CracksLatinoS! 2010

-* Eliminando la Shell de Sentinel *-

Programa	**Ferrocad**
Download	http://www.iproin.com/Descargas/Instalacion%20FerroCad.zip
Descripción	
Herramientas	OllyDbg v2.0, IDA, PUPE, Import Reconstructor.
Dificultad	Media
Compresor/Compilador	
Protección	Sentinel Shell
Objetivos	Hacer correr el programa sin mochila
Cracker	Lionel lionelgomezdu@yahoo.es **Fecha:** 26/01/10
Tutorial nº	

-** Introduccion **-

Hacía mucho tiempo que no escribía nada sobre todo por falta de tiempo, y aunque sigo con el problema del tiempo creo que ya iba siendo hora de devolver parte de lo que la lista me ha dado durante tantos años.

En esta ocasión el tutorial va a tratar sobre una protección con mochila, pero un tanto particular. Viene a ser algo similar a lo que es el HASP Envelope, pero en este caso aplicado a la mochila Sentinel. Es lo que llaman SENTINEL SHELL. Tengo que decir que sólo llegar a la conclusión de que el tipo de protección era ésta me llevó mucho tiempo, ya que la información que hay circulando por la red es mínima, y desde luego, no he encontrado nada en español.

Una vez que pude averiguar a qué me estaba enfrentando me vino muy bien un tutorial escrito en el año 2001 por un tal CyberHeg llamado Breaking the Shell, aunque la versión del Shell a la que nos vamos a enfrentar tiene algún truco más que la analizada en ese caso.

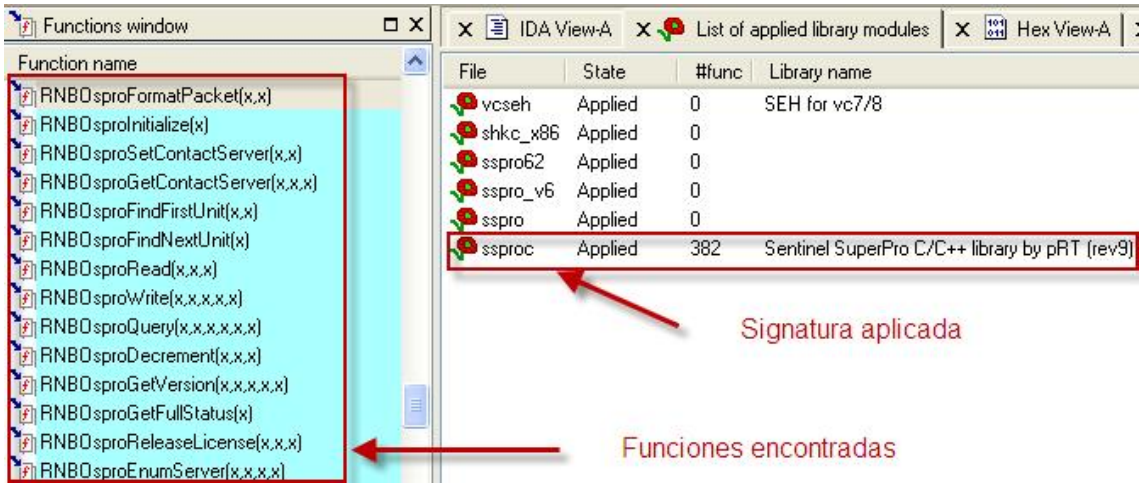
Vamos pues a comenzar el trabajo. Lo único que quiero aclarar es que voy a utilizar la nueva versión de Olly, la 2.0, que tiene como inconveniente el que no tiene plugins, pero así será más divertido y tendremos que enfrentarnos a las trampas a pecho descubierto.

-** Explorando el objetivo **-

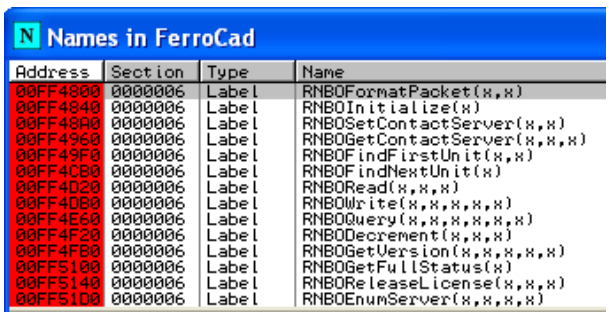
Tras instalar el programa lo ejecutamos normalmente y vemos que nos aparece la siguiente ventana:

Sabemos que utiliza una protección con mochila Sentinel porque en la página de descarga del programa aparece un enlace para bajarnos los drivers correspondientes. De todas maneras vamos a pasar el ejecutable por el RDG Packer Detector para ver si nos da alguna pista más, pero parece que no:

No nos queda más remedio que agarrarnos a lo único que sabemos e intentar exprimirlo al máximo. Lo que haremos antes de cargarlo en Olly es abrirlo con IDA, cargar un archivo de signaturas de Sentinel y que reconozca las funciones de la mochila para así poder analizarlo con Olly:



Ahí vemos todas las funciones que ha encontrado. Ahora podemos crear un MAP file para cargarlo con algún plugin de Olly, o bien como haré yo (ya que con este Olly no hay plugins), apuntar a mano las direcciones y nombre de cada función y en Olly añadir una etiqueta a mano en cada una y ponerle un BP a cada una:



Y ahora comienza lo que para mí fue lo realmente complicado. Al cargar el programa enseguida empieza a ejecutar un gran bucle que empieza en la dirección FF2861 y termina en FF2E0D. En cada pasada ECX toma un valor diferente, y dependiendo de ese valor ejecuta unos comandos u otros de ese bucle.

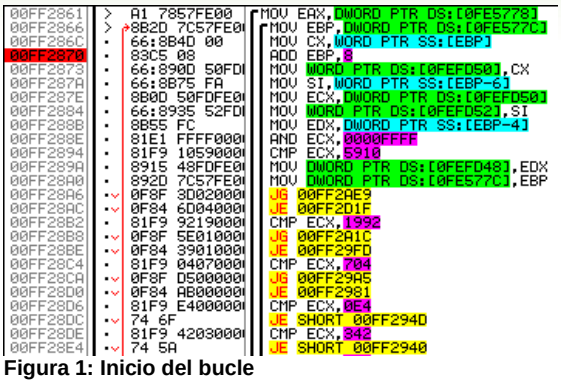


Figura 1: Inicio del bucle

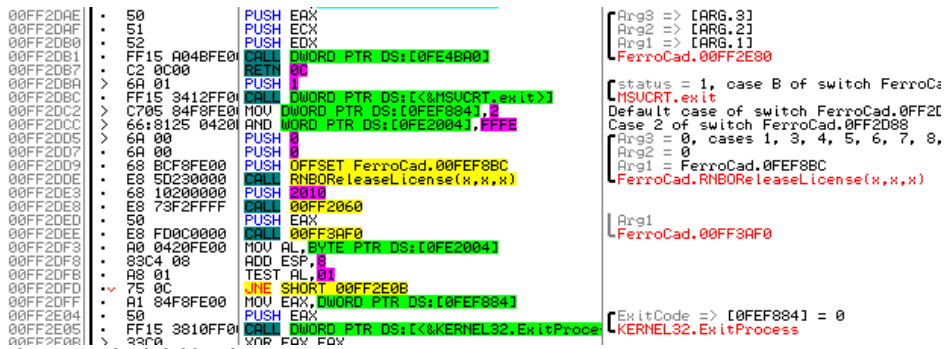


Figura 2: Final del bucle

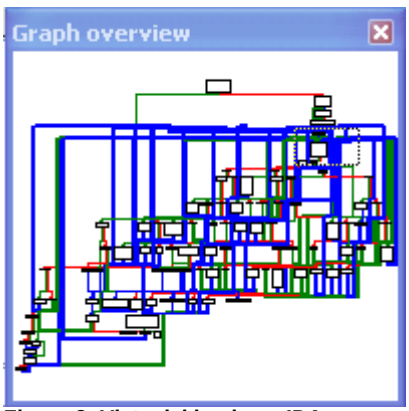


Figura 3: Vista del bucle en IDA

En la zona central pueden verse hasta 7 CALL EDX que en cada pasada del bucle llega con un valor distinto de EDX y por tanto nos lleva a una subrutina distinta, en las distintas pasadas del bucle una vez se pasará por alguno de esos CALL EDX y otras veces no. Esos CALL se encuentran en FF2962, FF29C9, FF2A78, FF2C59, FF2CA7, FF2CC1 y FF2CED

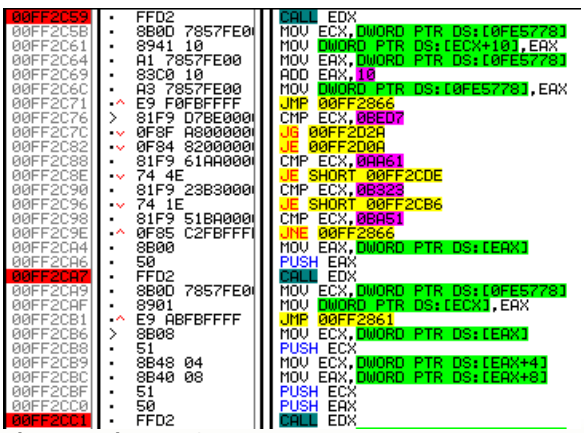


Figura 4: Algunos CALL EDX

También pude observar que había varios saltos condicionales que se tomaban o no dependiendo de la respuesta que recibiera de la emulación que pudiéramos hacer de la pastilla. Si la repuesta que hacíamos dar con nuestra emulación a la pastilla era correcta, el programa tomaba el salto adecuado y seguía la ejecución normal del programa. En caso contrario nos llevaba a la parte final del bucle donde nos sacaba del programa. Estos saltos son 4 y se encuentran en FF292F, FF2B44, FF2BB7 y FF2C10:

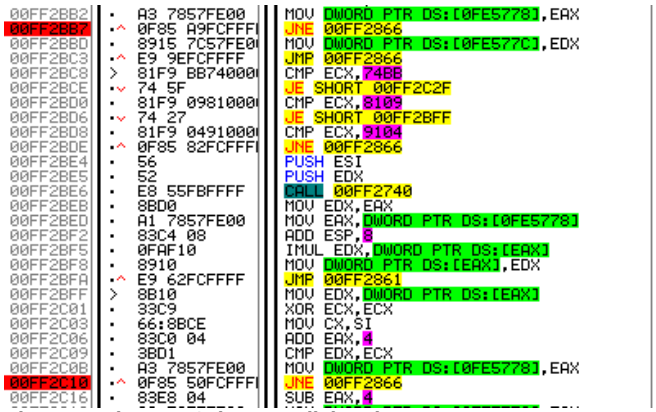
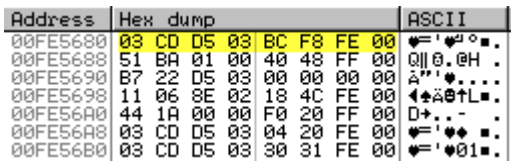


Figura 5: Algunos Saltos condicionales

Bueno pues así estuve dando palos de ciego y sin conseguir nada claro durante unos días. Puse BP en todas las funciones de la mochila, todas las llamadas CALL EDX y los 4 saltos condicionales que he mencionado. Al llegar a las funciones de Sentinel las emulaba como creía pero siempre llegaba a un punto en el que el programa me tiraba fuera o bien repetía las mismas instrucciones una y otra vez.

