UnpackMe

CONCURSO 2.
Objetivo: Unpacking!
Packer: Themida v2.1.8
Herramientas: (OllyDBG v1.10, y PlugIns), ImportRec.
PlugIns Fundamentales:
CommandBar v3.00.108
StrongOD v0.4.5.810
OllyDump v3.00.110

Por: Nox **PE**: UnpackMe.exe [Delphi]

Febrero – 2012

Introducción

Dos semanas, si no es más, de estar analizando y perdiendo el tiempo con este packer (entiéndase perder el tiempo como tracear como loco, hacer pruebas de ehm… como lo llamaría brute forcé?, jeje). Cuando empecé enserio a analizar el packer, fue tanto lo que quise lograr, hasta que el propio paker deje la iat correcta, antes ya había encontrado un método de hacerlo, pero quería que el packer caiga en este punto, cosa que no logré, peor aún, caí enfermo y eso disminuyó los días que me quedaban para investigar y escribir el tute.

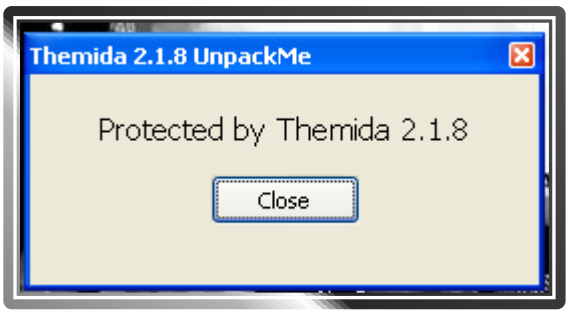
Ya que he visto bastante cosas, no especificaré todo los movimientos del OllyDBG que haga, es más, habrá lugares que sólo señalaré y diré el como llegar, más no mostraré la transición de llegada.

*Una **advertencia!** antes de que se me escape, que sin algún elite está leyendo esto, ya sin más puede dejarlo, yo soy un novato a comparación de muchos en la lista, poder unpackear Themida para mi es un gran logro y motivación, mis unpacks son contados (lo puedes hacer con los dedos de la mano), así que elite, esto no es un video de cómo mágicamente se unpackea themida, tampoco mostraré un script sin más que haga lo mismo, este es un **escrito**, de cómo un novato lo logró.*

Analizando

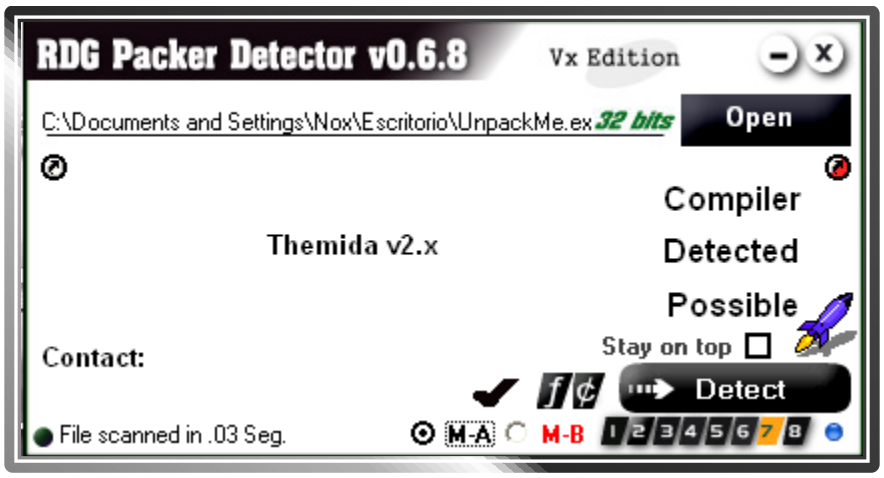
Ejecutamos.

Tarda algo de 5 segundo a más, pero esto es muy relativo.

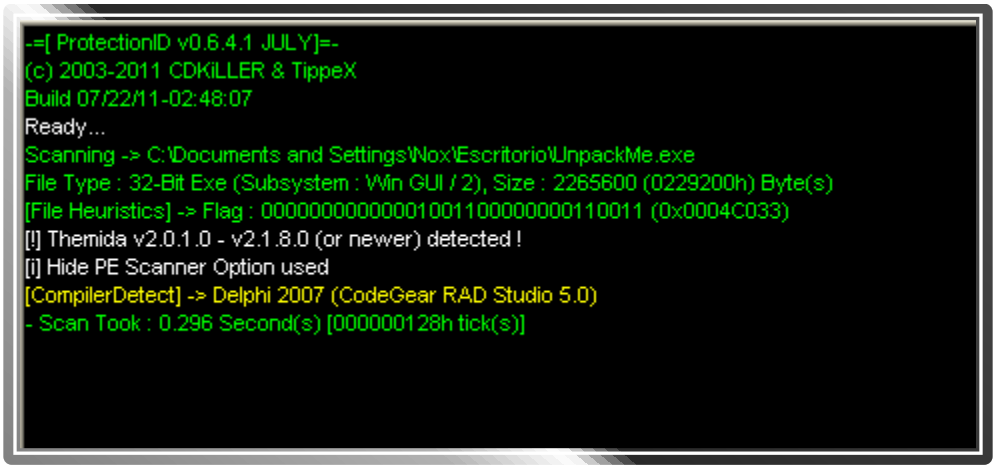


Bueno ahí nos dice que packer es, je, y la versión!

