Test6

CONCURSO 1.
Objetivo: Unpacking!
Packer: VMProtect v1.60 – 2.05
Herramientas: OllyDBG v1.10, y PlugIns.
PlugIns Fundamentales:

CommandBar v3.00.108
StrongOD v0.4.5.810
OllyDump v3.00.110
Break On Execution v1.1b

Por: Nox
PE: Test6.exe [VB 6.0]

Enero – 2012

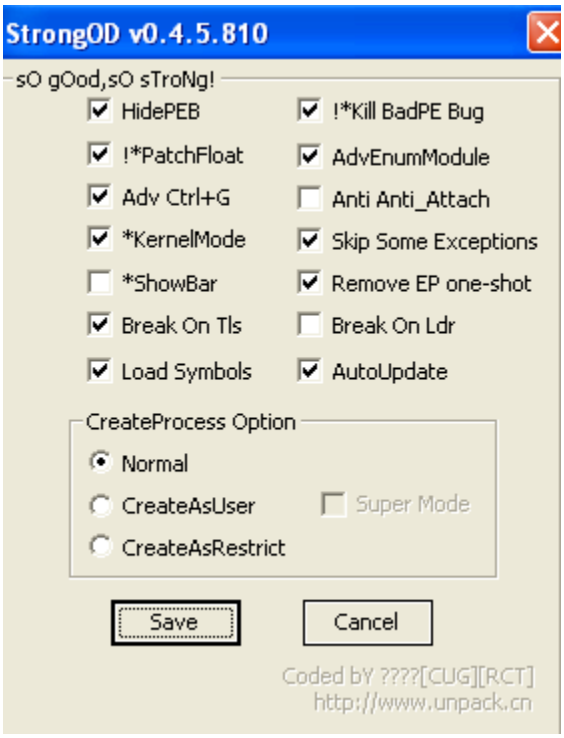
Introducción

Haciendo este VMProtect para mi es un gran paso, pero no puedo decir que estoy satisfecho, es más dude mucho en mandarlo así y escribir este tute, pero bueno u.u, como dicen en la lista siempre es bueno que haya bastante tutes de algo que poco o ninguno.

El crédito por este tute se lo lleva karmany, ya que buscando información sobre VB, encontré un escrito de él y cómo reparaba la iat de un Themida creo que era, pero estoy seguro que no era un VMProtect. A esto se suma al tute de Guan de Dio que hizo sobre este packer, creo que si no hubiera leído esos escritos no lo pudiera haber resuelto. Muchas gracias a los dos! Un humilde Unpacking…

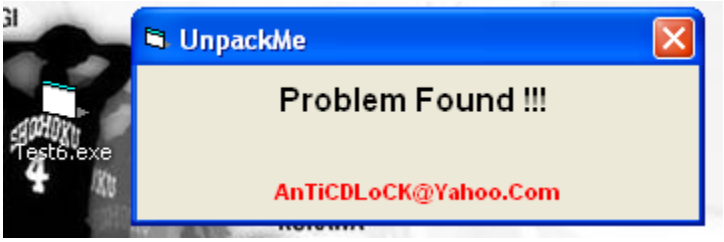
Configuración del PlugIn

Para este packer es necesario usar el StronOD, esta configuración es la que uso por defecto, así qué, si me funciona, no cambio nada.



Analizando

Ejecutamos.



Analizamos con el PiD.

```
-=[ ProtectionID v0.6.4.1 JULY]=-
(c) 2003-2011 CDKiLLER & TippeX
Build 07/22/11-02:48:07
Ready...
Scanning -> C:\Documents and Settings\Nox\Escritorio\Test6.exe
File Type : 32-Bit Exe (Subsystem : Win GUI / 2), Size : 1089536 (010A000h) Byte(s)
[File Heuristics] -> Flag : 00000000000000001100000100000011 (0x0000C103)
[!] VM Protect v1.60 - v2.05 detected !
- Scan Took : 0.172 Second(s) [0000000ACh tick(s)]
```

Nos sacó la versión.

Unpackeando

Aquí es donde comienza el laburo. Abrimos el proggie dentro del OllyDBG…

```
Address   Hex dump          Disassembly
005F9A05  E8 340A0000       CALL Test6.005FA43E
005F9A0A  3C F6             CMP AL, 0F6
005F9A0C  92                XCHG EAX, EDX
005F9A0D  06                PUSH ES
005F9A0E  F6                ???
005F9A0F  8C2F              MOV WORD PTR DS:[EDI], GS
005F9A11  6D                INS DWORD PTR ES:[EDI], DX
005F9A12  1C F3             SBB AL, 0F3
005F9A14  ^ E2 89           LOOPD SHORT Test6.005F999F
```

Este packer usa la tabla TLS para empezar a desempacarse, entonces para la resolución de este packer, se debe tener cómo obligación los PlugIns que he mencionado.

El VMProtect detecta los BPs, BPMs, HBPs, para poder usarlos debemos saber en que momento deben ser puestos, siendo así, el packer no lo detectará.

Ahí parados en el TLS CallBack…

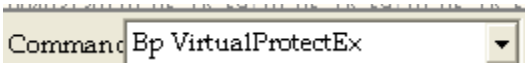
Como todo, debemos llegar al OEP y lo primero que haría, sería poner un BPM on Access (Execution) en la sección .CODE, pero como sabemos este packer detecta los BPM así que debemos saber dónde ponerlos.

Los BPMs no son más que cambios en los permisos de una terminada región, es decir quitar o agregar permisos de escritura, lectura o acceso + el flag PAGE_GUARD, cuando el progie accede a la región de memoria que queremos controlar, manda una notificación al depurador, generando que este pare, todo esto controlado nativamente por el Olly.

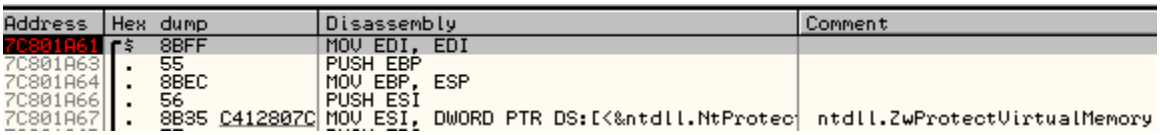
Para que el packer gestione los cambios que hemos hecho con estas banderas/flag, necesita obtener dicha información de las regiones de memoria, para esto puede usar la API VirtualQuery o también la API VirtualProtect u otra que desconozco o se me está pasando, pero estoy con un sueño ZzZzZzZz…

También sabemos que detecta los BPs, así que debemos ser cuidados al ponerlos, aparte de que este packer también emula las APIS, así que tenemos todo un espécimen.