-** Empezando a comprender **-

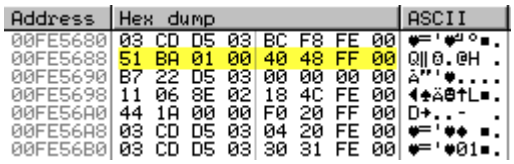
Como parecía que la clave estaba en los valores que iba tomando ECX empecé a buscar un patrón. Fue de casualidad que ví que en la primera pasada del bucle el primer valor que coge para ECX lo coge de FE5680:



Además en la línea FF288B pone en EDX lo que hay en FE5684 que vale FEF8BC

Cuando traceando llegué a FF2CA7 CALL EDX, ví que en la pila estaba precisamente el valor FEF8BC:

Así pude ver que cada vez que empezaba el bucle con un valor de ECX igual a CD03, esto hacía que cuando llegaba al siguiente CALL EDX el valor que había a continuación aparecía en la pila como argumento de esa subrutina. ¿Puede que estuviera ante una especie de máquina virtual? ¿Una especie de PCode casero?.
Seguí traceando y ví que cuando en ECX había el valor BA51 siempre se llegaba a FF2CA7 CALL EDX en el que EDX valía precisamente el valor almacenado 4 bits más adelante que ese BA51. En este caso:



Parece que había encontrado el patrón que buscaba que me hiciera comprender el funcionamiento. Así que me puse a rebuscar en la red y encontré esto:

```
courtesy of ketan:

ControlStruc Struc
val1            dw      ?               ; func id
val2            dw      ?               ; element id
val3            dd      ?               ; element
ControlStruc    Ends

; element id
0x068           *b
0x28e           *w
0x234           *d
0x3d5           d
0x0d4           d (index) into cElement (* dword array)

; func definitions
0x1a44, n/a, *func  ; cElement = func()
0xba51, n/a, *func  ; cElement = func(cElement)
0x1109, n/a, *func  ; cElement = func(cElement.04, cElement)
0xb323, n/a, *func  ; cElement = func(cElement.08, cElement.04, cElement)
0xaa61, n/a, *func  ; cElement = func(cElement.0C, cElement.08, cElement.04, cElement)
0x9b73, n/a, *func  ; cElement = func(cElement.10, cElement.0C, cElement.08, cElement.04, cElement)
0x00e4, n/a, *func  ; cElement = func(cElement.14, cElement.10, cElement.0C, cElement.08, cElement.04, cElement)
0xcd03, element id, element             ; cElement  = element
0x1092, element id, element             ; cElement += element
0x22b7, element id, element             ; cElement -= element
0x9104, element id, element             ; cElement *= element
0xfb52, element id, element             ; cElement /= element
0x3345, element id, element             ; cElement &= element
0x654d, element id, element             ; cElement |= element
0x74bb, element id, element             ; cElement ^= element
0x698a, element id, element             ; element = 0
0x1992, element id, element             ; element = cElement
0x5211, element id, element             ; element = cElement
0x0704, element id, element             ; element++
0x3dc8, element id, element             ; element--
0x5910, n/a, lpControlStruc             ; clpControlStruc = lpControlStruc
0x7123, element id, lpControlStruc      ; if (cElement == 0) then clpControlStruc = lpControlStruc
0x0611, element id, lpControlStruc      ; if (cElement != 0) then clpControlStruc = lpControlStruc
0x2245, index, n/a                      ; clpControlStruc = cElement, cElement += index*4
0x0342, index, value                    ; cElement[index*4] = value
0xbed7, n/a, n/a                        ; cElement = clpControlStruc
0x8109, id, n/a                         ; if (cElement == id) then cElement = clpControlStruc
0x65ab, id, n/a                         ; if (cElement != id) then cElement = clpControlStruc
0x096f, exitcode, n/a                   ; if (exitcode == 0) then SUCCESS, else display error message & exit
```

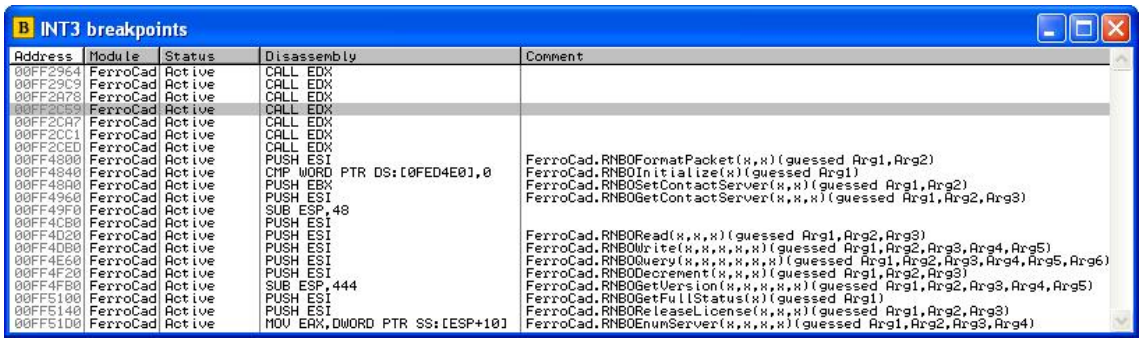
Que no es ni más ni menos que lo que estábamos buscando.

Resumiendo lo que tenemos básicamente es una máquina virtual que mediante el bucle que hemos descrito antes va leyendo las instrucciones que están codificadas en memoria entre las posiciones FE4BC8 y FE5780. Estas instrucciones se leen en bloques de 8 bits, en los que los 2 primeros indican en comando a ejecutar (ya vimos que por ejemplo CD03 significa PUSH), los 2 siguientes indican el tipo de argumento (byte, word, dword, etc) y los 4 últimos son el argumento de la función. De esta manera se va llamando a varias subrutinas, entre ellas

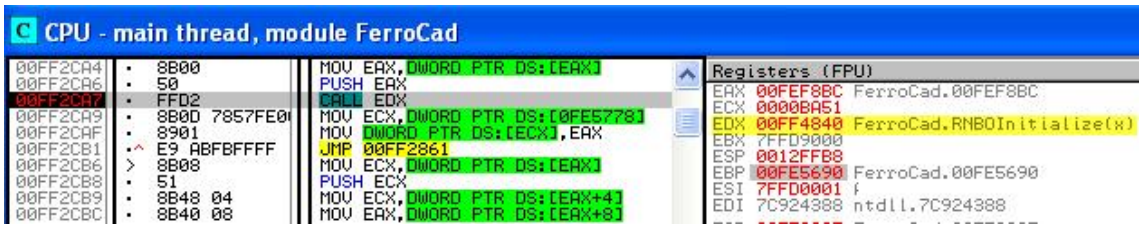
algunas funciones de la mochila Sentinel, de la que se obtendrán los datos necesarios para desencriptar parte del código del ejecutable. Una ver se haya desencriptado, este bucle nos hará saltar al OEP, desde el que podremos dumpear, y así tendremos el programa original que podrá ejecutarse sin mochila, ya que las llamadas a la mochila se hacen desde el bucle inicial que no volverá a llamarse.

- Primera función a emular: RNBOsproInitialize**-**

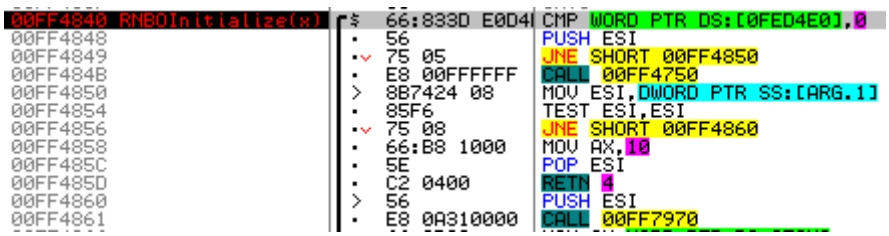
Ahora ya estamos en condiciones de saber cómo debemos trabajar. Para hacerlo reiniciamos Olly y ponemos BP en todas las funciones de Sentinel, y en todos los CALL EDX que vimos anteriormente. De esta manera tendremos controladas todas las llamadas a rutinas fuera del bucle principal.



En principio con eso bastaría. Ahora pulsamos F9 y enseguida llegamos a FF2CA7:



Vemos que EDX vale en esta ocasión FF4840, que es donde está situada la función RNBOInitialize(x). Si pulsamos F9 de nuevo apareceremos al principio de dicha función:



Según el Sentinel Developer Guide:

Format

```
unsigned short int RNBOsproInitialize(
RB_SPRO_APIPACKET packet
);
```

Parameters

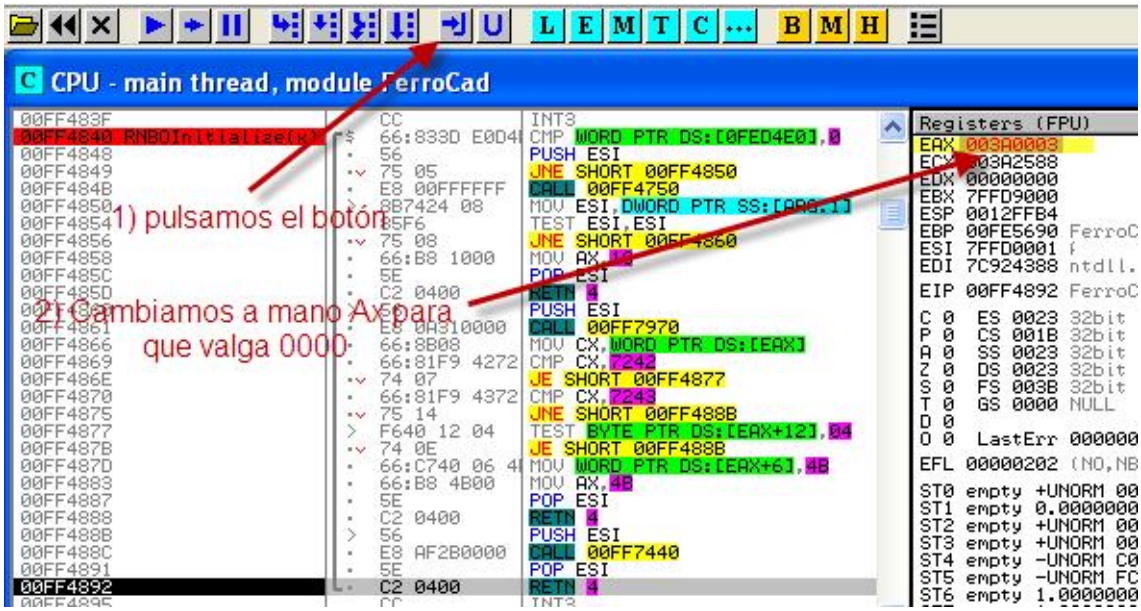
packet A pointer to the RB_SPRO_APIPACKET record.

Return Values

If successful, the function returns **SP_SUCCESS (0)**

Vemos que si la función tiene éxito, debe devolver 0. Eso equivale a poner AX=0. No vamos a hacer una emulación propiamentet dicha modificando el código del programa, porque nuestro

objetivo es quitar esta protección al programa original, así que si tenemos éxito, el programa no volverá a pasar por este punto.
Así pulsamos Ctrl+F9 para ejecutar hasta el RET, y una vez allí situados modificamos manualmente AX para que valga 0:

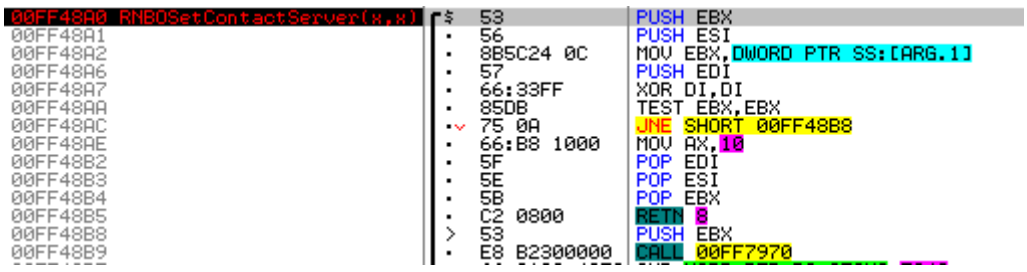


Después de cambiar AX:

Ya tenemos emulada la primera función.

- Segunda función: RNBOsproSetContactServer **-**

Ahora tendremos que pulsar varias veces F9 porque irá parando en algunos CALL EDX que de momento no nos interesan. Finalmente llegamos a:



Según el Sentinel Developer Guide:

Format

```
unsigned short int RNBOsproSetContactServer(
RBP_SPRO_APIPACKET packet,
char *serverName
);
```

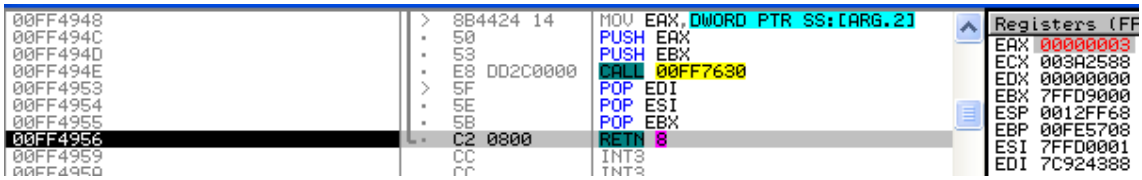
Parameters

packet A pointer to the RB_SPRO_APIPACKET record.
serverName The name to which the contact server is set.

Return Values

If successful, the function returns **SP_SUCCESS(0)**.

