidos entre el 0000 y el 1001 (9), no siendo válido un número superior. A partir de aquí, hay dos maneras de guardar estos números: BCD empaquetado y BCD desempaquetado. En el formato empaquetado se guardan dos dígitos en un byte, en el desempaquetado se guarda un dígito en un byte, del cual quedan a cero los cuatro bits de mayor valor. Veremos un par de ejemplos:

Decimal	BCD empaq.	BCD desemp.
76	0111 0110	00000111 00000110
91	1001 0001	00001001 00000001

Con números representados en este formato, se pueden realizar operaciones aritméticas, pero se efectúan en dos tiempos. En el primero se lleva a cabo la operación aritmética, y en el segundo, se ajusta el resultado.

II.IV. Caracteres ASCII - ANSI

Aunque no es un tema directamente relacionado con el lenguaje ensamblador, es bueno conocer algo sobre cómo se las arregla el ordenador para escribir caracteres en la pantalla, mandarlos a la impresora, etc. Lo hace asignando un código a cada carácter que se quiera representar. Así nació en su día el juego de caracteres ASCII (American Standard Code for Information Interchange), que algunas veces se identifica como OEM (Original Equipment Manufacturer), compuesto por 256 caracteres, que era el utilizado por el MS-DOS. Window$ en cambio, utilizó el ANSI (American National Standards Institute) que consta de 224 caracteres, y posteriormente, en las versiones NT el Unicode, que al tener dos bytes de longitud en vez de uno, como los sistemas anteriores, es capaz de representar 65536 caracteres.

Los 32 primeros caracteres (del 0 al 31) del formato ASCII son códigos de control, como por ejemplo el 13 que equivale a CR (Enter), o el 8 que equivale a BS (Retroceso). Los caracteres del 32 al 127 son idénticos en ASCII y ANSI (El juego de caracteres ANSI empieza en el 32). Finalmente los caracteres del 128 al 255 (llamados también extendidos) son distintos para ambos sistemas y varían según los sistemas operativos o configuraciones nacionales.

Los 96 caracteres comunes en ambos sistemas son los siguientes:

Dec.	Hex.	Caract.	Dec.	Hex.	Caract.	Dec.	Hex.	Caract.
32	20	esp	64	40	@	96	60	`
33	21	!	65	41	A	97	61	a
34	22	"	66	42	B	98	62	b
35	23	#	67	43	C	99	63	c
36	24	$	68	44	D	100	64	d
37	25	%	69	45	E	101	65	e
38	26	&	70	46	F	102	66	f
39	27	'	71	47	G	103	67	g
40	28	(	72	48	H	104	68	h
41	29	)	73	49	I	105	69	i
42	2A	*	74	4A	J	106	6A	j
43	2B	+	75	4B	K	107	6B	k
44	2C	,	76	4C	L	108	6C	l
45	2D	-	77	4D	M	109	6D	m
46	2E	.	78	4E	N	110	6E	n
47	2F	/	79	4F	O	111	6F	o
48	30	0	80	50	P	112	70	p
49	31	1	81	51	Q	113	71	q
50	32	2	82	52	R	114	72	r
51	33	3	83	53	S	115	73	s
52	34	4	84	54	T	116	74	t
53	35	5	85	55	U	117	75	u
54	36	6	86	56	V	118	76	v
55	37	7	87	57	W	119	77	w
56	38	8	88	58	X	120	78	x
57	39	9	89	59	Y	121	79	y
58	3A	:	90	5A	Z	122	7A	z
59	3B	;	91	5B	[	123	7B	{
60	3C	<	92	5C	\	124	7C	|
61	3D	=	93	5D	]	125	7D	}
62	3E	>	94	5E	^	126	7E	~
63	3F	?	95	5F	_	127	7F	

Es bastante frecuente encontrarse, al seguirle la pista a un serial number, con instrucciones en las que se le resta 30h al código ASCII de un número para obtener su valor numérico real, o bien se le suma 30h a un número para obtener el carácter ASCII que lo representa, o con otras que suman o restan 20h al código de una letra para pasarla de mayúscula a minúscula o viceversa.

ANEXO III

OPERACIONES LÓGICAS

Una de las funciones de la Unidad Aritmético Lógica (ALU), situada en el núcleo del procesador es la de realizar las operaciones lógicas con los datos contenidos en una instrucción del programa. Pero, ¿qué es una operación lógica?

Una operación lógica asigna un valor (TRUE o FALSE) a la combinación de condiciones (TRUE o FALSE) de uno o más factores (variables). Los factores o las variables que intervienen en una operación lógica solo pueden ser TRUE o FALSE. Y el resultado de una operación lógica puede ser, tan solo, TRUE o FALSE.

A continuación se verán cuatro operaciones lógicas que se ejecutan bit a bit según las tablas que se muestran a continuación.

III.I. AND

El resultado es 1 si los dos operandos son 1, y 0 en cualquier otro caso.

1 and 1 = 1

1 and 0 = 0

0 and 1 = 0

0 and 0 = 0

Ejemplo: 1011 and 0110 = 0010

III.II. OR

El resultado es 1 si uno o los dos operandos es 1, y 0 en cualquier otro caso.

1 or 1 = 1

1 or 0 = 1

0 or 1 = 1

0 or 0 = 0

Ejemplo: 1011 or 0110 = 1111

III.III. XOR

El resultado es 1 si uno y sólo uno de los dos operandos es 1, y 0 en cualquier otro caso

1 xor 1 = 0

1 xor 0 = 1

0 xor 1 = 1

0 xor 0 = 0

Ejemplo: 1011 xor 0110 = 1101

III.IV. NOT

Simplemente invierte el valor del único operando de esta función

not 1 = 0

not 0 = 1

Ejemplo: not 0110 = 1001

Estos ejemplos se hicieron empleando números binarios. En el caso de números hexadecimales lo más corriente es pasarlos antes a binario. O bien utilizar la calculadora de Window$ en formato científico.