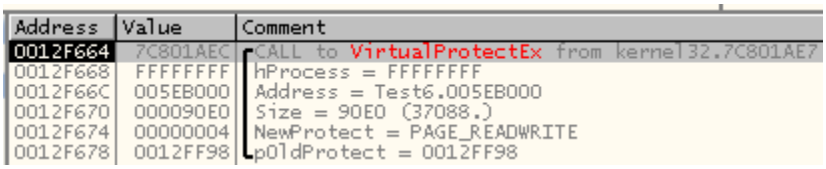
Nos evitamos ese problema si ponemos un BP en VirtualProtectEx.



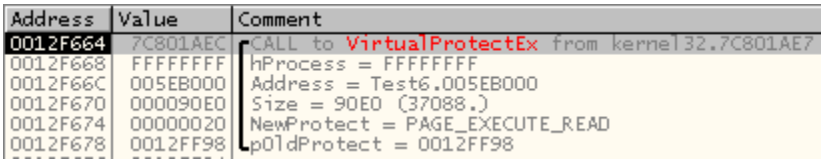
Damos F9 RUN.



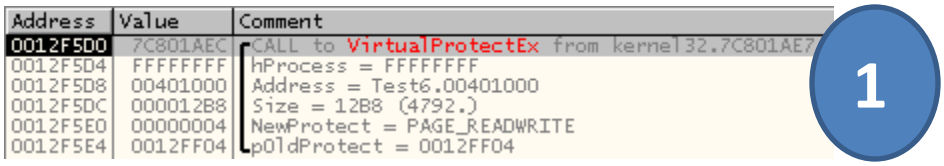
Para por primera vez, si vemos el stack →



Nada interesante… Seguimos con F9 RUN.



Nada tampoco seguimos con F9 RUN, hasta ver lo siguiente →



Si hubiéramos puesto un BPM en la sección .CODE que comienza en la dirección 00401000h, al pasar esta API el packer detectaría el cambio que hicimos y nos sacaría a patadas.

Entonces lo que debemos hacer es poner el BPM una vez que pase la comprobación del packer a la sección .CODE.

F9 RUN, otra vez.

```
Address   Value     Comment
0012F5D0  7C801AEC  CALL to VirtualProtectEx from kernel32.7C801AE7
0012F5D4  FFFFFFFF  hProcess = FFFFFFFF
0012F5D8  00405000  Address = Test6.00405000
0012F5DC  0000F60E  Size = F60E (62990.)
0012F5E0  00000004  NewProtect = PAGE_READWRITE
0012F5E4  0012FF04  pOldProtect = 0012FF04
```

Seguimos en la sección .CODE.

F9 RUN, otra vez.

```
Address   Value     Comment
0012F5D0  7C801AEC  CALL to VirtualProtectEx from kernel32.7C801AE7
0012F5D4  FFFFFFFF  hProcess = FFFFFFFF
0012F5D8  00401000  Address = Test6.00401000
0012F5DC  000012B8  Size = 12B8 (4792.)
0012F5E0  00000020  NewProtect = PAGE_EXECUTE_READ
0012F5E4  0012FF04  pOldProtect = 0012FF04
```

F9 RUN de nuevo.

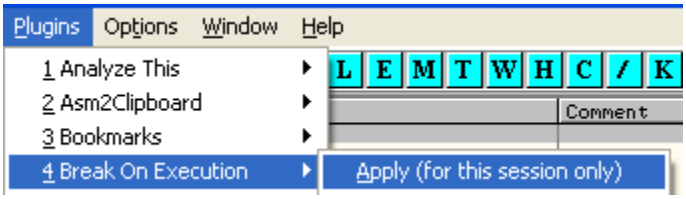
```
Address   Value     Comment
0012F5D0  7C801AEC  CALL to VirtualProtectEx from kernel32.7C801AE7
0012F5D4  FFFFFFFF  hProcess = FFFFFFFF
0012F5D8  00405000  Address = Test6.00405000
0012F5DC  0000F60E  Size = F60E (62990.)
0012F5E0  00000020  NewProtect = PAGE_EXECUTE_READ
0012F5E4  0012FF04  pOldProtect = 0012FF04
```

El packer sigue gestionando la protección a la determinada región de memoria.

F9 RUN.

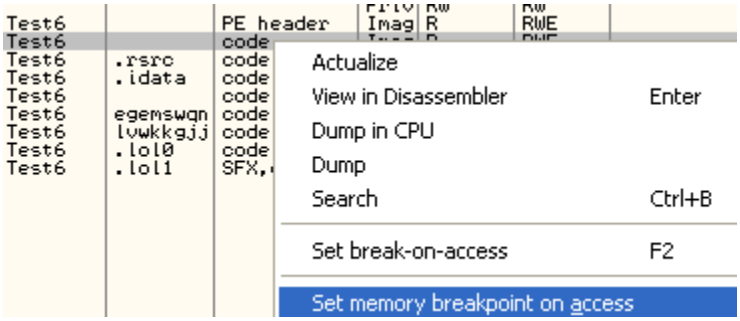
```
Address   Value     Comment
0012FBB4  7C801AEC  CALL to VirtualProtectEx from kernel32.7C801AE7
0012FBB8  FFFFFFFF  hProcess = FFFFFFFF
0012FBBC  02C70000  Address = 02C70000
0012FBC0  00006000  Size = 6000 (24576.)
0012FBC4  00000020  NewProtect = PAGE_EXECUTE_READ
0012FBC8  0012FBF4  pOldProtect = 0012FBF4
```

De aquí hacia delante ya no vuelve a tocar la sección .CODE, entonces para intentar llegar al OEP lo hacemos de la siguiente manera:

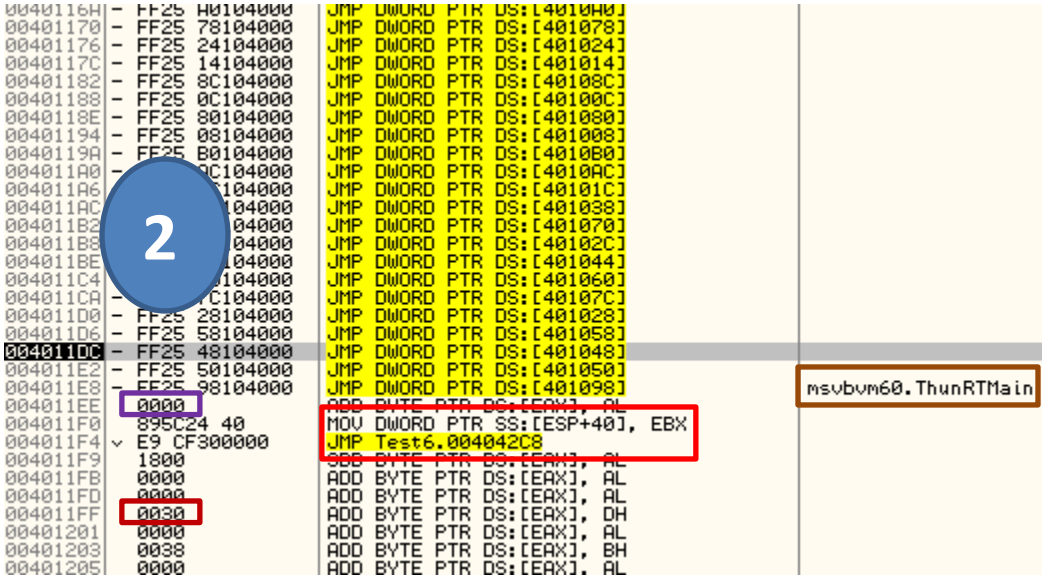


De esta forma, al poner un BPM *on Access*, parará sólo por o*n Execution*.