Vemos que si hacemos Ctrl+F9 para ir al RET de la función y ponemos AX=0 la tendremos emulada:

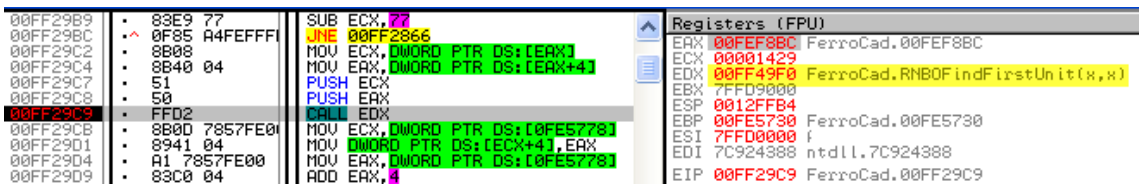


Después de editar EAX:

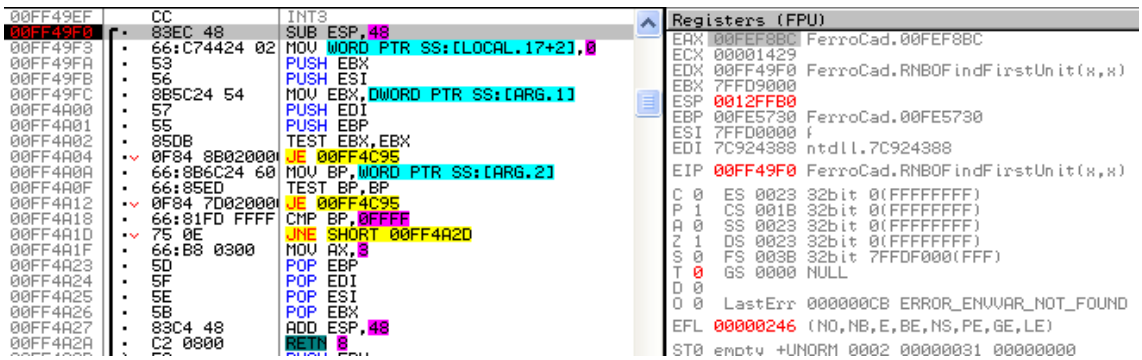
Ya tenemos la 2ª función emulada, esto va bien.

- 3ª función: RNBOsproFindFirstUnit **-**

Pulsamos F9 para continuar y llegamos a:



Vemos que en EDX está la dirección de la función RNBOFindFirstUnit. Pulsamos F9 una vez más y llegamos a:



Sabemos que:

Format

```
unsigned short int RNBOsproFindFirstUnit(
RB_SPRO_APIPACKET packet,
unsigned short int developerID
);
```

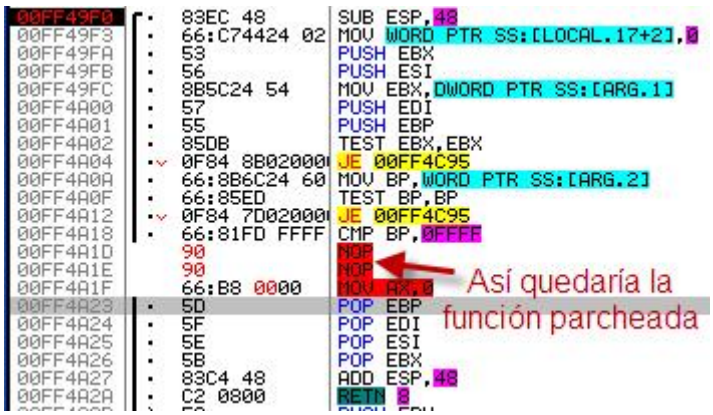
Parameters

packet A pointer to the RB_SPRO_APIPACKET record.
developerID This is assigned to you by Rainbow Technologies or your distributor. It identifies the SentinelSuperPro device to search for.

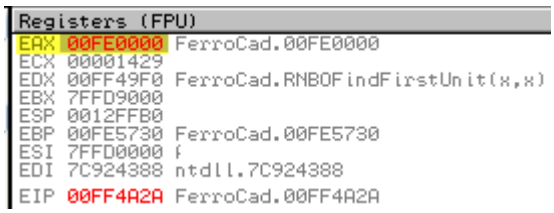
Return Values

If successful, the function returns **SP_SUCCESS (0)**.

Por lo que de nuevo lo que tenemos que hacer es hacer que AX valga 0 cuando estemos en el RET, pero esta vez sí vamos a modificar el ejecutable ya que de lo contrario comienza con llamadas anidadas a funciones de Sentinel y es un poco lioso salir de ahí sin perderse. Las modificaciones a hacer son las siguientes (recordar que EIP sigue estando al principio de la función):



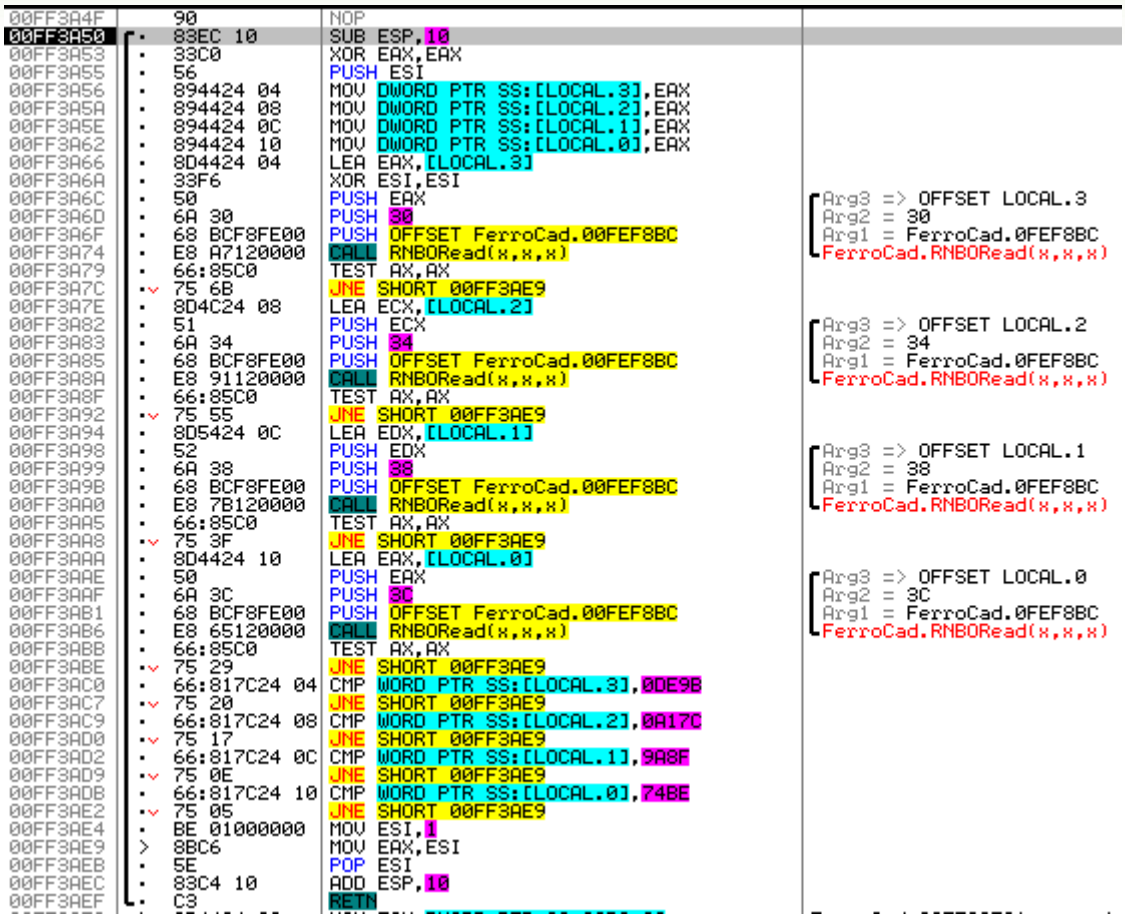
Ahora si pulsamos Ctrl+F9 vemos que salimos de la función con AX=0:



Vamos a por la siguiente.

- Primera trampa: Llamadas a RNBOsproRead **-**

Si continuamos con F9 paramos enseguida en FF2A78 CALL EDX en el que EDX vale FF3A50. Si entramos en esa subrutina con F7 nos encontramos con esto:



Primero vemos qué significa cada argumento de la función RNBOsproRead:

Format

```
unsigned short int RNBOsproRead(
RB_SPRO_APIPACKET packet,
unsigned short int address,
unsigned short int *data
);
```

Parameters

packet A pointer to the RB_SPRO_APIPACKET record.
address The SentinelSuperPro key memory cell address of the word to read.
data A pointer to the location that will contain the data read from the SentinelSuperPro key.

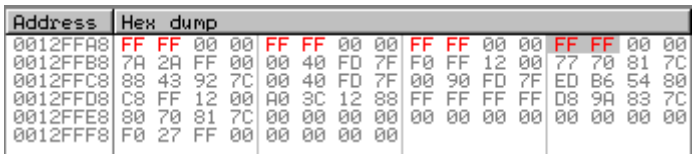
Return Values

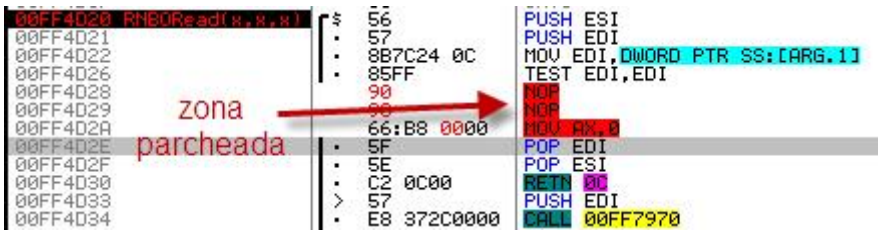
If successful, the function returns **SP_SUCCESS(0)**.

Vemos que el primer argumento es un puntero a APIPACKET, el segundo es la dirección de la celda de la que se quiere leer el Word, y el tercero es un puntero a la dirección donde se escribirá el dato leído. Si la función tiene éxito, escribirá el Word correspondiente en la dirección indicada por el puntero y además AX valdrá 0.

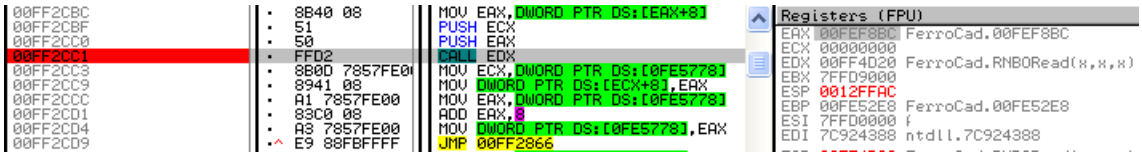
Lo primero que pensé cuando vi esto es que los Word de las celdas 30, 34, 38 y 3C debían valer DE9B, A17C, 9A8F y 74BE para cumplir con las comparaciones que se realizan a partir de FF3AC0. Sin embargo cuando hacía esto el programa me tiraba fuera. Yo creía que estaba haciendo mal la emulación, pero por más que le daba vueltas no veía qué hacía mal, hasta que pensé: Si estoy haciendo bien la emulación, puede que los datos que creo que debe leer no sean esos.

Así que lo que hice fue emular que en esas posiciones de celda hay FFFF en cada una. Me puse al principio de la función Read y apunté, en cada una de las veces que paró, en qué dirección iba a escribir el Word leído de la celda de la mochila. La primera vez fue en 12FFA8, después 12FFAC, 12FFB0 y 12FFB4. En cada una de esas direcciones escribí a mano FFFF, y esta vez antes que nada parcheé la función para que siempre saliera con AX=0:



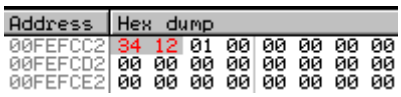


Con esto estaría solucionada también esta parte.
Al pasar los 4 Read que hemos visto, si pulsamos F9 de nuevo vamos a parar a:



En esta nueva llamada a RNBOsproRead se lee la celda de posición 0 y se escribe su valor en FEFCC2:

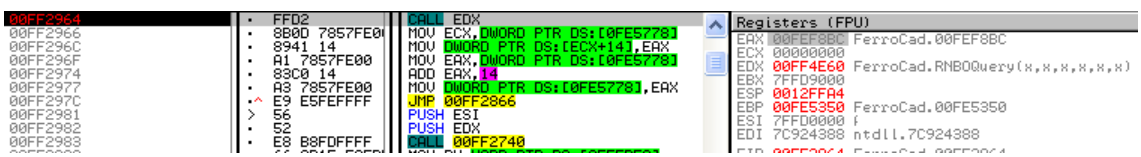
Según la documentación de Sentinel en la celda 0 está almacenado el número de serie de la llave, y éste es secuencial y varía desde 0000 a FFFF, por lo que podremos poner ahí cualquier número. Nosotros pondremos, una vez parados en el RET de la función, 1234 en la posición FEFCC2, y como la función la parcheamos anteriormente sólo comprobaremos que efectivamente salimos de la función con AX valiendo 0:



Ahora pulsamos F9 para continuar, y esta vez no nos saca fuera. Buena señal.

-** Empieza lo bueno: RNBOsproQuery **-

La siguiente parada es en FF2964:



Vemos que se va a llamar a RNBOsproQuery:

Format

```
unsigned short int RNBOsproQuery(
RB_SPRO_APIPACKET packet,
unsigned short int address,
VOID *queryData,
VOID *response,
unsigned long *response32,
unsigned short int length
);
```

Parameters

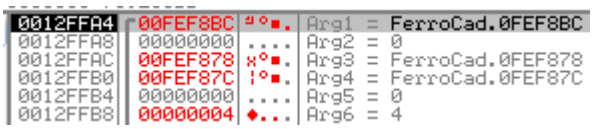
packet A pointer to the RB_SPRO_APIPACKET record.
address The address of the word to query.
queryData The pointer to the first byte of the query bytes.
response The pointer to the first byte of the response bytes.
response32 The pointer to the location that will contain a copy of the last four bytes of the query response.
length This is the number of query bytes to send to the active algorithm and also the length of the response buffer.