Lo segundo que hice fue pasarle el RDG.



Si marco la opción M-B dice que no tiene nada de nada. Luego le pasé el PID.



Y vaya sorpresa! Me saca la versión, y encima me dice que lenguaje es el proggie, es más, el compilador usado, que buena tool, o el packer como lo dijo el maestro Ricardo es fácil?.

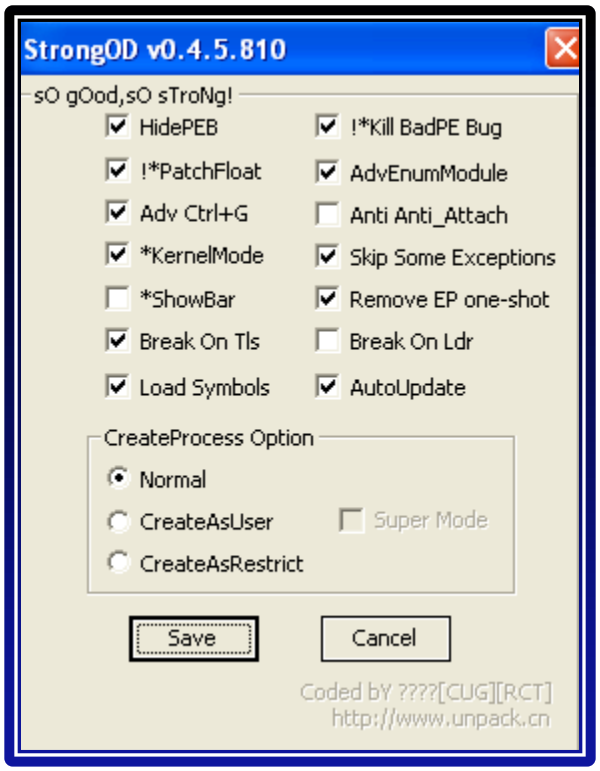
Themida v2.0.1.0 – v2.1.8.0 – Delphi 2007 (CodeGear RAD Studio 5.0), que buena información nos dio ;).

Esto se me ocurrió ahorita al escribir estas líneas, a las 11:40pm, abrir el RDG y usar su OEP detector, si se me hubiera ocurrido al principio, no hubiera sufrido con el OEP, (si, lo sé soy muy novato no?).

Ahí está el posible OEP 45770C.

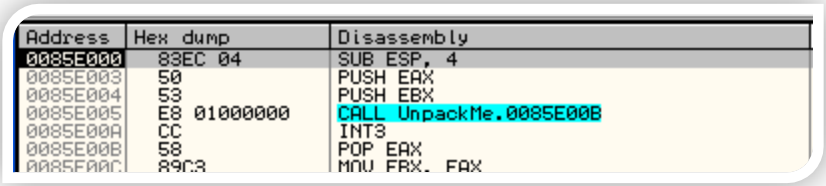
Configuración del PlugIn

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



Unpackeando

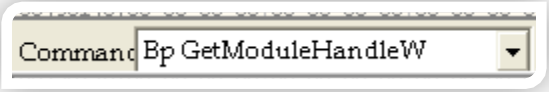
Aquí es donde comienza el laburo. Abrimos el proggie dentro del OllyDBG…



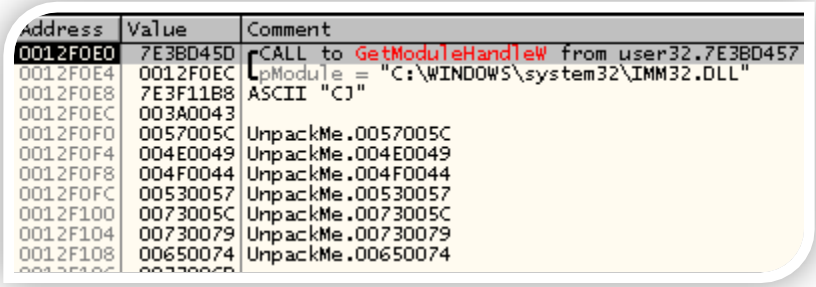
En este caso no usa la TlsTable para comenzar a desempaquetar, y estamos parados en el EP cambiado por el packer.

Lo primero que debemos hacer es llegar al OEP, pero como yo no había adquirido la información del RDG OEP Detector, si no sólo sabía que era un Delphi, pues gracias al tip de mi amigo Eddy, pude resolverlo de la siguiente manera.

Lo que debemos pensar, es un Delphi, cual es la primera API que llama?, Exacto! GetModuleHandle, en este caso pondré un BP en GetModuleHandleW para que no me detecte el BP.

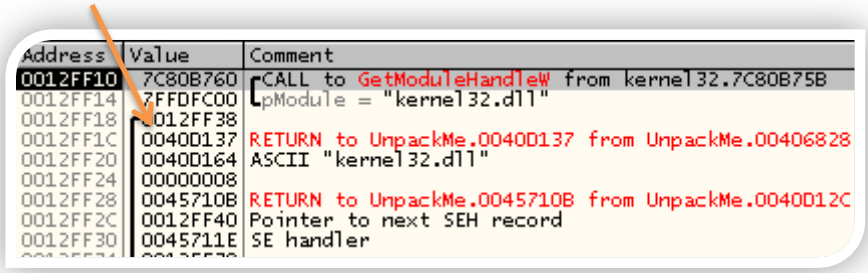


RUN, y paró.