Y ponemos el BPM en la sección .CODE.



Quitamos el BP del VirtualProtectEx, y damos F9 RUN.



Uppss...!!!

Algo pasó aquí! Listemos lo que observamos en estos momentos.

Paramos en un JMP indirecto de la IAT, y más abajo vemos la función de la VM de VB que nunca cambian la entrada o la emulan (al menos nunca lo he visto yo).

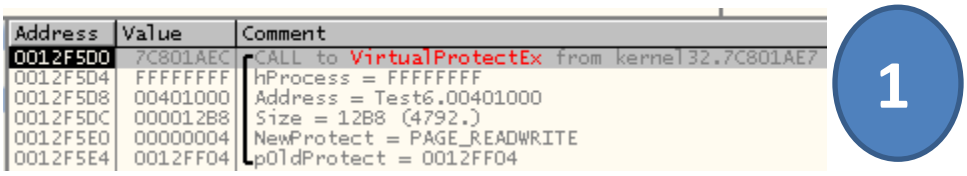
- EIP en un JMP indirecto que no es ThunRTMain.
- ThunRTMain, primera función a ejecutarse en un VB (normalmente).
- 0x0000, En todo OEP "normal" de VB, encontramos estos bytes después del JMP ThunRTMain, y antes del OEP, es decir en el medio.
- Normalmente antes del OEP se encuentra los bytes 0x0000, y los JMPs indirectos de la IAT.
- Stolen Bytes!, aquí debería ir la siguiente instrucción:
  PUSH ADDR
  CALL ThunRTMain
  Siendo ADDR un puntero a la cadena "*VB5!*" (Sin comillas).
- Luego de la instrucción CALL (del OEP) encontramos en un determinada longitud de bytes más abajo, el byte 0x30.

De todo esto podemos deducir: Reconstruir el OEP!!!!

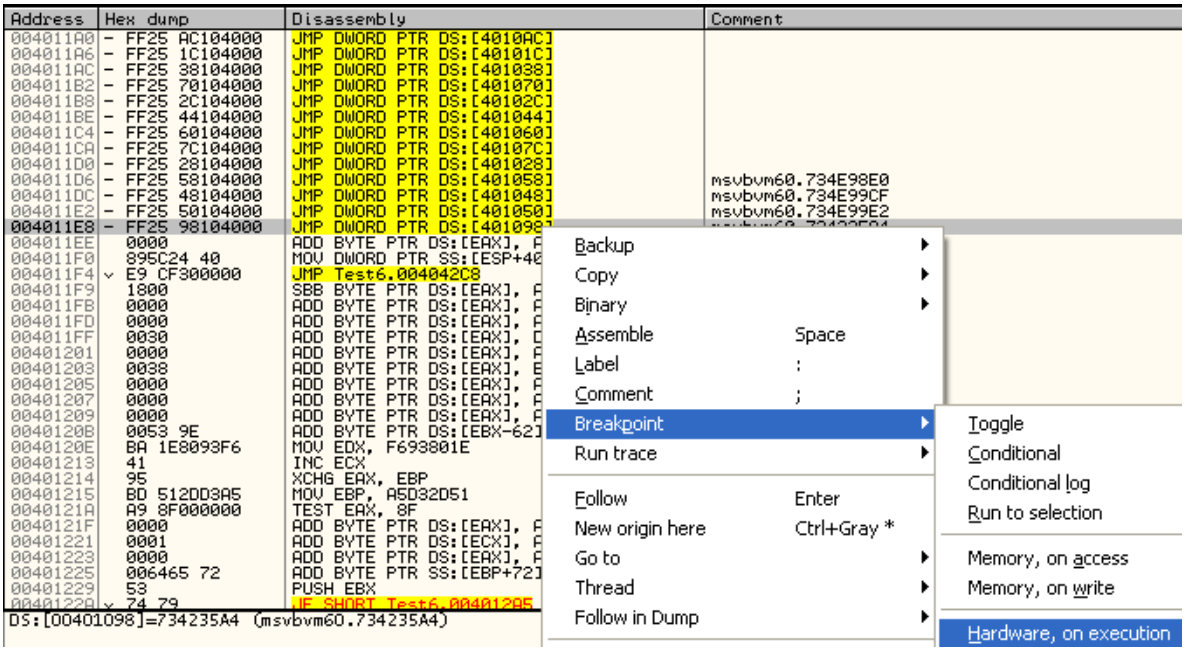
Para poder reconstruir debemos estar parados en el JMP indirecto a la función ThunRTMain.

Reiniciamos.

Hacemos todos los pasos hasta llegar al Paso1 es decir poner un Bp En VirtualProtectEx y llegar hasta aquí→



Nos vamos a la dirección JMP indirecto a la función ThunRTMain → 004011E8h, y ponemos un HE.



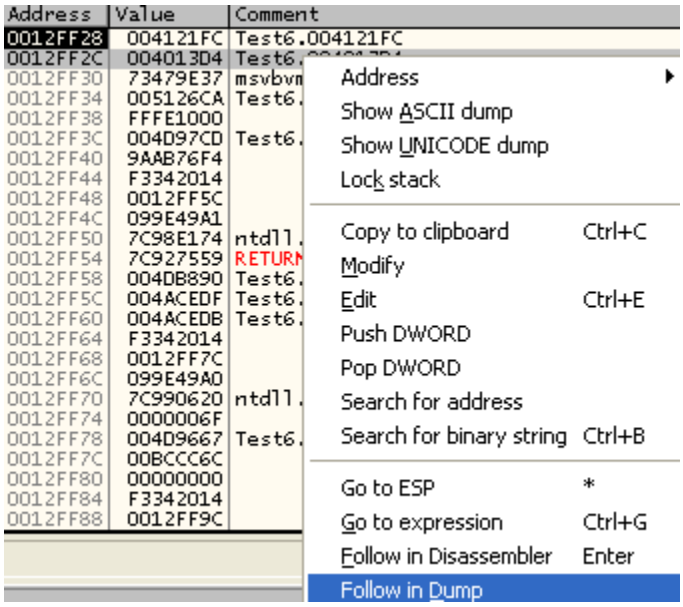
Hay 2 razones por lo que hacemos esto aquí.