Return Values

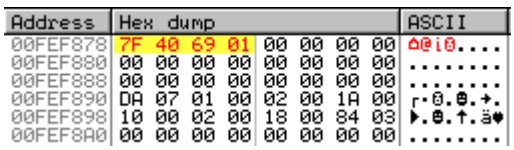
If successful, the function returns **SP_SUCCESS (0)**.

Esta función llama a un algoritmo activo en la celda especificada (segundo parámetro). A este algoritmo le pasa como argumento los datos situados en la dirección a la que apunta queryData (tercer argumento) y guarda el resultado en la dirección a la que apunta response (cuarto argumento). Tanto queryData como response tienen una longitud en bytes indicada por el argumento length. Si la función tiene éxito devuelve AX=0.

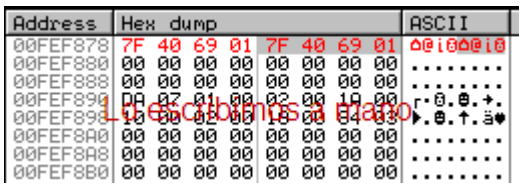
Ahora veremos nuestro caso particular:



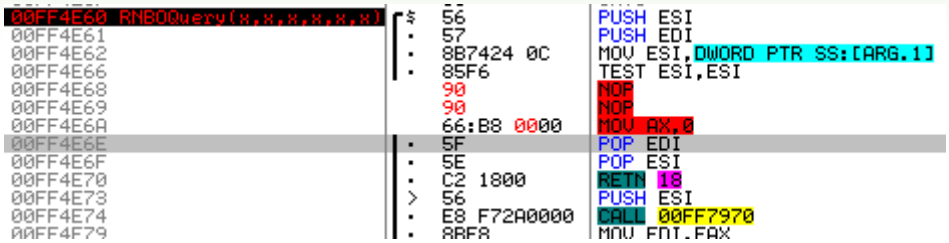
Vemos que el algoritmo se aplica sobre la celda 0. Este detalle es importante, porque para este algoritmo siempre se cumple que response = queryData. En nuestro caso length=4:



Podemos ver que queryData = 0169407F por lo que en FEF87C deberemos escribir a mano el mismo valor:



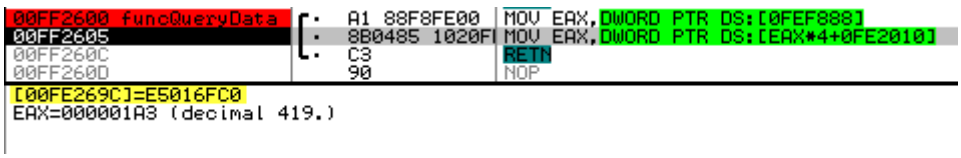
Hay que destacar que en cada ejecución del programa el valor de queryData se genera aleatoriamente y cambia. Además vamos a parchear la función para que siempre salga con AX=0:



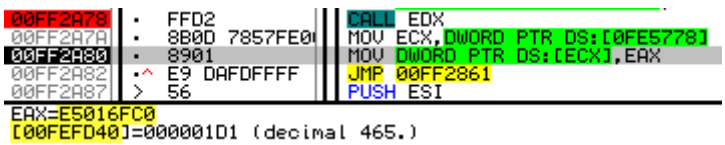
Con esto ya tendríamos resuelta la primera llamada a RNBOsproQuery.

-** Llegan las complicaciones: 2ª llamada a RNBOsproQuery **-

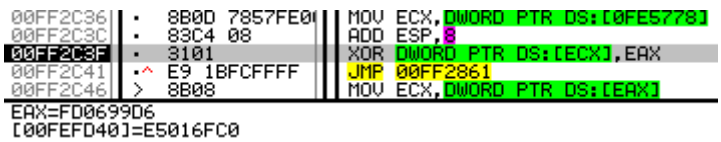
En este punto llegaremos a la segunda llamada a RNBOsproQuery. Esta va a ser bastante más complicada que la anterior. Veamos:
Si seguimos con F9 paramos primero en un CALL EDX en el que EDX vale FF2600. Si entramos en esa función (que yo he llamado funcQueryData) vemos que lo que hace es coger un número de una tabla que empieza en FE2010. Posteriormente he visto que el número que escoge de la tabla es aleatorio y cambia en cada ejecución del programa.



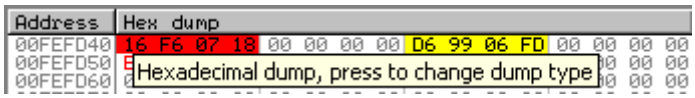
Traceando con F8 vemos que guarda el valor en FEFD40:



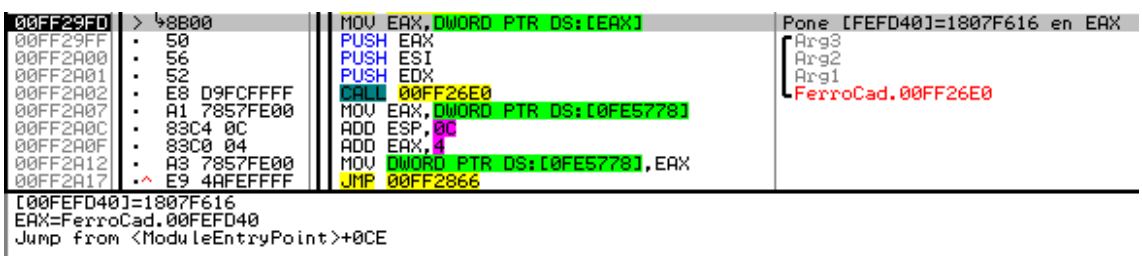
Vamos a ponerle un BPM en FEFD40 para ver qué hace con ese valor. Pulsamos F9 para continuar y



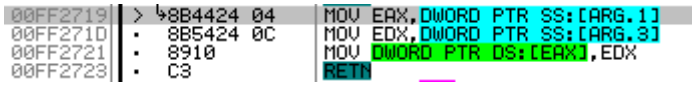
Olly para ahí por el BPM, debido a que va a Xorear el valor con FD0699D6 (este valor es constante y no varía en otras ejecuciones del programa), y guarda el resultado en FEFD40:



Continuamos la ejecución con F9 y llegamos a:

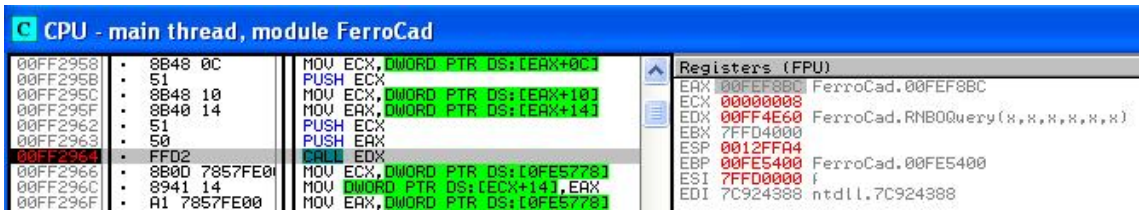


Donde guarda el valor en EAX, en el CALL FF26E0 lo que hace es intercambiar los valores de EAX y EDX y guardar el valor en FEF878. Así que ahora pondremos un BPM en FEF878 y además tendremos EDX=1807F616, pero inmediatamente se sobreescribe EDX con otro valor,

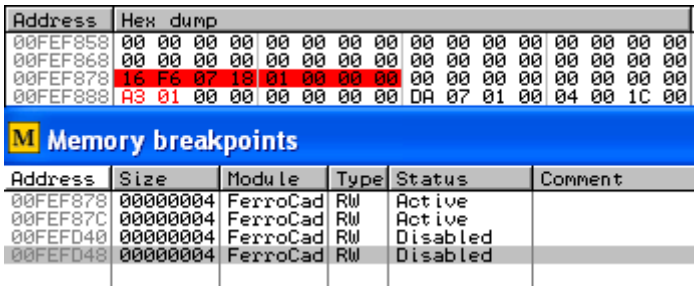


Así que podemos continuar con F9 hasta que llegamos a un punto donde se sobreescribe el valor que teníamos almacenado en FEFD40, por lo que podemos quitar ese BPM:

Damos a F9 otra vez y llegamos a la segunda llamada a RNBOsproQuery en la que se aplica el algoritmo sobre la celda 8 por lo que la facilidad que tuvimos antes de saber fácilmente cuál era el valor de response no la vamos a tener ahora. Además podemos ver que el argumento queryData es FEF878 que es precisamente donde tenemos guardado el resultado de aplicar Xor FD0699D6 al número obtenido de la tabla que está en FE2010:



Ahora el problema es ver cómo averiguamos la respuesta correcta que debe dar la mochila. En este caso queryData vale 1807F616. Así vamos a llegar hasta el RET de RNBOsproQuery ponemos AX=0 y ponemos un valor cualquiera en FEF87C y le pondremos un memory breakpoint:



Ahora pulsamos F9 y llegamos a:

En el que se pasa el contenido de FEF87C a EAX, si ahora traceamos con F8 llegamos a FF2D64:

```
00FF2D5F  .  E8 DCF9FFFF   CALL 00FF2740                    FerroCad.00FF2740
00FF2D64  .  8B0D 7857FE0( MOV ECX,DWORD PTR DS:[0FE5778]   Pone en ECX el valor [FE5778]=FEFD44
00FF2D6A  .  83C4 08       ADD ESP,8
00FF2D6D  .  83E9 04       SUB ECX,4
00FF2D70  .  A3 48FDFE00   MOV DWORD PTR DS:[0FEFD48],EAX   Almacena nuestro response en FEFD48
00FF2D75  .  890D 7857FE0( MOV DWORD PTR DS:[0FE5778],ECX
00FF2D7B  .  8901          MOV DWORD PTR DS:[ECX],EAX       Almacena nuestro response en FEFD40
00FF2D7D  .^ E9 DFFAFFFF   JMP 00FF2861                     Vuelve al inicio del bucle
```

En los comentarios de las lineas se explica lo que hace el código. Por lo que podemos ver, está guardando nuestra response fictica en FEFD44 y FEFD40, por lo que pondremos BPM en ambas posiciones de memoria (si no utilizais el Olly 2 podeis utilizar HBP):

```
Address   Hex dump
00FEFD40  01 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00
00FEFD50  03 CD 34 02 00 00 00 00 00 00 00 00 00 00 00 00
00FEFD60  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00FEFD70  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Memory breakpoints

Address	Size	Module	Type	Status	Comment
00FEF878	00000004	FerroCad	RW	Active	
00FEF87C	00000004	FerroCad	RW	Active	
00FEFD40	00000004	FerroCad	RW	Active	
00FEFD48	00000004	FerroCad	RW	Active	

Damos F9 de nuevo y vuelve a parar en:

```
00FF2894  .  81F9 1059000( CMP ECX,5910
00FF289A  .  8915 48FDFE0( MOV DWORD PTR DS:[0FEFD48],EDX
00FF28A0  .  892D 7C57FE0( MOV DWORD PTR DS:[0FE577C],EBP
00FF28A6  .v 0F8F 3D02000( JG 00FF2AE9
00FF28AC  .v 0F84 6D04000( JE 00FF2D1F
EDX=FerroCad.00FEF878
[00FEFD48]=1
```

Donde va a sobreescribir lo que tenemos en FEFD48, por lo tanto ya podemos quitar el BPM que teníamos ahí, ya que nuestro valor ya no está, sólo quedarán los otros 3:

Memory breakpoints

Address	Size	Module	Type	Status
00FEF878	00000004	FerroCad	RW	Active
00FEF87C	00000004	FerroCad	RW	Active
00FEFD40	00000004	FerroCad	RW	Active
00FEFD48	00000004	FerroCad	RW	Disabled

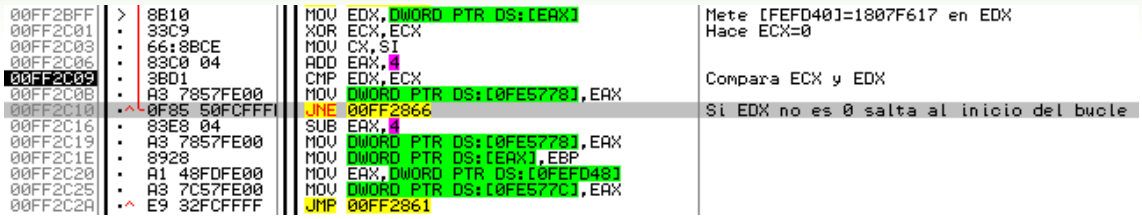
Continuamos con F9 y para debido al BPM en FEF878 llegamos a un punto en el que guarda el valor de queryData en EAX y posteriormente lo Xorea con nuestra response:

```
00FF2C36  .  8B0D 7857FE0( MOV ECX,DWORD PTR DS:[0FE5778]
00FF2C3C  .  83C4 08       ADD ESP,8
00FF2C3F  .  3101          XOR DWORD PTR DS:[ECX],EAX       Xorea nuestra response con 1807F616
00FF2C41  .^ E9 1BFCFFFF   JMP 00FF2861                     Vuelve al inicio del bucle
00FF2C46  >  8B08          MOV ECX,DWORD PTR DS:[EAX]
00FF2C48  .  51            PUSH ECX
EAX=1807F616
[00FEFD40]=00000001
```

El resultado (1807F617) se guarda en FEFD40.