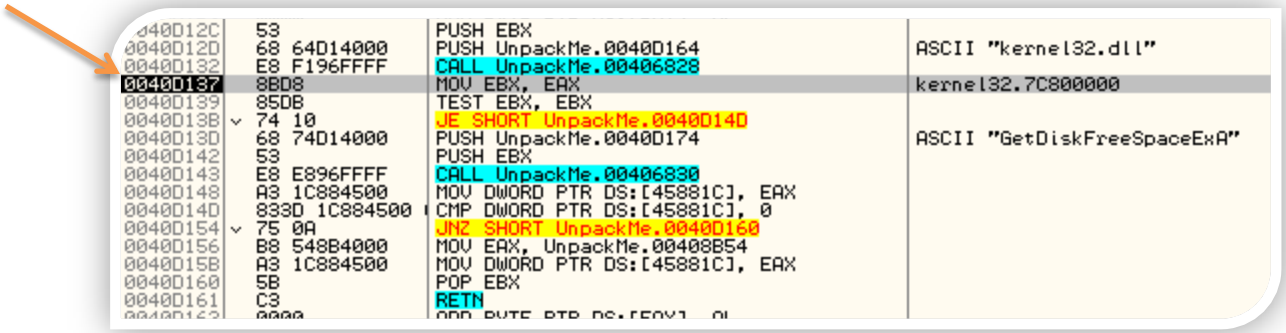
Esta es la primera vez que para, y aquí no va a ningún lugar que nos interese.

Seguimos apretando F9, hasta que lleguemos aquí →



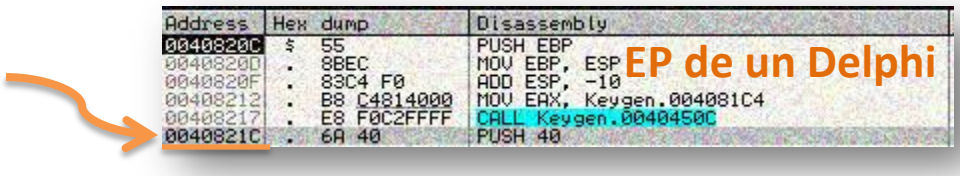
Ctrl + F9, para llegar al RETN 4, F7 para pasarlo, Ctrl + F9 de nuevo para llegar al RETN 4, y miremos la pila.

Por fin un retorno correcto, pasamos el RETN 4.



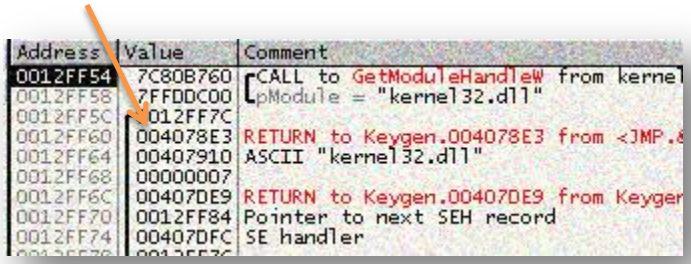
Pues… alguien sabe dónde estamos?

Fácil!, abrimos otro proggie Delphi je!, en otra instancia.



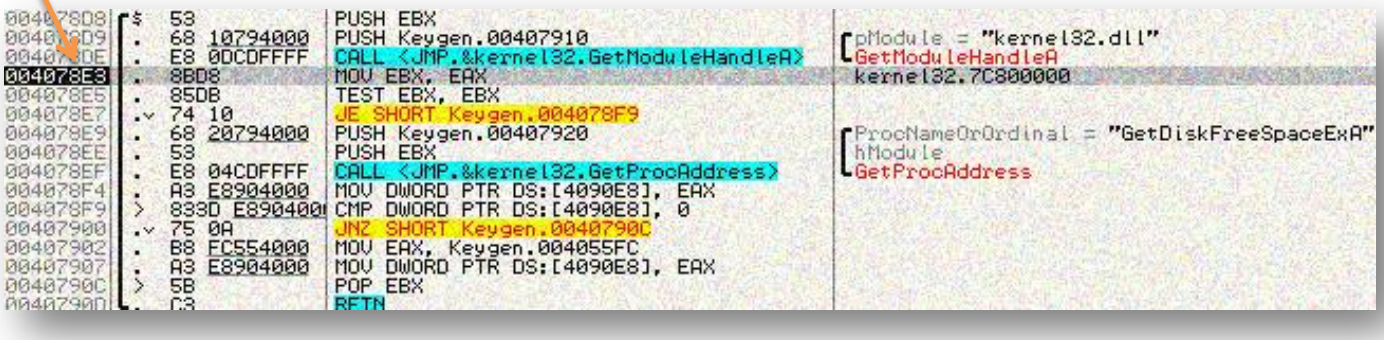
Nos familiarizamos con el EP de un Delphi.

Y hacemos lo mismo ponemos un BP en GetModuleHandleW hasta llegar al mismo lugar (del packer pe!).