- Primero al llegar al Punto1, ya fue desencriptado los bytes de la sección .CODE y la vemos como en la imagen anterior, si viéramos la sección antes del Punto1 encontraríamos basura.
- Cuando llegamos en el Punto1, ya pasamos la comprobación de los HBPs, y podemos poner uno sin ningún temor a que nos detecte.

F9 y RUN. Luego de terminar el último break de VirtualProtectEx haciendo referencia la sección .CODE (00405000h), rompe por el HE.

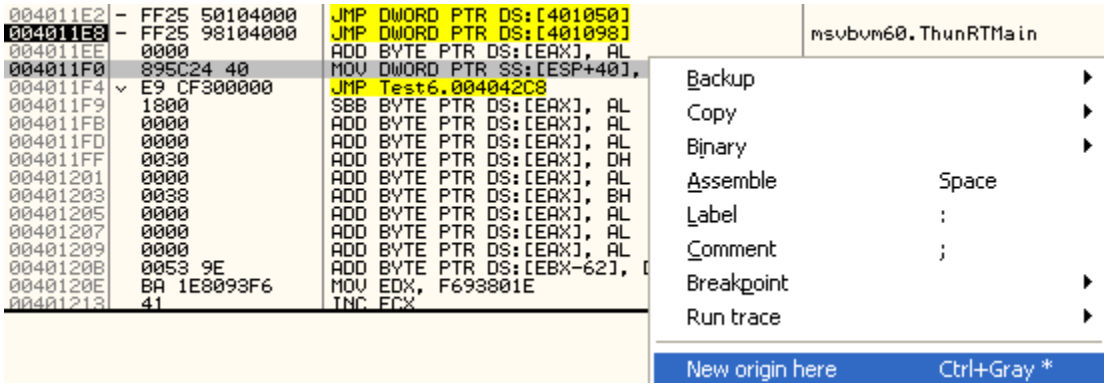
```
004011DC  - FF25 48104000   JMP DWORD PTR DS:[401048]
004011E2  - FF25 50104000   JMP DWORD PTR DS:[401050]
004011E8  - FF25 98104000   JMP DWORD PTR DS:[401098]      msvbvm60.ThunRTMain
004011EE    0000            ADD BYTE PTR DS:[EAX], AL
004011F0    895C24 40       MOV DWORD PTR SS:[ESP+40], EBX
004011F4  v E9 CF300000     JMP Test6.004042C8
004011F9    1800            SBB BYTE PTR DS:[EAX], AL
004011FB    0000            ADD BYTE PTR DS:[EAX], AL
004011FD    0000            ADD BYTE PTR DS:[EAX], AL
004011FF    0030            ADD BYTE PTR DS:[EAX], DH
```

Si miramos el Stack

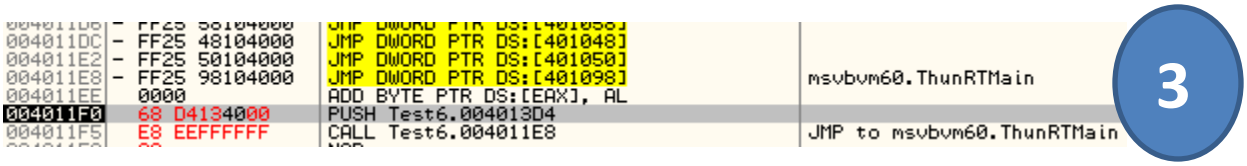


Volcamos la dirección 004013D4h.

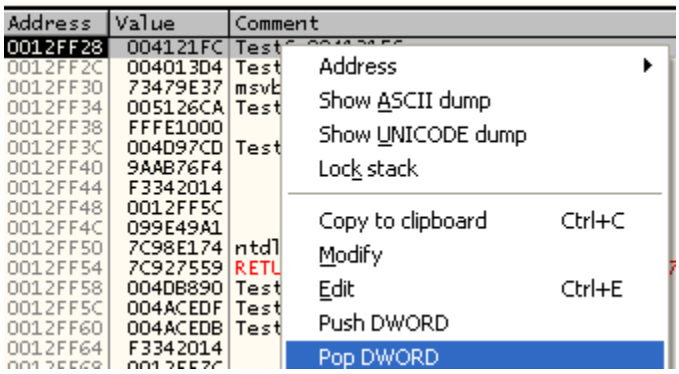
Comenzamos a reparar el OEP, haciendo lo siguiente.



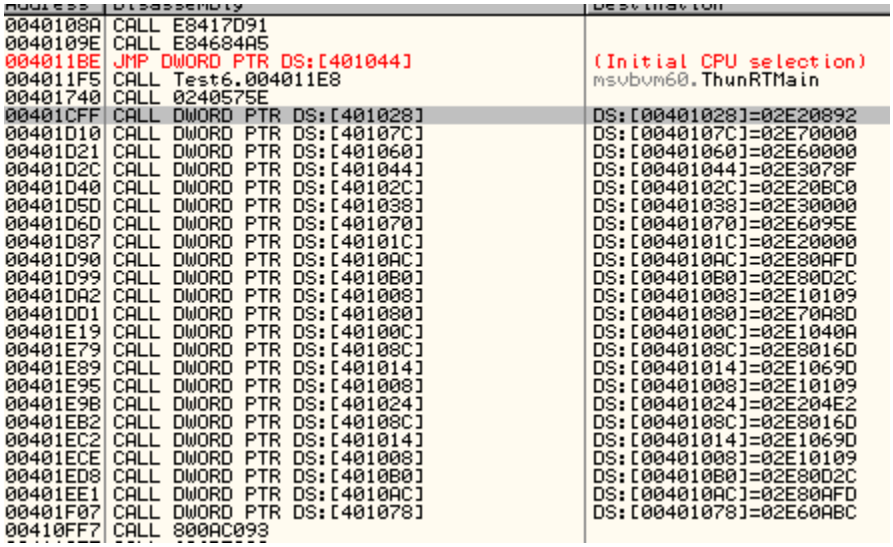
Luego parchamos por estas instrucciones →



Luego hago un POP DWORD dos veces en el stack para que quede bonito ;)

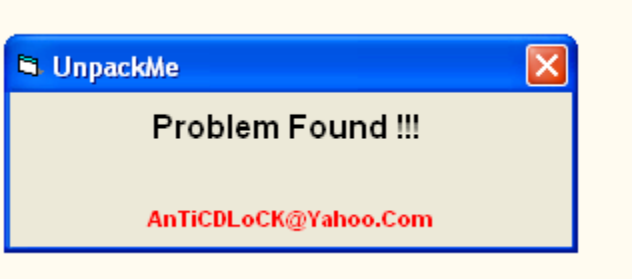


Quiten el BP a VirtualProtectEx, y para probar podemos hacer F9 RUN, y ver cómo corre pero NO lo haremos en estos momentos, primero le echaremos una miradita a los JMPs Indirectos.