```
Address   Hex dump
00FEFD20  00 00 00 00 00 00 00 00 00 00 00 00 04 00 00 00
00FEFD30  00 00 00 00 7C F8 FE 00 78 F8 FE 00 08 00 00 00
00FEFD40  17 F6 07 18 00 00 00 00 78 F8 FE 00 00 00 00 00
00FEFD50  BB 74 34 02 00 00 00 00 00 00 00 00 00 00 00 00
00FEFD60  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Esto es muy extraño, pero bueno, pulsemos F9 otra vez con lo que llegamos a FF2BFF:



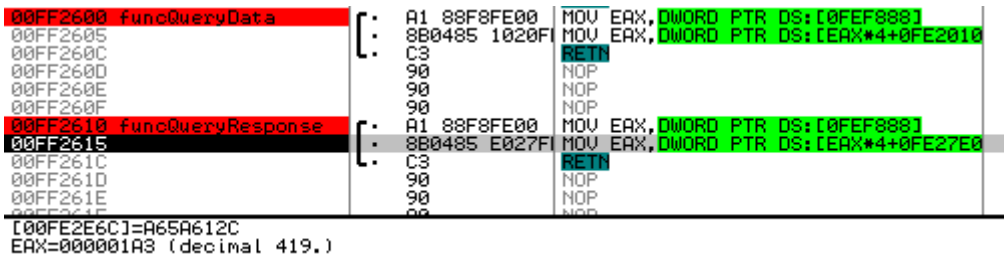
Lo que hace el código está comentado en cada linea, pero resumiendo lo que ha hecho el código es lo siguiente:

EDX = XOR response, queryData
CMP EDX, 0
JNE FF2866

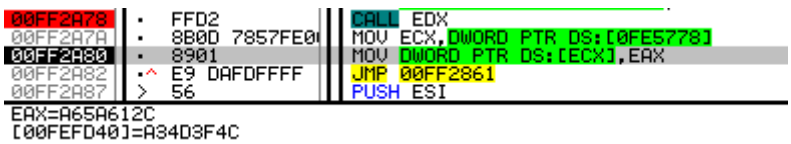
O lo que es lo mismo:

CMP response, queryData
JNE FF2866

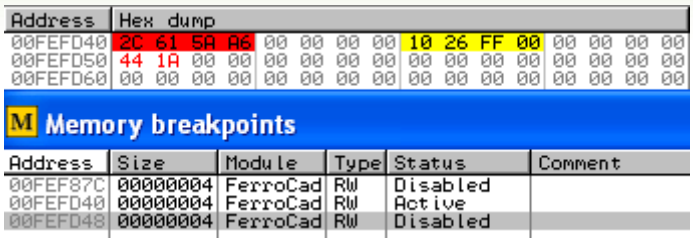
Vamos que nos está tendiendo otra pequeña trampa, ya que en este caso nunca va a ser response = queryData, ya que la celda sobre la que estamos aplicando la función Query no es la 0, sino la 8, por lo que siempre será response ≠ query Data. De hecho, si forzáramos a que el salto no se tomara el programa nos sacaría a FindNextUnit, lo que es señal que vamos por mal camino. Así que dejamos todo como está y continuamos con F9 y llegamos primero a un CALL EDX en el que EDX=FF2610:



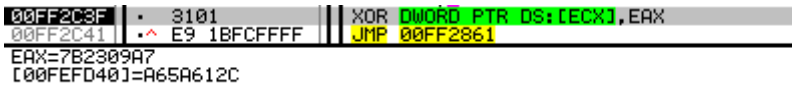
Curiosamente, esta función es muy similar a la que vimos antes que cogía un número aleatorio de una tabla para a partir de ahí empezar a calcular queryData, y se encuentra justo debajo Ahí se coge un número almacenado en FEF888 y que varía en cada ejecución y en base a ese número se coge el valor que se encuentra en una tabla que empieza en FE27E0. En este caso va a coger el valor que se encuentra en FE2E6C = A65A612C. Además, si nos fijamos, cuando llegó a la función funcQueryData el valor que había en FEF888 era el mismo que hay ahora, con lo que el elemento que va a coger de la tabla estará situado en una dirección FE27E0 – FE2710 = 7D0. Como tenemos el valor en EAX, vamos a tracear con F8 para ver qué hace con él:



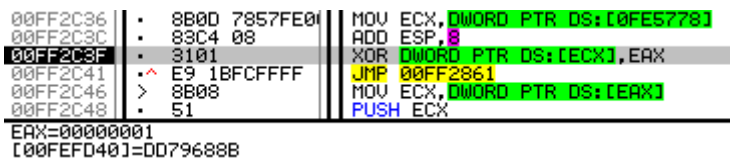
Vemos que guarda el valor en FEFD40, por lo que ponemos un BPM en esa posición de memoria:



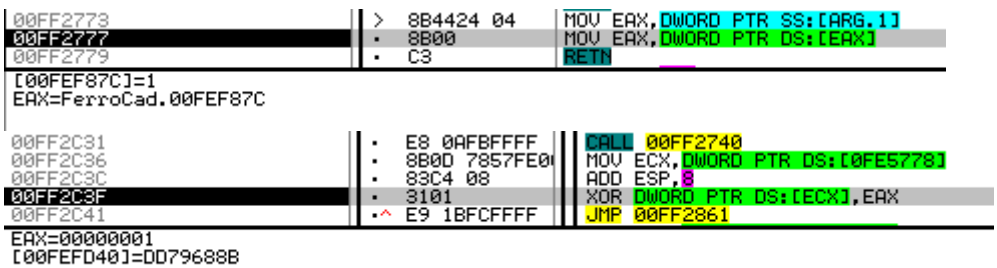
Y continuamos con F9 hasta que vuelve a parar en FF2C3F donde hace XOR del valor con 7B2309A7:



Después de muchas repeticiones sabemos que este valor con el que Xorea el valor obtenido de la tabla es siempre constante. El resultado de este XOR (DD79688B) sigue guardado en FEFD40 así que si seguimos con F9:



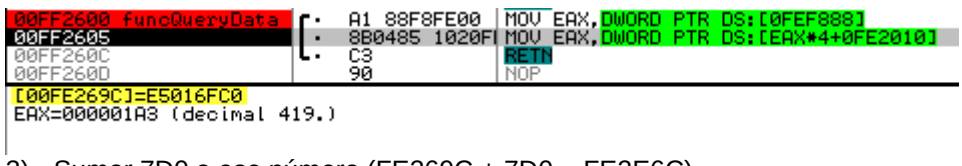
Llegamos al punto en que se compara ese valor con la response que pusimos a la función Query (1)



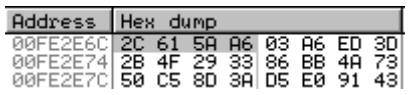
Si el resultado de este XOR no es 0 (si response ≠ DD79688B) entonces el programa nos tira a FindNextUnit, por lo que debemos hacer que sea 0, y entonces nos llevará al siguiente Query.

Resumiendo, para obtener la respuesta correcta al 2º RNBOsproQuery debemos hacer:

1) Tener un BP puesto en FF2600.
2) Cuando el programa llegue ahí apuntar la dirección de la que toma el dato, en nuestro caso ha sido FE269C:



3) Sumar 7D0 a ese número (FE269C + 7D0 = FE2E6C)
4) Vamos a esa dirección de memoria y apuntamos lo que hay (A65A612C):



5) Le aplicamos A65A612C XOR 7B2309A7 = DD79688B

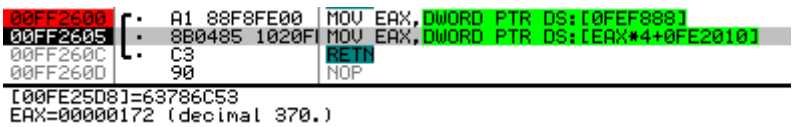
6) Apuntamos ese valor, y cuando lleguemos a RNBOsproQuery lo ponemos como response.

Con esto ya tendríamos emulado el 2º RNBOsproQuery.

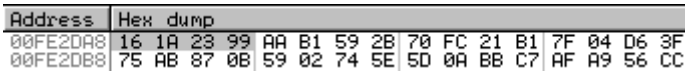
-** Repetimos: 3ª llamada a RNBOsproQuery **-

Una vez superada la 2ª llamada a RNBOsproQuery la tercera es muy fácil ya que se repite el mismo esquema que la anterior, así que no voy a volver a explicar todo el proceso, sino que lo haré según lo resumido antes:

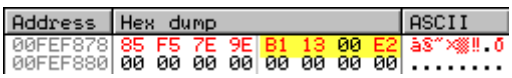
Una vez hemos emulado correctamente el 2º Query mantenemos el BP en la función FF2600. Así cuando seguimos con la ejecución después del Query anterior Olly para precisamente en esa función, pero esta vez cambia la posición de la que lee el número:



La posición es FE25D8. Si le sumamos 7D0: FE25D8 + 7D0 = FE2DA8. Si vamos a esa posición tenemos el valor 99231A16:



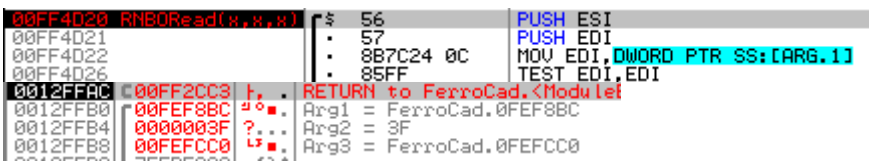
Hacemos 99231A16 XOR 7B2309A7 = E20013B1.
Pulsamos F9 para continuar hasta RNBOsproQuery y vemos que también aplica el algoritmo sobre la celda 8 por lo que en la dirección a la que apunta el parámetro response escribimos ese número:

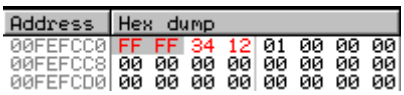


Llegamos hasta el RET del Query, ponemos EAX=0 y damos F9 y la ejecución continúa sin problemas:

-** Nueva llamada a RNBOsproRead **-

Esta vez lee el contenido de la celda 3F y lo guarda en FEFCC0. Lo solucionamos poniendo en esa posición el valor FFFF:



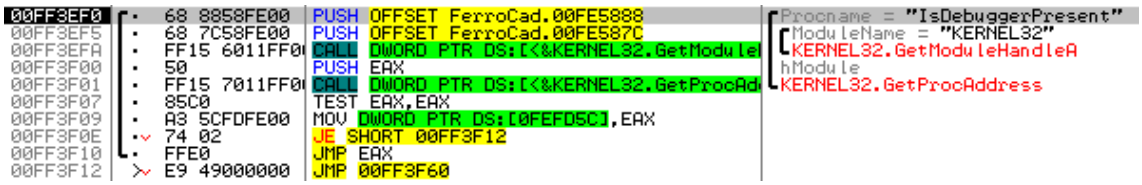


Y no se nos olvide poner AX=0 cuando lleguemos al RET.

Así damos F9 de nuevo y llegamos otra llamada a EDX.

-** Sencillo antidebug **-

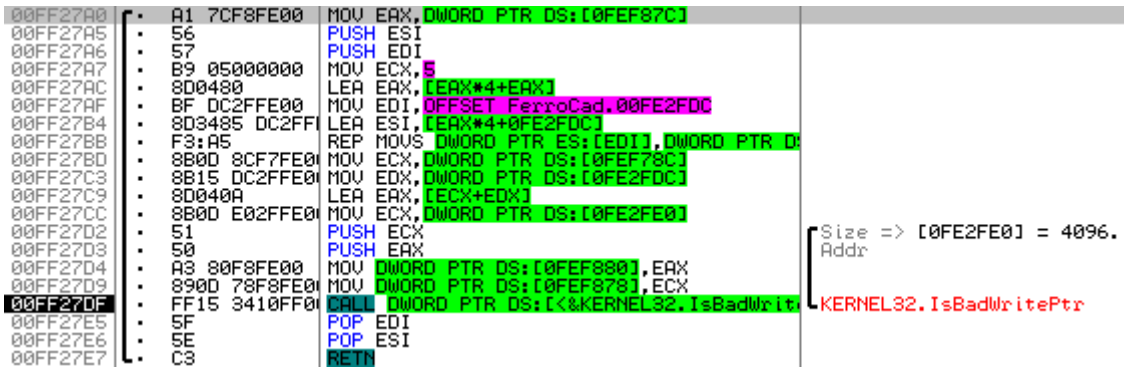
Este CALL EDX donde EDX vale FF3EF0 es donde se hace la única comprobación antidebug de la protección, mediante una simple llamada poco disimulada a IsDebuggerPresent:



Sabemos que a la salida de esa API debemos hacer que EAX=0. Y con esto ya queda solucionado el problema del antidebug.