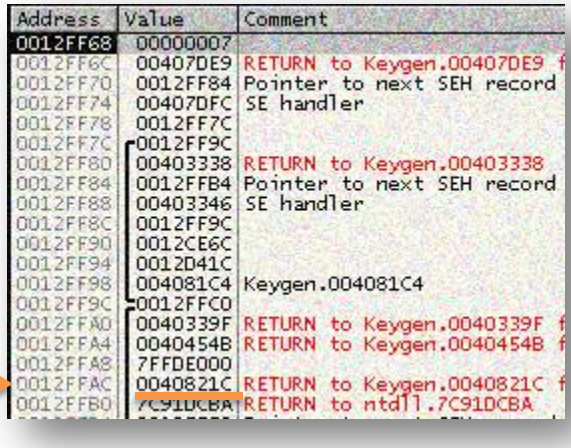


Si comparamos la pila de este proggie en Delphi con el empaquetado, verán que son casi idénticos.

Volvamos al módulo del proggie en Delphi, con Ctrl + F9, y pasar los dos RETN 4, como lo hicimos con el empaquetado, para así llegar definitivamente aquí →

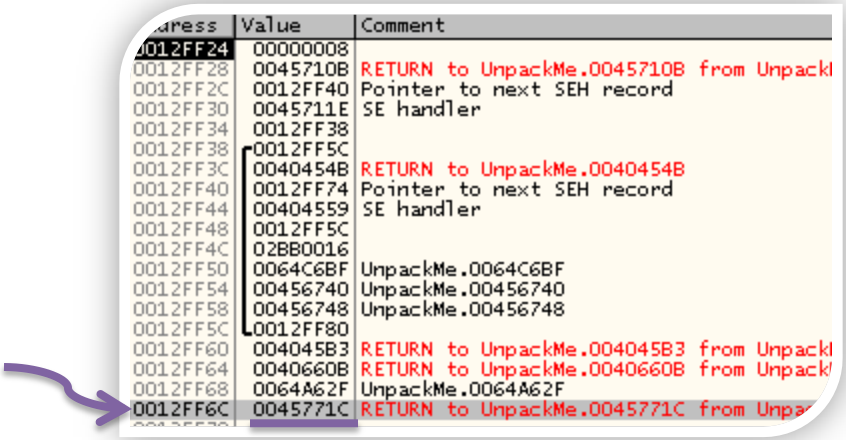


Estamos en el mismo lugar que el empaquetado, ahora si miramos la pila.

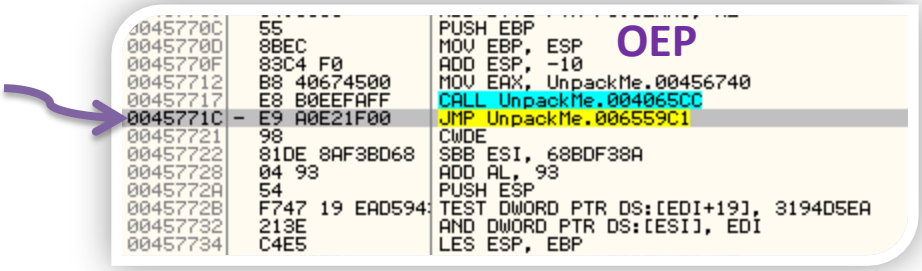


El RETURN es **0040821Ch**.

Regresamos al UnpackMe.exe, y miremos la pila.



Si comparamos las imágenes son casi idénticos, de acuerdo a eso, podemos deducir que, el RETURN es en la dirección **0045771Ch**.

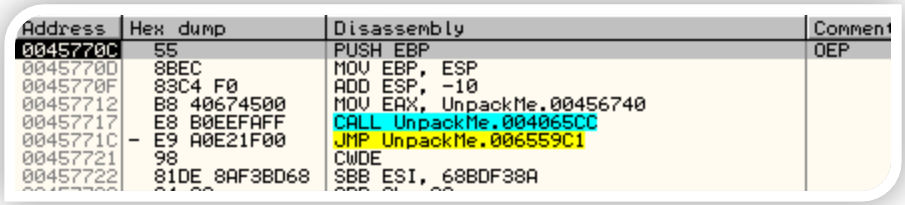


Y ahí tenemos el OEP, recalcando, si comparamos nuevamente las imágenes, pero esta vez con el EP del Delphi, verán lo idéntico, de esa manera podemos estar seguros de que estamos en el OEP, sin ninguna duda.

OEP = **0045770Ch**

El RDG OEP Detector estaba en lo correcto, en fin…

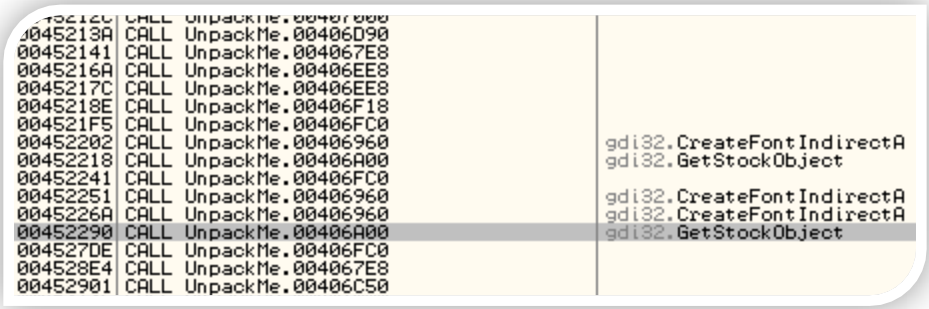
Si ponemos un HE en el OEP, reiniciamos y damos RUN, veremos como para fácilmente en el OEP.