Ufa!, si entramos a algún call vemos que esta aplicación usa CALLs indirectos de los que ocupan 6 bytes XD.

Bien con esta Info podemos darle F9 RUN, y vemos cómo hemos reconstruido de manera correcta el OEP ☺.



La debilidad que mencionaba antes es que al poner nosotros su VM es decir la DLL MSVBVM60.DLL en la misma carpeta que el proggie, le dará prioridad a esta y será cargada por el ejecutable en vez de la del sistema, lo podemos ver en la subventana Executables.

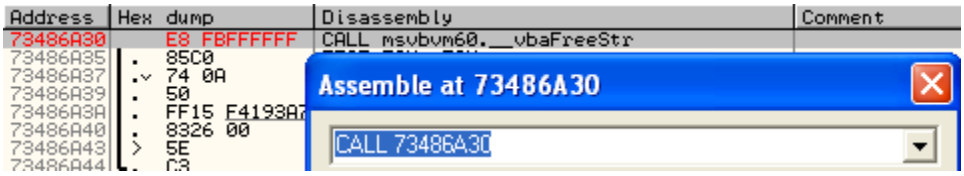
Para eso pongamos el proggie dentro de algún directorio, y dentro de ella la DLL MSVBVM60.DLL.

Cómo el packer emula las APIS, lo que haríamos es modificar el *head of the functions* de la DLL, haciendo un CALL hacia su misma dirección.

Hagamos una Prueba.

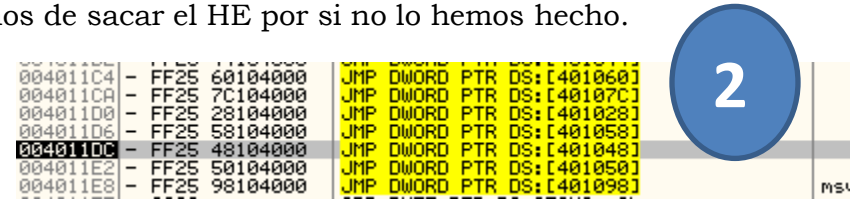
Una de las funciones muy muy usadas es la __vbaFreeStr.

Cargamos la DLL y hacemos la modificación a esa función.

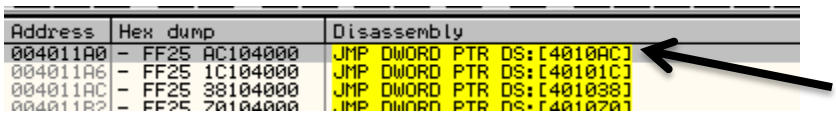


Y ahí está, cambiamos, el mismo Olly nos da la referencia a la dirección que hemos hecho el CALL y el nombre de la función.

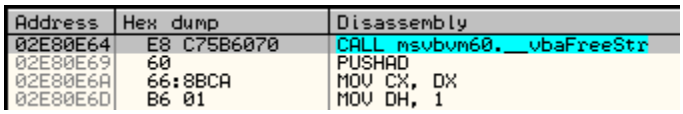
Guardamos los cambios y cargamos el proggie, llegando hasta el Punto2, no olvidándonos de sacar el HE por si no lo hemos hecho.



Ahora buscamos, en cual entrada está esa API jeje :P.



Bueno yo ahorro el trabajo, un FOLLOW



Aja! Esto nos facilita todo, ahí podríamos irnos al ImportRec (previamente reparada el OEP) y usar la opción TraceLevel1 y obtendríamos que entrada es la correcta.

Ahora cómo no somos adivinos, no sabemos todas las APIs que son usadas en este proggie.

Para esto! Me cree un Script, que modifica el Head of Functions de la DLL según el rango que le ponga. ☺

Como la explicación de este Script escapa el objetivo de tute, ya que, el COMO HACERLO ya lo mencioné. Sólo haré unos pantallazos de lo que deben saber.

```
32 ADD AofName, bMSDLL
33 ADD AofFun, bMSDLL // Como es RVA.
34 SUB AofName, 4  // Rango FINAL!
35 MOV  Val, OFF, 1
36
37 ADD AofFun, 50 //Rango de Inicio
```

Ahí tenemos el Rango de inicio y el final, no recomiendo tocar el rango final para nada, pero si el de inicio le diremos, que 50 bytes después de la primera función comience a remplazar el Head.

50h / 4h = 14h

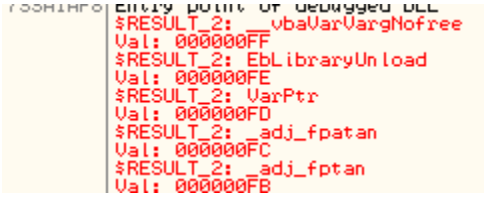
Es decir las primeras 14 funciones no serán tocadas, todas las demás posteriores sí.

```
69 0xFFXX:
70 //MOV eip, Addy1
71 GN Addy1
72 LOG $RESULT_2
73 LOG Val
74 MOV [Addy1], Val, 1
75
76 DEC Val
77
78 CMP AofFun, AofName
79 JZ Final
80 ADD AofFun, 4
81 JMP Bucle
```

No todas las funciones tienen cómo longitud de la misma 5 bytes, tamaño requerido para el CALL que rescribiremos, entonces lo que haremos es poner el valor de 0xFF, e irá disminuyendo según encuentre más funciones con menos de 5bytes. Para saber que funciones son, las logeo así que, nos vamos a esa ventana y ahí encontraremos esa información.

Sin más floro, abrimos la DLL MSVBVM60.DLL (que está en nuestro directorio), y mandamos el script!

Aceptamos el MSGBOX que aparece al terminar el script, y nos vamos a la ventana LOG.

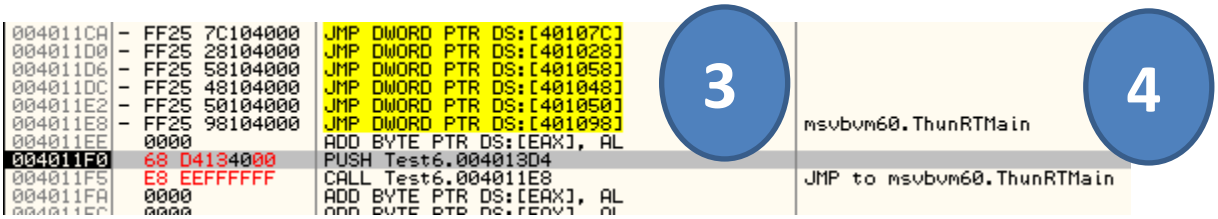


Y ahí están las funciones que se han escrito los valores indicados en la imagen.

Lo guardamos en un fichero de texto para no perderlo.