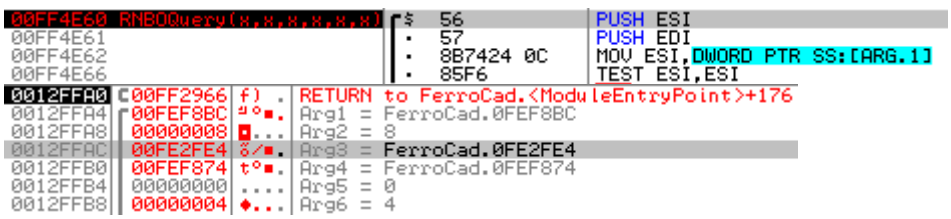
-** Más problemas: 4ª llamada a RNBOsproQuery **-

Pulsamos F9 y llegamos primero a un CALL EDX en el que se verifican permisos de escritura en una zona de memoria:

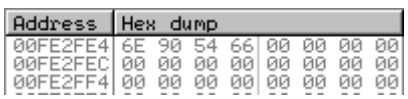


Aquí no debemos hacer nada, sólo pulsar F9 para llegar al 4º RNBOsproQuery. En este caso es diferente a los anteriores, ya que aunque también aplica el algoritmo Query sobre la celda 8, nos llama la atención que ni pasa por la función FF2600 para generar el queryData ni por la FF2610 para generar el response. Además utiliza el valor que le demos como response como key para desencriptar una zona del ejecutable. Por tanto, o averiguamos el valor correcto, o el ejecutable no se desencriptará bien y por tanto no funcionará. Veamos cómo hacerlo.

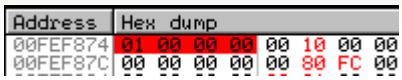
Nos encontramos aquí:



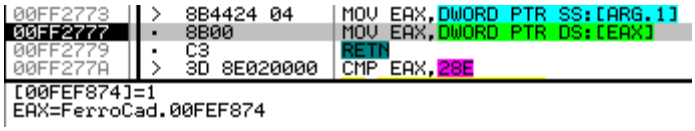
Como valor de QueryData tenemos el valor 6654906E:



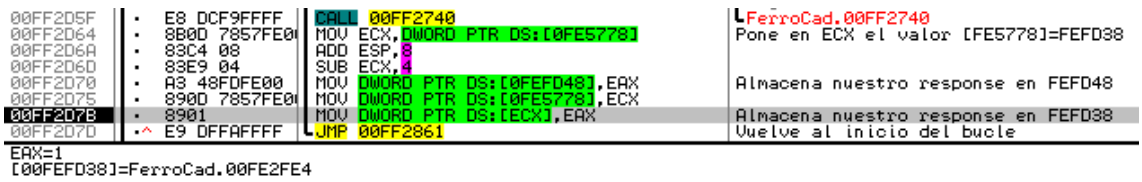
Este valor es constante en todas las ejecuciones del programa, un problema menos, pero no sabemos qué valor poner en FEF874. Pondremos un valor cualquiera y le pondremos un BPM para ver qué hace con él, además recordar como siempre poner AX=0 a la salida de la función:



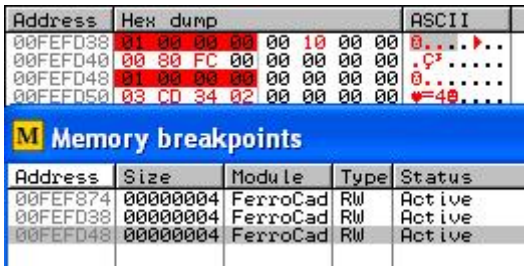
Si damos a F9 paramos en FF2777 donde se guarda en EAX el valor que hemos escrito en FEF874:



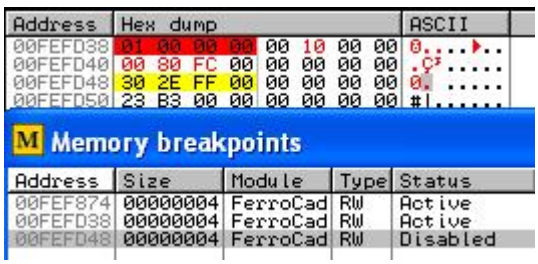
Si seguimos con F8 llegamos a:



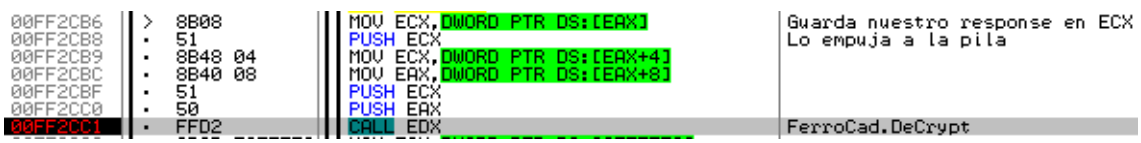
Así que ponemos BPM en FEFD38 y FEFD48:



Si pulsamos F9 para en una zona en la que se sobreescribe lo que hay en FEFD48 con otro valor por lo que podemos quitar ese BPM, con lo que nos quedamos sólo con los otros 2.

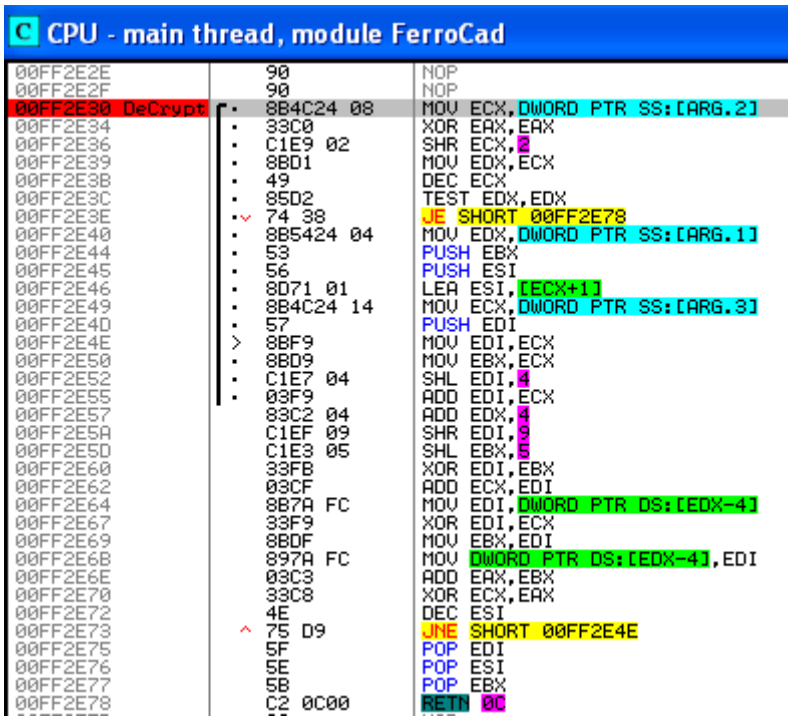


Ahora volvemos a pulsar en F9 y llegamos a un punto en que empuja nuestra response a la pila para después hacer una llamada a una función que yo he llamado Decrypt y que está situada en FF2E30:



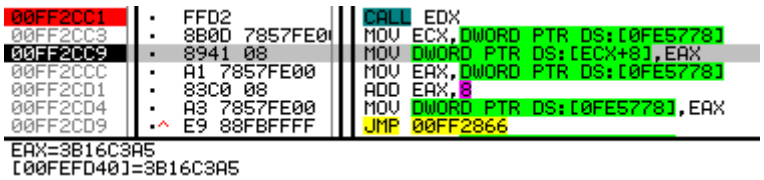
Esta función utiliza como argumentos 3 parámetros: Decrypt(direccion de memoria a desencriptar, longitud a desencriptar en bytes, key desencriptadora). Como keydesencriptadora se utiliza la response que hemos dado a la función RNBOsproQuery:

Si traceamos dentro de la funicón va haciendo una serie de operaciones sobre las direcciones de memoria, las sobreescribe y además va sumando los valores que va obteniendo en EAX, de manera que cuando la función llega al RETN 0C, tenemos un valor en EAX que varía según la key desencriptadora que utilicemos. :

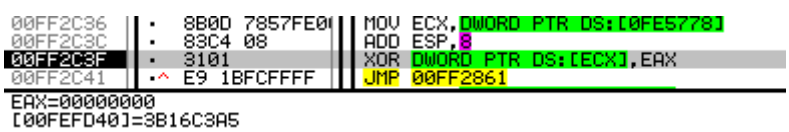


En este caso, EAX sale valiendo 3B16C3A5:

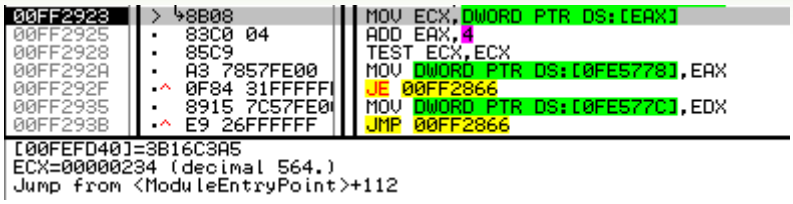
Sigamos traceando a ver si hace algo con este valor:



Pues en FF2CC9 parece que guarda el valor en FEFD40, así que le ponemos un BPM ahí también y continuamos llegamos a:



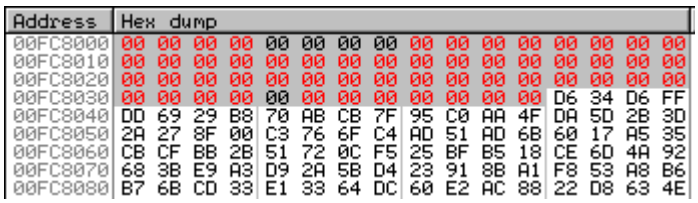
Donde aplica XOR a nuestro valor con 0 y guarda el resultado en FEFD40, y si damos a F9 parará en:



Donde guarda el resultado de ese XOR en ECX, y dependiendo de si ECX es 0 o distinto de 0, tomará o no el salto condicional que hay en FF292F. He comprobado que si ECX es distinto de 0 la ejecución nos saca fuera del programa, por lo que ya sabemos que para que todo vaya bien el valor de EAX cuando se sale de la función Decrypt debe ser 0.

Para conseguir esto solo puede hacerse con fuerza bruta. Una vez hecho esto, vemos que el valor de response en el 4ª RNBOsproQuery debe ser 28E3AE12.

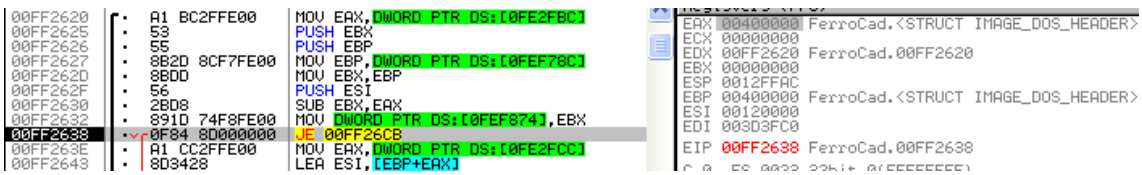
Si reiniciamos todo y volvemos a llegar a este 4ª RNBOQuery y ponemos en response el valor obtenido cuando llegamos a la función Decrypt vemos que poco a poco todos los valores que hay a partir de FC8000 se van transformando en ceros:



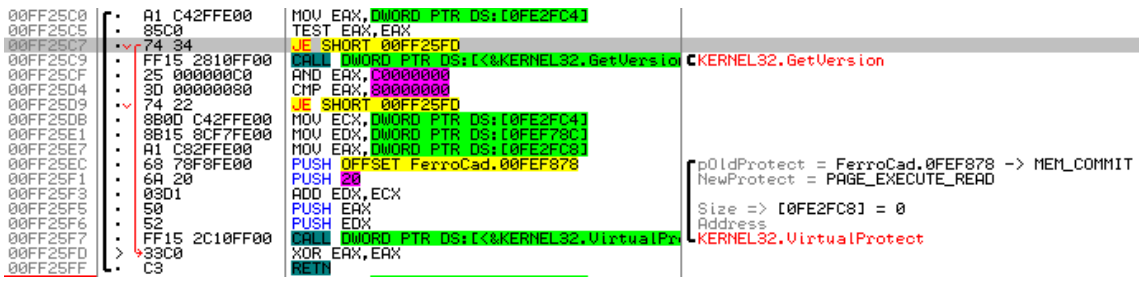
Así que ya tenemos el 4ª RNBOsproQuery solucionado y podemos quitar todos los BPMs.