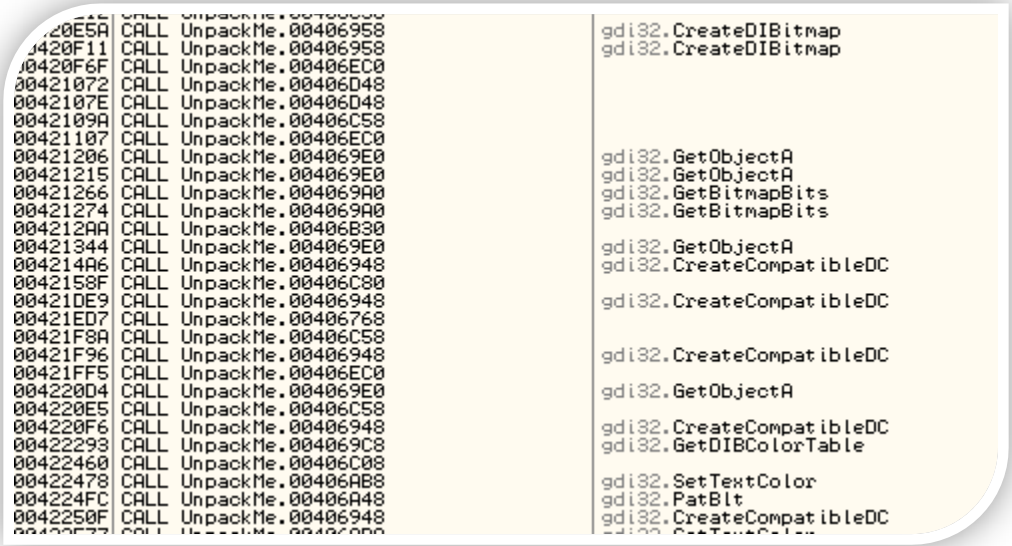
Address	Hex dump	Disassembly	Comment
0045770C	55	PUSH EBP	OEP
0045770D	8BEC	MOV EBP, ESP	
0045770F	83C4 F0	ADD ESP, -10	
00457712	B8 40674500	MOV EAX, UnpackMe.00456740	
00457717	E8 B0EEFAFF	CALL UnpackMe.004065CC	
0045771C -	E9 A0E21F00	JMP UnpackMe.006559C1	
00457721	98	CWDE	
00457722	81DE 8AF3BD68	SBB ESI, 68BDF38A	

Toca ver la IAT, los JMPs indirectos, y los CALLs que hacen referencia a ellos.

Intermodular Calls:



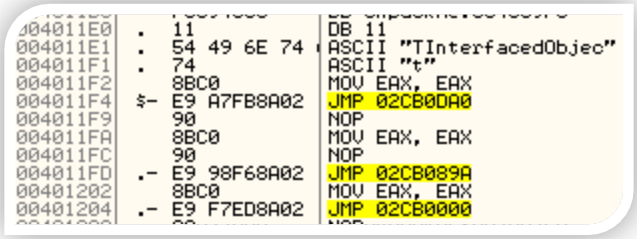
0045213A	CALL UnpackMe.00406D90	
00452141	CALL UnpackMe.004067E8	
0045216A	CALL UnpackMe.00406EE8	
0045217C	CALL UnpackMe.00406EE8	
0045218E	CALL UnpackMe.00406F18	
004521F5	CALL UnpackMe.00406FC0	
00452202	CALL UnpackMe.00406960	gdi32.CreateFontIndirectA
00452218	CALL UnpackMe.00406A00	gdi32.GetStockObject
00452241	CALL UnpackMe.00406FC0	
00452251	CALL UnpackMe.00406960	gdi32.CreateFontIndirectA
0045226A	CALL UnpackMe.00406960	gdi32.CreateFontIndirectA
00452290	CALL UnpackMe.00406A00	gdi32.GetStockObject
004527DE	CALL UnpackMe.00406FC0	
004528E4	CALL UnpackMe.004067E8	
00452901	CALL UnpackMe.00406C50	



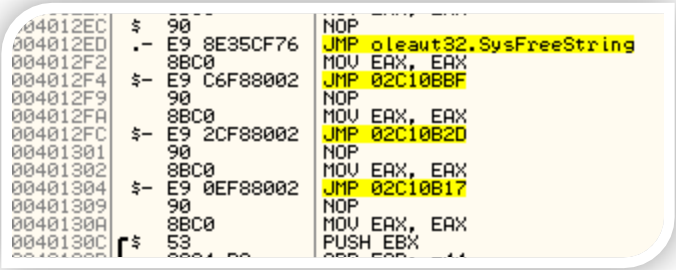
0420E5A	CALL UnpackMe.00406958	gdi32.CreateDIBitmap
0420F11	CALL UnpackMe.00406958	gdi32.CreateDIBitmap
0420F6F	CALL UnpackMe.00406EC0	
00421072	CALL UnpackMe.00406D48	
0042107E	CALL UnpackMe.00406D48	
0042109A	CALL UnpackMe.00406C58	
00421107	CALL UnpackMe.00406EC0	
00421206	CALL UnpackMe.004069E0	gdi32.GetObjectA
00421215	CALL UnpackMe.004069E0	gdi32.GetObjectA
00421266	CALL UnpackMe.004069A0	gdi32.GetBitmapBits
00421274	CALL UnpackMe.004069A0	gdi32.GetBitmapBits
004212AA	CALL UnpackMe.00406B30	
00421344	CALL UnpackMe.004069E0	gdi32.GetObjectA
004214A6	CALL UnpackMe.00406948	gdi32.CreateCompatibleDC
0042158F	CALL UnpackMe.00406C80	
00421DE9	CALL UnpackMe.00406948	gdi32.CreateCompatibleDC
00421ED7	CALL UnpackMe.00406768	
00421F8A	CALL UnpackMe.00406C58	
00421F96	CALL UnpackMe.00406948	gdi32.CreateCompatibleDC
00421FF5	CALL UnpackMe.00406EC0	
004220D4	CALL UnpackMe.004069E0	gdi32.GetObjectA
004220E5	CALL UnpackMe.00406C58	
004220F6	CALL UnpackMe.00406948	gdi32.CreateCompatibleDC
00422293	CALL UnpackMe.004069C8	gdi32.GetDIBColorTable
00422460	CALL UnpackMe.00406C08	
00422478	CALL UnpackMe.00406AB8	gdi32.SetTextColor
004224FC	CALL UnpackMe.00406A48	gdi32.PatBlt
0042250F	CALL UnpackMe.00406948	gdi32.CreateCompatibleDC