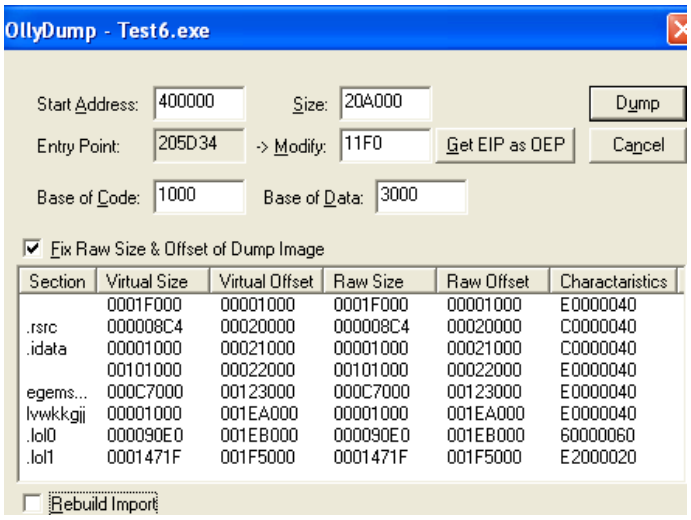
Ahora si miramos el directorio apareció la DLL que hemos modificado, la cambiamos al nombre original "MSVBVM60.DLL", y cargamos nuestro proggie.

Llegamos al Paso3 y hacemos el Paso4 por seguridad.

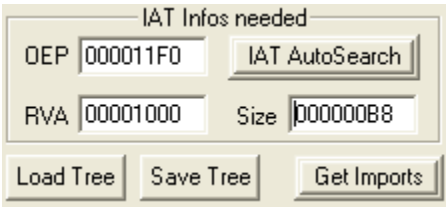


Ahí estamos si comenzamos a revisar los JMPs Indirectos, nos daremos cuenta que las primera línea está el CALL a la función que sería la correcta para cada entrada.

Para proseguir, debemos hacer el DUMP.

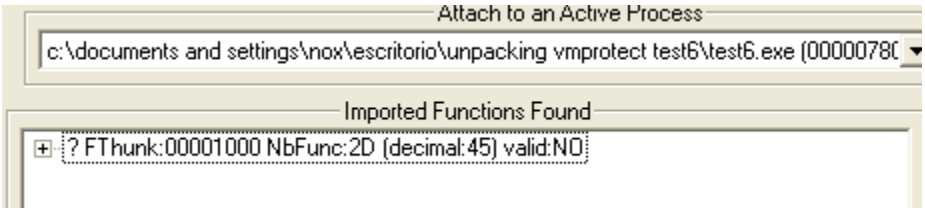


Luego abrimos el ImportRec, ponemos el OEP, 0x11F0 y damos IAT Auto Search.



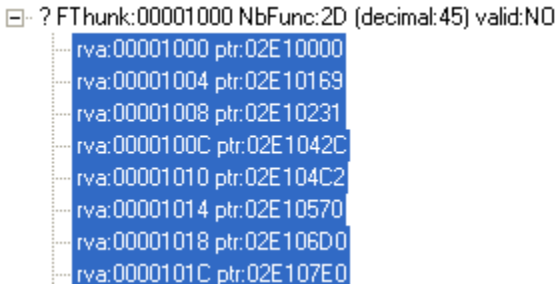
Ahí tenemos los datos correctos (aunque el Size es 0xB4, pero bue… los otros 4 bytes son puros ceros :)

Luego clic a "*Get Imports*".

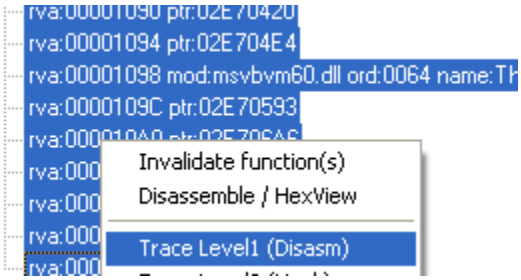


Bueno… así es la vida

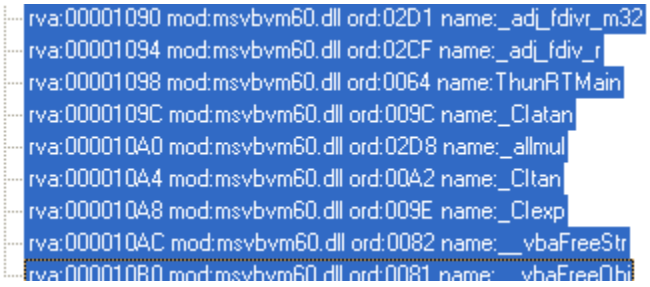
Desplegamos, y sombreamos todos desde el inicio.



Hasta el final.

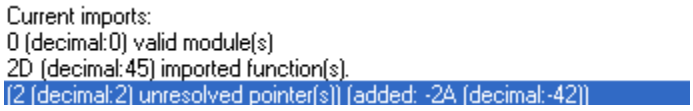


Y Luego miramos como quedó →



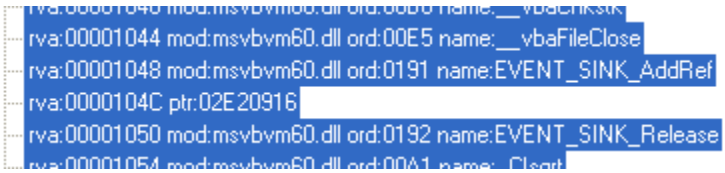
Jeje, muchos mas ponito no? ;)

Miramos el Log.

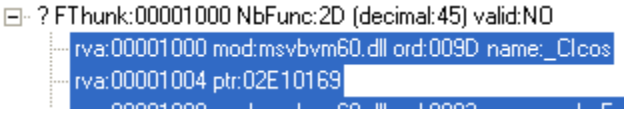


Uuu…. Dos entradas sin resolver.

Subiendo…

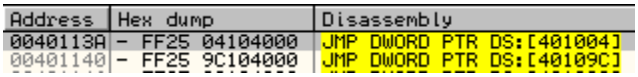


Hay 1 una entrada ahí, seguimos para arriba…



2 entradas por resolver.