-** Preparacion de la llegada al OEP y otro RNBOsproRead**-

Si continuamos la ejecución llegamos a un CALL EDX en que llama a una función que comprueba algo de la cabecera del ejecutable pero nosotros no tenemos que hacer nada, sólo continuar con F9:



La siguiente parada es en otro CALL EDX que llama a FF25C0 en que no hace nada:



La siguiente parada tras volver a pulsar en F9 es en otro CALL EDX más, esta vez para llamar a la función que está en FF2250 que creo que lo que hace es crear la IAT del verdadero ejecutable:

```
00FF2250  ┌·  83EC 1C          SUB ESP,1C
00FF2253  ·   8B0D 8CF7FE00    MOV ECX,DWORD PTR DS:[0FEF78C]
00FF2259  ·   A0 4431FE00      MOV AL,BYTE PTR DS:[0FE3144]
00FF225E  ·   53               PUSH EBX
00FF225F  ·   55               PUSH EBP
00FF2260  ·   8B59 3C          MOV EBX,DWORD PTR DS:[ECX+3C]
00FF2263  ·   56               PUSH ESI
00FF2264  ·   57               PUSH EDI
00FF2265  ·   33F6             XOR ESI,ESI
00FF2267  ·   8D3C19           LEA EDI,[EBX+ECX]
00FF226A  ·   C74424 18 0000(  MOV DWORD PTR SS:[LOCAL.4],0
```

A continuación hace otro CALL EDX esta vez para volver a llamar a RNBOsproRead para leer de nuevo la celda 3F y escribirla en FEFCC0. Igual que antes, nosotros escribiremos FFFF en esa dirección, cambiamos AX=0 al salir y continuamos con F9

```
00FF4D20 RNBORead(x,x,x)  ┌$  56           PUSH ESI
00FF4D21                  ·   57           PUSH EDI
00FF4D22                  ·   8B7C24 0C    MOV EDI,DWORD PTR SS:[ARG.1]
00FF4D26                  ·   85FF         TEST EDI,EDI
0012FFAC  00FF2CC3  ├, .  RETURN to FerroCad.<Modu
0012FFB0  ┌00FEF8BC  ╝°■.  Arg1 = FerroCad.0FEF8BC
0012FFB4  │0000003F  ?...  Arg2 = 3F
0012FFB8  └00FECC0  ╚F■.  Arg3 = FerroCad.0FEFCC0
```

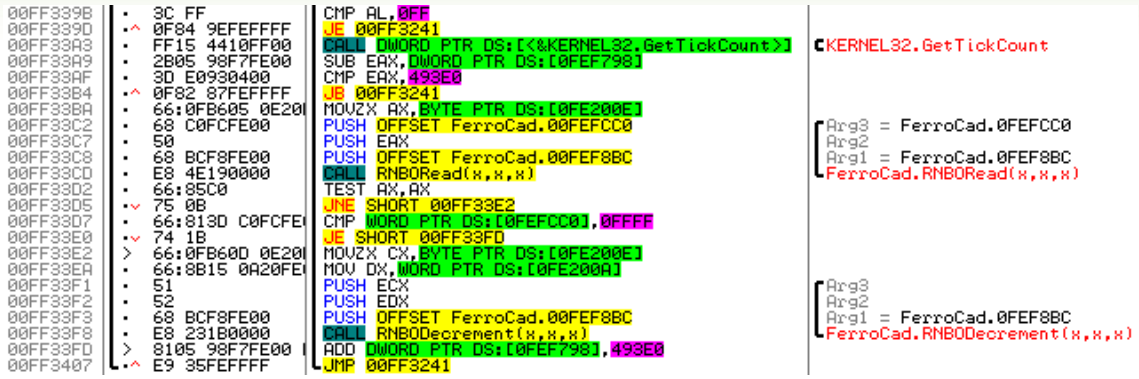
```
Address   Hex dump
00FEFCC0  FF FF 34 12 01 00 00 00 00 00 00 00 00 00 00 00
00FEFCD0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

-** Últimos obstáculos: Creación de hilos**-

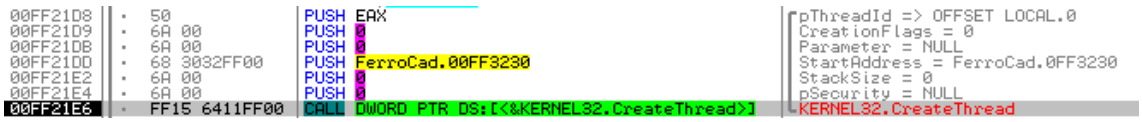
Si al salir del último RNBOsproRead damos a F9 llegamos un CALL EDX a la última función que intenta complicarnos la vida:

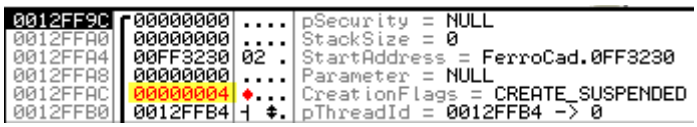
```
00FF2190 Crea_hilo_timer  ┌·  51               PUSH ECX
00FF2191                  ·   FF15 2810FF00    CALL DWORD PTR DS:[<&KERNEL32.GetVersion>]      KERNEL32.GetVersion
00FF2197                  ·   25 000000C0      AND EAX,C0000000
00FF219C                  ·   3D 00000080      CMP EAX,80000000
00FF21A1                  ·v  74 63            JE SHORT 00FF2206
00FF21A3                  ·   56               PUSH ESI
00FF21A4                  ·   FF15 C010FF00    CALL DWORD PTR DS:[<&KERNEL32.GetCurrentProcess>]  KERNEL32.GetCurrentProcess
00FF21AA                  ·   6A 02            PUSH 2                                          ┌Options = DUPLICATE_SAME_ACCESS
00FF21AC                  ·   6A 01            PUSH 1                                          │InheritHandle = TRUE
00FF21AE                  ·   8BF0             MOV ESI,EAX                                     │
00FF21B0                  ·   6A 00            PUSH 0                                          │DesiredAccess = 0
00FF21B2                  ·   68 ACF8FE00      PUSH OFFSET FerroCad.00FEF8AC                   │pTargetHandle = FerroCad.0FEF8AC -> NULL
00FF21B7                  ·   56               PUSH ESI                                        │hTargetProcess
00FF21B8                  ·   FF15 BC10FF00    CALL DWORD PTR DS:[<&KERNEL32.GetCurrentThread>]  KERNEL32.GetCurrentThread
00FF21BE                  ·   50               PUSH EAX                                        │hSourceHandle
00FF21BF                  ·   56               PUSH ESI                                        │hSourceProcess
00FF21C0                  ·   FF15 5811FF00    CALL DWORD PTR DS:[<&KERNEL32.DuplicateHandle>]  └KERNEL32.DuplicateHandle
00FF21C6                  ·   85C0             TEST EAX,EAX
00FF21C8                  ·   5E               POP ESI
00FF21C9                  ·v  74 3B            JE SHORT 00FF2206
00FF21CB                  ·   A1 ACF8FE00      MOV EAX,DWORD PTR DS:[0FEF8AC]
00FF21D0                  ·   85C0             TEST EAX,EAX
00FF21D2                  ·v  74 32            JE SHORT 00FF2206
00FF21D4                  ·   8D4424 00        LEA EAX,[LOCAL.0]
00FF21D8                  ·   50               PUSH EAX                                        ┌pThreadId => OFFSET LOCAL.0
00FF21D9                  ·   6A 00            PUSH 0                                          │CreationFlags = 0
00FF21DB                  ·   6A 00            PUSH 0                                          │Parameter = NULL
00FF21DD                  ·   68 3032FF00      PUSH FerroCad.00FF3230                          │StartAddress = FerroCad.0FF3230
00FF21E2                  ·   6A 00            PUSH 0                                          │StackSize = 0
00FF21E4                  ·   6A 00            PUSH 0                                          │pSecurity = NULL
00FF21E6                  ·   FF15 6411FF00    CALL DWORD PTR DS:[<&KERNEL32.CreateThread>]     └KERNEL32.CreateThread
00FF21EC                  ·   85C0             TEST EAX,EAX
00FF21EE                  ·v  74 16            JE SHORT 00FF2206
00FF21F0                  ·   68 A031FF00      PUSH FerroCad.00FF31A0
00FF21F5                  ·   C705 B0F8FE00 (  MOV DWORD PTR DS:[0FEF8B0],1
00FF21FF                  ·   68 E8030000      PUSH 3E8
00FF2204                  ·v  EB 1A            JMP SHORT 00FF2220
00FF2206                  >   33C0             XOR EAX,EAX
00FF2208                  ·   68 A030FF00      PUSH FerroCad.00FF30A0
00FF220D                  ·   66:A1 0620FE00   MOV AX,WORD PTR DS:[0FE2006]
00FF2213                  ·   8D0480           LEA EAX,[EAX*4+EAX]
00FF2216                  ·   8D0480           LEA EAX,[EAX*4+EAX]
00FF2219                  ·   8D0C80           LEA ECX,[EAX*4+EAX]
00FF221C                  ·   C1E1 03          SHL ECX,3
00FF221F                  ·   51               PUSH ECX
00FF2220                  >   6A 00            PUSH 0
00FF2222                  ·   6A 00            PUSH 0
00FF2224                  ·   68 D857FE00      PUSH OFFSET FerroCad.00FE57D8                   ┌Procname = "SetTimer"
00FF2229                  ·   68 D057FE00      PUSH OFFSET FerroCad.00FE57D0                   │┌ModuleName = "USER32"
00FF222E                  ·   FF15 6011FF00    CALL DWORD PTR DS:[<&KERNEL32.GetModuleHandleA>]  ││KERNEL32.GetModuleHandleA
00FF2234                  ·   50               PUSH EAX                                        │hModule
00FF2235                  ·   FF15 7011FF00    CALL DWORD PTR DS:[<&KERNEL32.GetProcAddress>]   └KERNEL32.GetProcAddress
00FF223B                  ·   FFD0             CALL EAX
00FF223D                  ·   A3 90F7FE00      MOV DWORD PTR DS:[0FEF790],EAX
00FF2242                  ·   59               POP ECX
00FF2243                  └·  C3               RETN
```

En esta rutina vemos que en FF21E6 se trata de crear un hilo que llama a una función dentro del ejecutable (FF3230). Si vamos ahí vemos que al final hay una llamada a RNBOsproRead:



Esto lo que trata es de seguir haciendo llamadas a la mochila, incluso cuando la parte del Shell ha terminado de ejecutarse, por lo tanto lo que vamos a hacer es que cuando lleguemos a la linea FF21E6 vamos a cambiar el parámetro CreationFlag del valor 0 al valor 4 que significa CREATE_SUSPENDED, para crearlo suspendido, y de esta manera que no se ejecuten esas llamadas a RNBOsproRead





Si seguimos traceando con F8 llegamos a:

Quiere ponernos un timer de manera que la función que hay en FF31A0 se ejecute cada segundo. Si vamos a esa dirección:

```
00FF31A0  .  A1 B4F8FE00    MOV EAX,DWORD PTR DS:[0FEF8B4]
00FF31A5  .  53             PUSH EBX
00FF31A6  .  33DB           XOR EBX,EBX
00FF31A8  .  3BC3           CMP EAX,EBX
00FF31AA  .v 74 74          JE SHORT 00FF3220
00FF31AC  .  391D B8F8FE00  CMP DWORD PTR DS:[0FEF8B8],EBX
00FF31B2  .v 75 6C          JNE SHORT 00FF3220
00FF31B4  .  56             PUSH ESI
00FF31B5  .  C705 B8F8FE00  MOV DWORD PTR DS:[0FEF8B8],1
00FF31BF  .  891D B4F8FE00  MOV DWORD PTR DS:[0FEF8B4],EBX
00FF31C5  .  BE 02000000    MOV ESI,2
00FF31CA  .  E8 C1FCFFFF    CALL 00FF2E90
00FF31CF  .  66:85C0        TEST AX,AX
00FF31D2  .v 74 3F          JE SHORT 00FF3213
00FF31D4  >  46             INC ESI
00FF31D5  .  891D B4F8FE00  MOV DWORD PTR DS:[0FEF8B4],EBX
00FF31DB  .  83FE 01        CMP ESI,1
00FF31DE  .v 7C 29          JL SHORT 00FF3209
00FF31E0  .  C705 84F8FE00  MOV DWORD PTR DS:[0FEF884],1
00FF31EA  .  E8 21020000    CALL 00FF3410
00FF31EF  .  68 10200000    PUSH 2010
00FF31F4  .  E8 67EEFFFF    CALL 00FF2060
00FF31F9  .  50             PUSH EAX
00FF31FA  .  E8 F1080000    CALL 00FF3AF0
00FF31FF  .  83C4 08        ADD ESP,8
00FF3202  .  E8 99020000    CALL 00FF34A0
00FF3207  .  33F6           XOR ESI,ESI
00FF3209  >  E8 82FCFFFF    CALL 00FF2E90
00FF320E  .  66:85C0        TEST AX,AX
00FF3211  .^ 75 C1          JNE SHORT 00FF31D4
00FF3213  >  891D B4F8FE00  MOV DWORD PTR DS:[0FEF8B4],EBX
00FF3219  .  891D B8F8FE00  MOV DWORD PTR DS:[0FEF8B8],EBX
00FF321F  .  5E             POP ESI
00FF3220  >  5B             POP EBX
00FF3221  L. C2 1000        RETN 10
```

Hay una serie de llamadas a otras funciones. Por ejemplo dentro de FF2E90 hay una llamada a RNBOsproQuery y a RNBOsproWrite. Por tanto también debemos tratar de evitar esto. Lo haremos cambiando el parámetro Timeout de la función SetTimer de 3E8 a 0, para que nunca tenga lugar la llamada a esas funciones:

```
0012FFA4  00000000  ....  hWnd = NULL
0012FFA8  00000000  ....  TimerID = 0
0012FFAC  00000000  ....  Timeout = 0
0012FFB0  00FF31A0  á1 .  TimerFunc = FerroCad.0FF31A0
```

-** Llegando al OEP**-

Ahora si pulsamos F9 llegamos a un último CALL EDX esta vez a la función que está en FF3BE0, que básicamente comprueba la integridad del ejecutable, así que podemos continuar hasta el RET de esa función. Una vez alli, vamos poco a poco traceando con F8 hasta que llegamos a FF2DB1, que es un CALL que está casi al final del gran bucle que hemos estado ejecutando hasta el momento:

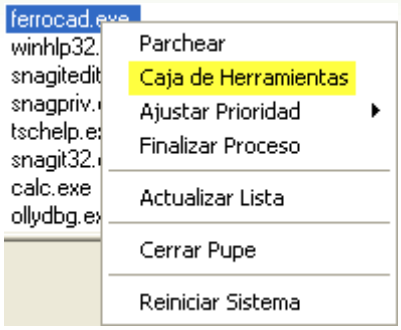
```
00FF2DAA  .  8B5424 04      MOV EDX,DWORD PTR SS:[ARG.1]
00FF2DAE  .  50             PUSH EAX                      Arg3 => [ARG.3]
00FF2DAF  .  51             PUSH ECX                      Arg2 => [ARG.2]
00FF2DB0  .  52             PUSH EDX                      Arg1 => [ARG.1]
00FF2DB1  .  FF15 A04BFE00  CALL DWORD PTR DS:[0FE4BA0]   FerroCad.0040DCA8
00FF2DB7  .  C2 0C00        RETN 0C
```

Vemos que está haciendo CALL a 40DCA8. Entremos con F7 y llegamos a:

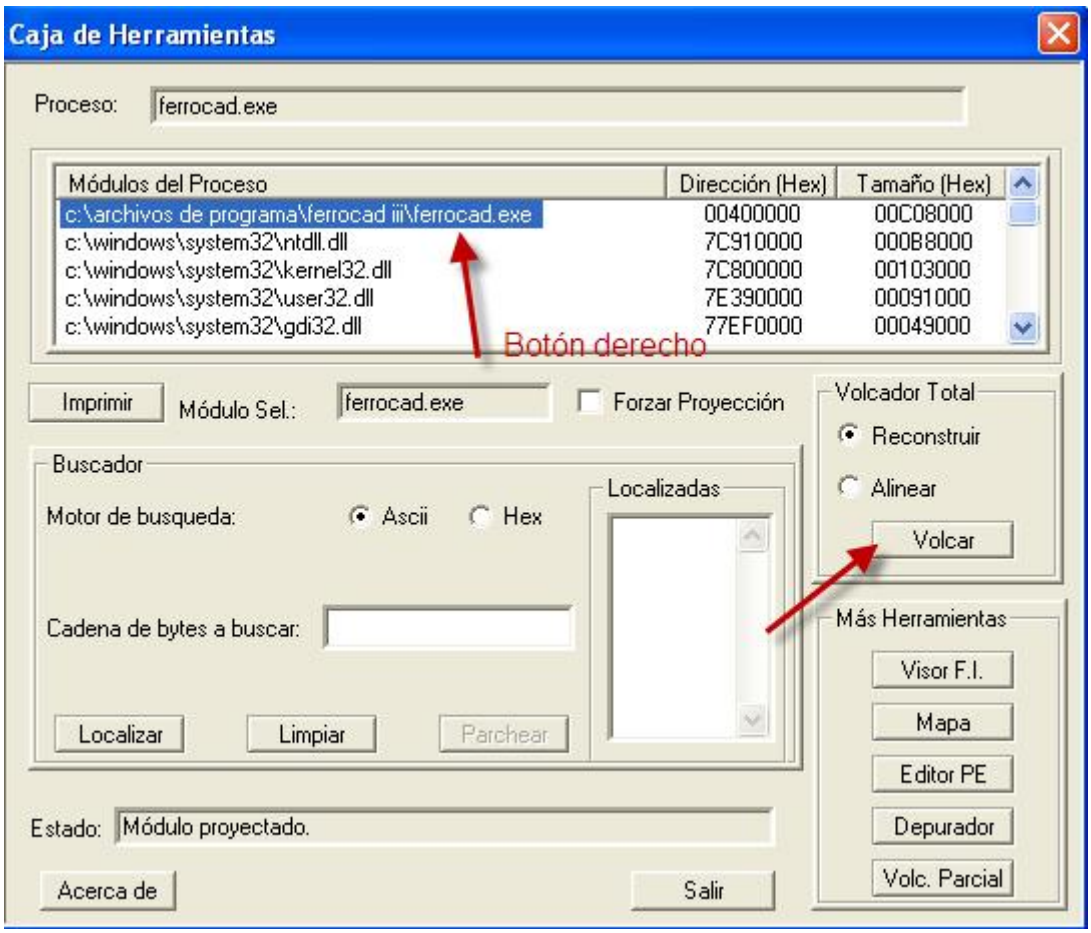
```
0040DC96  - FF25 38114000   JMP DWORD PTR DS:[401138]
0040DC9C  - FF25 9C114000   JMP DWORD PTR DS:[40119C]
0040DCA2  - FF25 84124000   JMP DWORD PTR DS:[401284]
0040DCA8    68 40DD4000     PUSH OFFSET FerroCad.0040DD40
0040DCAD    E8 F0FFFFFF     CALL <JMP.ThunRTMain>         Jump to MSVBVM60.ThunRTMain
```

Que es el EP típico de los programas hechos en Visual Basic. Así que ESE ES NUESTRO OEP!!!.

Para hacer el dumpeado como no tenemos plugins, lo haré con el PUPE, buscamos el proceso, y con el botón derecho abrimos el menú contextual y pulsamos en la Caja de Herramientas:

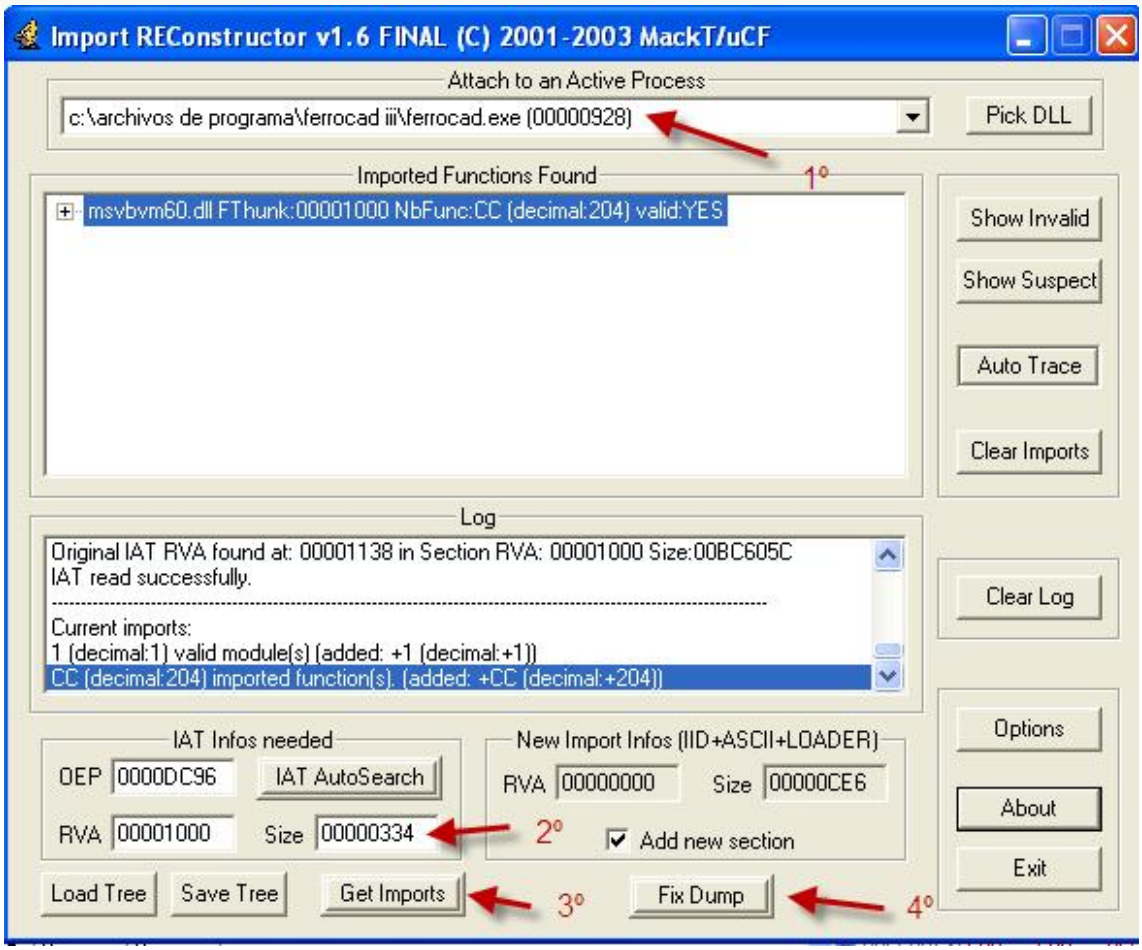


Aparecemos en la siguiente ventana en la que debemos pulsar con el botón derecho en derrocad.exe y luego con el izquierdo sobre el botón Volcar:

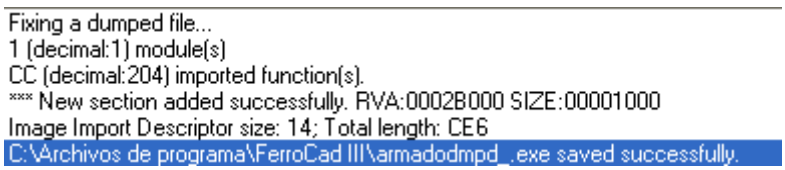


Le damos el nombre que queramos y ya lo tenemos.

Sólo falta arreglar la IAT, lo haremos con el Import Reconstructor, donde elegimos 1º el proceso derrocad.exe, después rellenamos los datos de la IAT y pulsamos en GetImports. Salen todas las funciones resueltas, así que damos a Fix Dump, y elegimos como archivo a fijar el dumpeado que generamos con el Pupe:



Vemos que lo genera satisfactoriamente:



Cogemos el ejecutable que ha generado el Import Reconstructor, lo ejecutamos y … FUNCIONA!!!:

-** Resumiendo **-

Bueno, pues este es el resumen de todo lo que tuve que hacer para saltar esta protección. Visto así no es nada del otro mundo, sin embargo, a mí me ha llevado casi un mes desde que empecé a darle vueltas hasta que por fin conseguí que funcionara. Pero por eso mismo la satisfacción ha sido mayor que cualquier otro programa que he conseguido destripar. Con esto quiero decir que por muy duro que pueda parecer algo, casi siempre el esfuerzo tiene su recompensa, y la mayoría de protecciones tienen un hueco por donde hincar el diente, así que a no desanimarse.

Básicamente en lo que se resume esta protección es en aplicar las funciones Read y Query y dirigir el flujo del programa en función de las respuestas recibidas, así que si veíamos que el flujo no era el deseado, pues cambiábamos la respuesta de manera que el flujo fuera por donde queríamos.

-** Saludos y/o Agradecimientos **-

Este tuto va dedicado a todos los CracksLatinos, y en especial a Ricardo Narvaja, ya que sin su curso de Olly desde cero yo no hubiera sido capaz ni de atreverme a intentar ningún tipo de reto. También quiero nombrar desde aquí a Vortice y El_Cid que me ayudaron mucho con otro programa que ellos saben cuál es **J**.

Un saludo a todos y hasta el próximo