Lo bueno es que las CALLs no fueron redireccionadas a una sección creado por el packer, están intactas, es más… hay CALLs que hacen referencia a APIs.

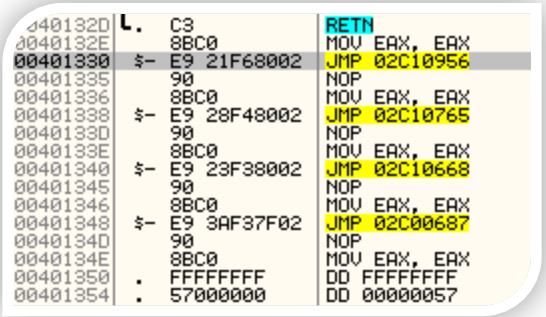
Turno de los JMPs, estos están esparcidos por varios rangos de direcciones, no como estamos acostumbrados a ver, que están en un solo rango, pero bueno esto es posible, veamos un poco de lo que menciono.



Inicio_1: 004011F4h

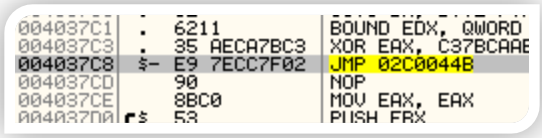


Final_1: 00401309h



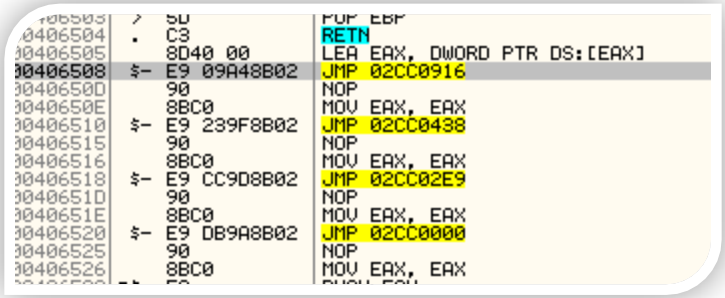
Inicio_2: 00401330h

Final_2: 0040134Dh



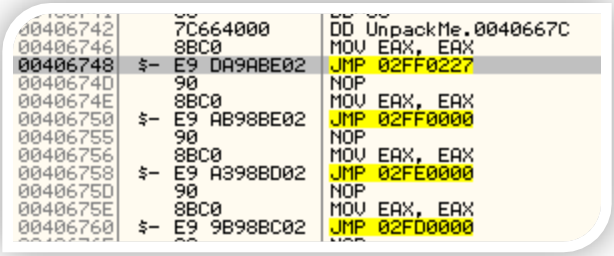
Inicio_3: 004037C8h

Final_3: 004037CDh

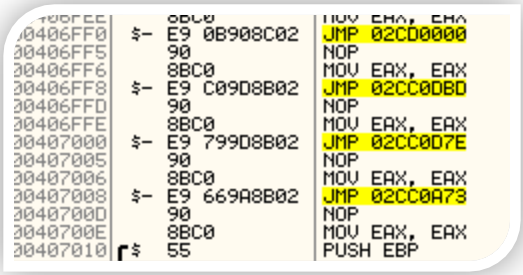


Inicio_4: 00406508h

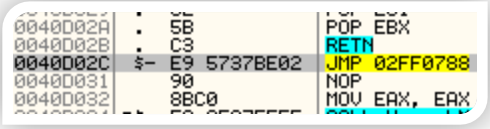
Final_4: 00406525h



Inicio_5: 00406748h

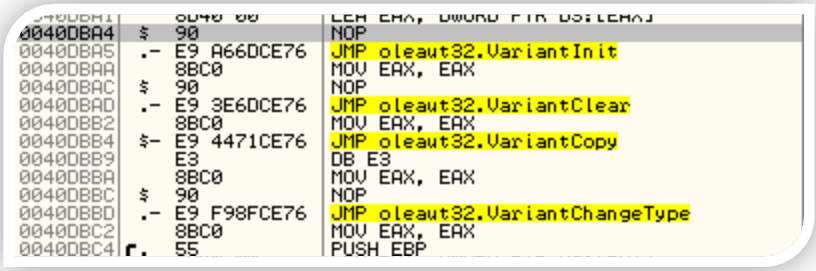


Final_5: 0040700Dh



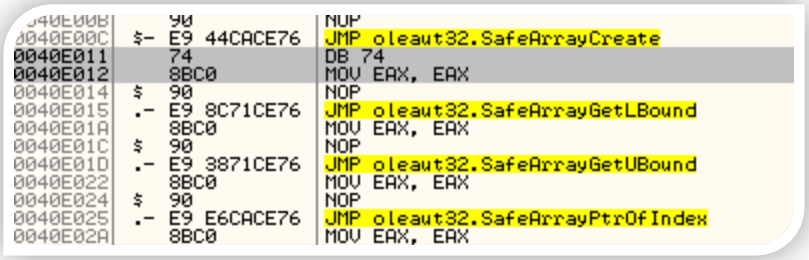
Inicio_6: 0040D02Ch

Final_6: 0040D0231h



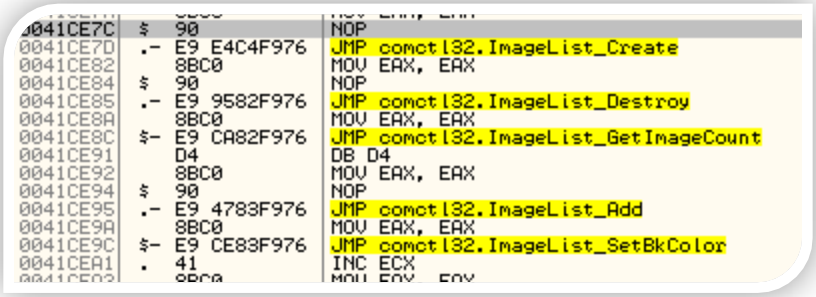
Inicio_7: 0040DBA4h

Final_7: 0040DBC1h

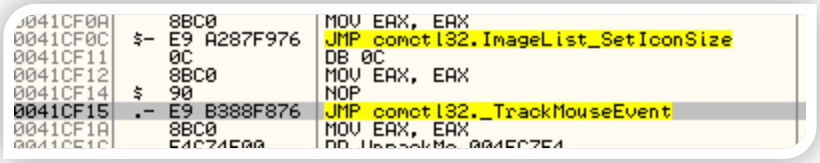


Inicio_8: 0040E00Bh

Final_8: 0040E029h



Inicio_9: 0041CE7Ch

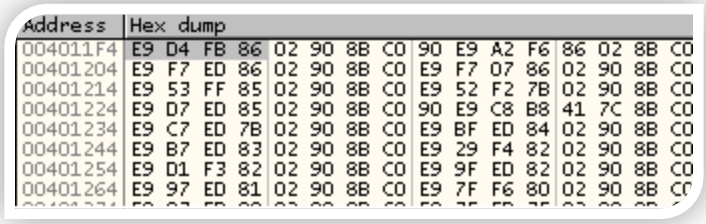


Final_9: 0041CF19h