La primera →RVA = 1004 y la VA 401004



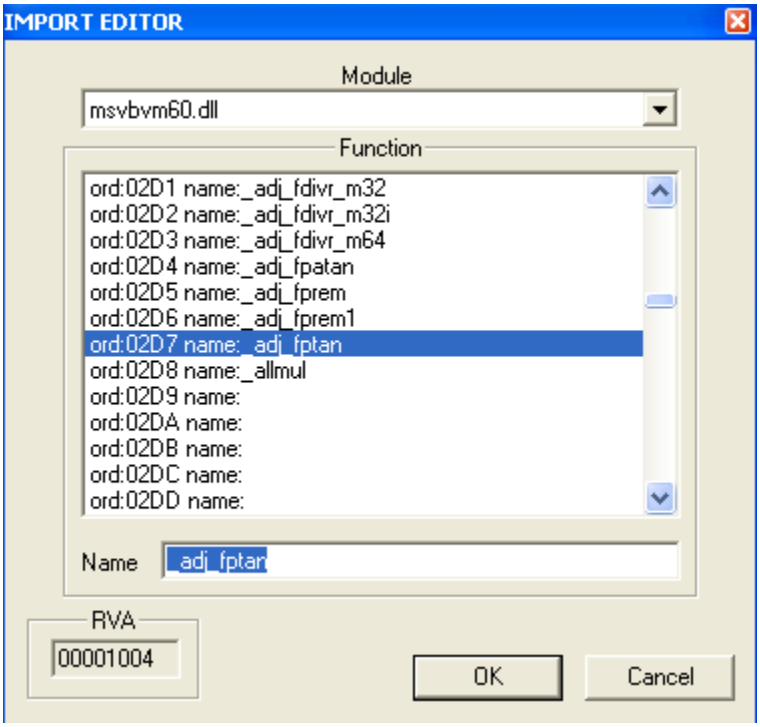
FOLLOW.

Se acuerdan? El byte 0xFB, si lo miro en el log que guardé… por que lo guardaron no?

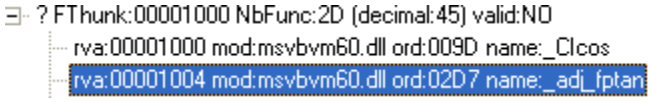
$RESULT_2: _adj_fptan

Val: 000000FB

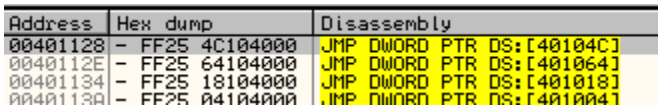
Y lo corregimos todo en el ImportRec, hacemos doble clic a la entrada mala.



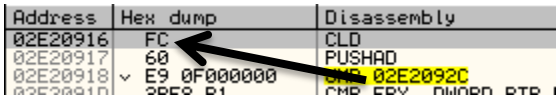
Y clic al botón "*OK*".



Listo! Queda la última → RVA = 0x104C VA= 40104C



FOLLOW.

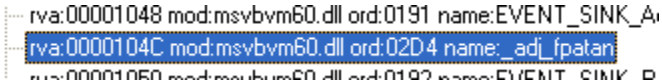


Y ese valor es de la API…

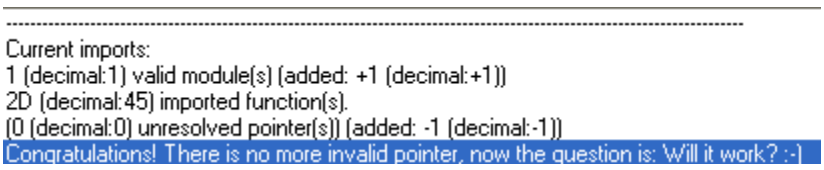
$RESULT_2: _adj_fpatan

Val: 000000FC

Hacemos el mismo procedimiento al anterior, y lo agregamos en el ImportRec.



Listo entrada buena!



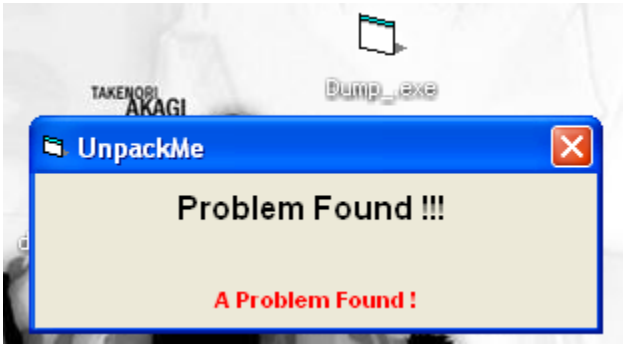
Ahí el log nos dice que todo está de poca ☺

Yo voy a adjuntar el Log del ImportRec, por si las caiguas y ya ustedes sólo le queda modificar el path.

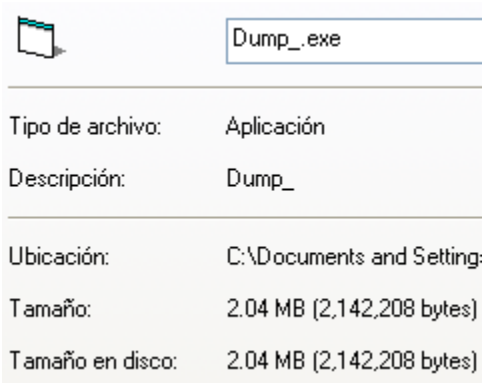
Fixiando

Fixiamos nuestro DUMP, lo saco fuera del directorio para que no tome su DLL modificada.

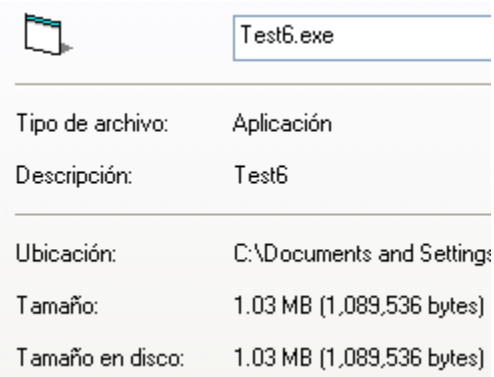
Doble Clic y… tan tan, ni que cosas ahí está el screenshot →



Unpack resuelto, miremos el peso del Unpacked.

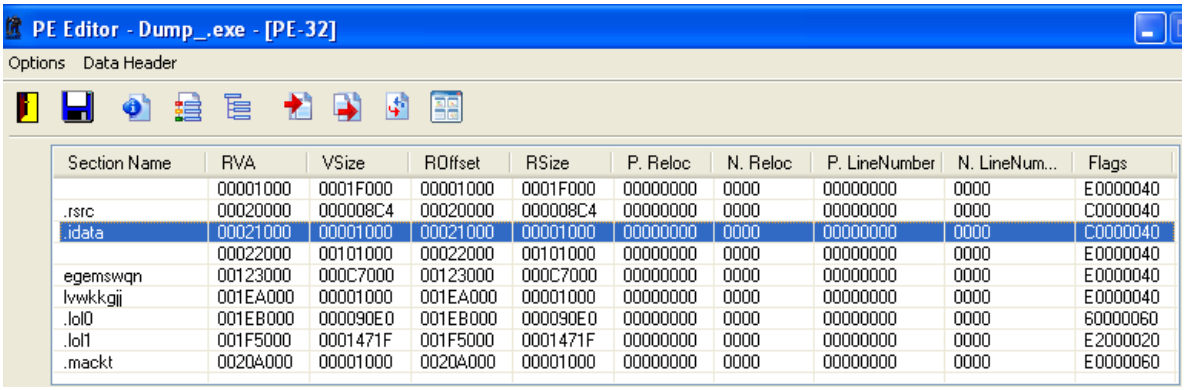


Vaya! Muy pesado, ahora miremos el Packed.