JMPs hacia secciones creadas por el propio packer, algo que también es notorio es el NOP que podemos ver en la mayoría de los casos antes o después del JMP relativo, esto es por los JMPs indirectos y el tamaño de estos; al ser los JMPs relativos de 5 bytes y los JMPs indirectos de 6 bytes (los que se deben de usar), el packer rellena el espacio sobrante (1 byte) con NOP, o con alguna u otra cosa que se le pegue la regalada gana.

Como verán está todo dispersado los JMPs con referencia a las APIs (que deberían de ser indirectos), pero bueno esto no será problema para nosotros.

Lo próximo es reparar la IAT y los JMPs relativos a indirectos, para empezar volcamos la primera dirección de los JMPs que se encontró, **004011F4h** en el DUMP.



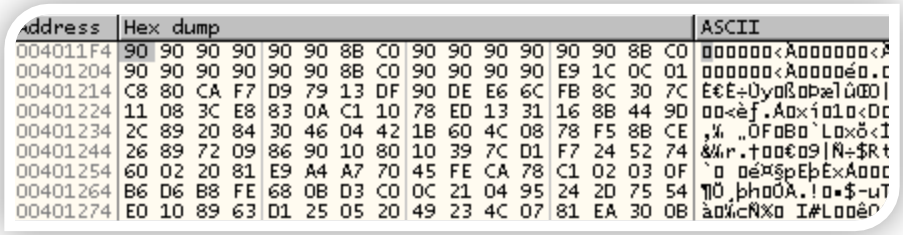
Address	Hex dump
004011F4	E9 D4 FB 86 02 90 8B C0 90 E9 A2 F6 86 02 8B C0
00401204	E9 F7 ED 86 02 90 8B C0 E9 F7 07 86 02 90 8B C0
00401214	E9 53 FF 85 02 90 8B C0 E9 52 F2 7B 02 90 8B C0
00401224	E9 D7 ED 85 02 90 8B C0 90 E9 C8 B8 41 7C 8B C0
00401234	E9 C7 ED 7B 02 90 8B C0 E9 BF ED 84 02 90 8B C0
00401244	E9 B7 ED 83 02 90 8B C0 E9 29 F4 82 02 90 8B C0
00401254	E9 D1 F3 82 02 90 8B C0 E9 9F ED 82 02 90 8B C0
00401264	E9 97 ED 81 02 90 8B C0 E9 7F F6 80 02 90 8B C0

Lo próximo sería poner un HBP on Write --> DWORD, en la dirección **004011F4h** hasta parar en el momento que escriba los bytes que hemos visto en la imagen anterior.