Cómo que muy pesado no?, pues agradecimientos a mi amigo Eddy, por el tip de sacarles las secciones que no usa.

Para eso abrimos el SirPE.

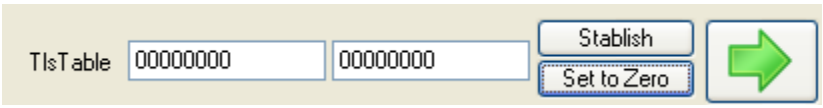


Section Name	RVA	VSize	ROffset	RSize	P. Reloc	N. Reloc	P. LineNumber	N. LineNum...	Flags
	00001000	0001F000	00001000	0001F000	00000000	0000	00000000	0000	E0000040
.rsrc	00020000	000008C4	00020000	000008C4	00000000	0000	00000000	0000	C0000040
.idata	00021000	00001000	00021000	00001000	00000000	0000	00000000	0000	C0000040
	00022000	00101000	00022000	00101000	00000000	0000	00000000	0000	E0000040
egemswqn	00123000	000C7000	00123000	000C7000	00000000	0000	00000000	0000	E0000040
lvwkkgjj	001EA000	00001000	001EA000	00001000	00000000	0000	00000000	0000	E0000040
.lol0	001EB000	000090E0	001EB000	000090E0	00000000	0000	00000000	0000	60000060
.lol1	001F5000	0001471F	001F5000	0001471F	00000000	0000	00000000	0000	E2000020
.mackt	0020A000	00001000	0020A000	00001000	00000000	0000	00000000	0000	E0000060

Tenemos los siguientes, y debemos borrar las secciones dando clic al botón "*Remove Section*" y luego al botón "*Align All to Virtual*", para que quede así →

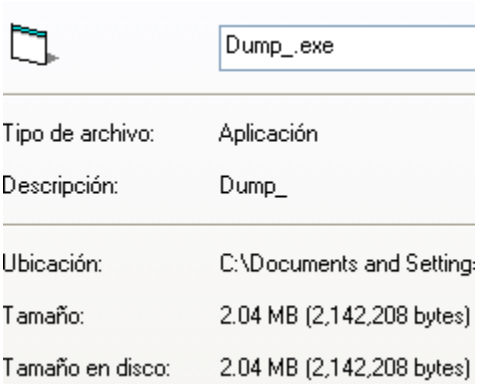
Section Name	RVA	VSize	ROffset	RSize	P. Reloc	N. Reloc	P. LineNumber	N. LineNum...	Flags
.Nox	00001000	0001F000	00001000	0001F000	00000000	0000	00000000	0000	E0000040
.rsrc	00020000	00001000	00020000	00001000	00000000	0000	00000000	0000	C0000040
.mackt	0020A000	00001000	0020A000	00001000	00000000	0000	00000000	0000	E0000060

Recordemos que este packer usa la TLS Table para comenzar a desempacar.

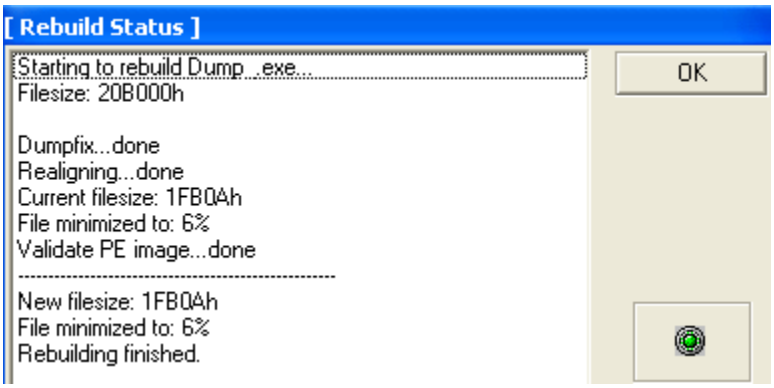


Clic a la Tls Table –el ítem- y luego nos dirigimos a la parte inferior del SirPE, cliqueamos el botón "Set to Zero", y luego al botón "Stablish", para finalmente guardar los cambios.

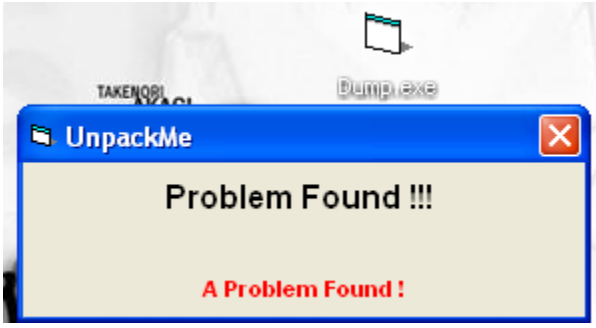
Hacemos clic derecho al proggie para ver si en algo a cambiado el peso.



No nada, y claro falta hacerle un ReBuild, abrimos el LordPE, elegimos el proggie unpacked, y le damos con todo jeje.

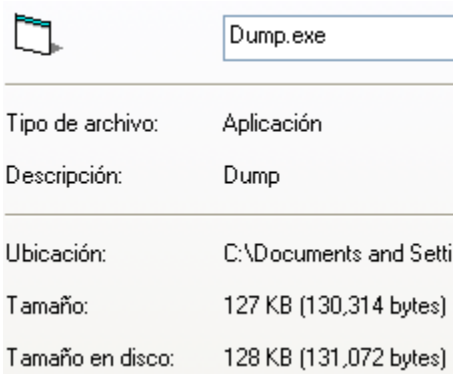


Ejecutamos!



Sin ningún problema.

Clic derecho para mirar el peso del Unpacked!



Je! disminuyó inmensamente.

Adjunto en la descarga:

Modifica - Head of the Functions.txt (Script que modifica a la MSVBVM60.DLL).

Log - ImportRec.txt (Log de todas las APIS de este proggie sólo basta con cambiar el path y darle al botón Load Tree, y se evitan el laburo en el IR).

FunVal.txt (El Log que guardamos de la DLL MSVBVM60.DLL)

MSVBVM60_MODIFICADA.DLL (Bueno ya saben x).

Test6.exe (Proggie enpaquetado)

Unpacked + Fixed.exe (reparado, sin basura, y menos peso ☺).

Y este Tute!.

Hay muchas cosas más que debería agregar y otras que estoy pasando por alto de seguro, pero bueno, creo que con lo explicado se cumple el objetivo, espero que le haya gustado el ortodoxo y humilde Unpacking!

Gracias por la lectura!

28 de Enero del 2012.
Nox.