Reiniciamos y RUN.

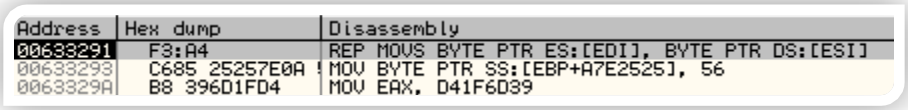
Paró, miramos el dump...

Y no son los bytes que estamos buscando, RUN nuevamente.



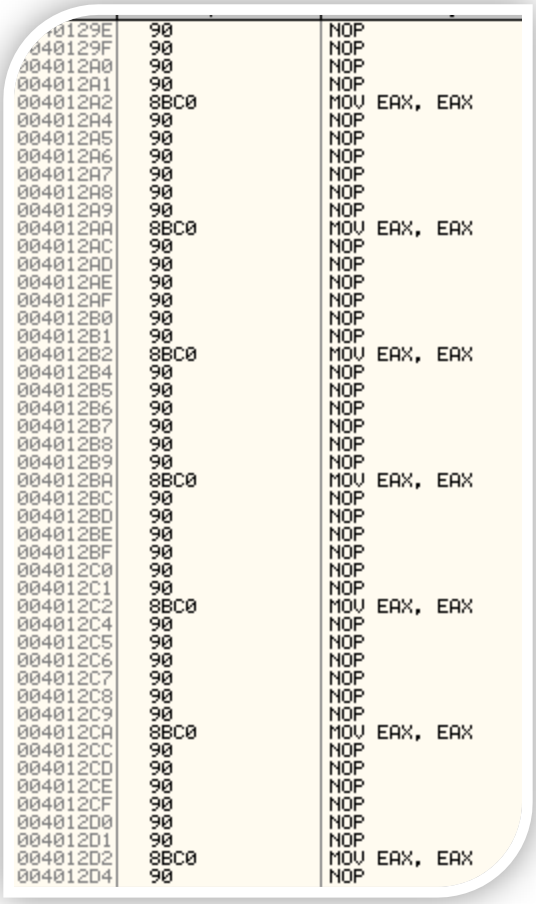
Address	Hex dump	ASCII
004011F4	90 90 90 90 90 90 8B C0 90 90 90 90 90 90 8B C0	▯▯▯▯▯▯‹À▯▯▯▯▯▯‹À
00401204	90 90 90 90 90 90 8B C0 90 90 90 90 E9 1C 0C 01	▯▯▯▯▯▯‹À▯▯▯▯é▯.▯
00401214	C8 80 CA F7 D9 79 13 DF 90 DE E6 6C FB 8C 30 7C	È€Ê÷Ùy▯ß▯Þælû▯0|
00401224	11 08 3C E8 83 0A C1 10 78 ED 13 31 16 8B 44 9D	▯▯<èƒ.Á▯x í▯1▯‹D▯
00401234	2C 89 20 84 30 46 04 42 1B 60 4C 08 78 F5 8B CE	,‰ „0F▯B▯`L▯xõ‹Î
00401244	26 89 72 09 86 90 10 80 10 39 7C D1 F7 24 52 74	&‰r.†▯▯€▯9|Ñ÷$Rt
00401254	60 02 20 81 E9 A4 A7 70 45 FE CA 78 C1 02 03 0F	`▯ ▯é¤§pEþÊxÁ▯▯▯
00401264	B6 D6 B8 FE 68 0B D3 C0 0C 21 04 95 24 2D 75 54	¶Ö¸þh▯ÓÀ.!▯•$-uT
00401274	E0 10 89 63 D1 25 05 20 49 23 4C 07 81 EA 30 0B	à▯‰cÑ%▯ I#L▯▯ê0▯

Y comenzó a nopear los bytes que serán escritas con los opcodes del JMP relativo, nopeando 6 bytes (antes, un JMP indirecto).



Address	Hex dump	Disassembly
00633291	F3:A4	REP MOVS BYTE PTR ES:[EDI], BYTE PTR DS:[ESI]
00633293	C685 25257E0A !	MOV BYTE PTR SS:[EBP+A7E2525], 56
0063329A	B8 396D1FD4	MOV EAX, D41F6D39

Quitemos el HBP, pasemos la instrucción con F8, y si miramos todas las direcciones de los JMPs que obtuvimos, verán como fueron escritos con el mismo patrón.



Y Así queda.

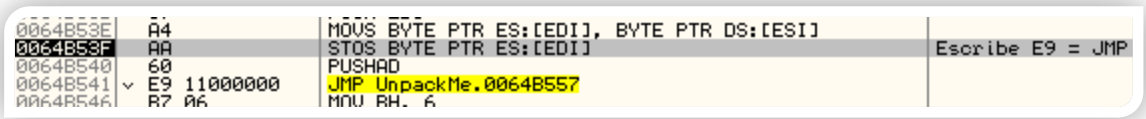
En estos momentos se pone un BPM on Write (si lo ponemos antes no lo toma :P), de la siguiente manera:

BPM on Write, Address: 4011F4h, Size: 114h

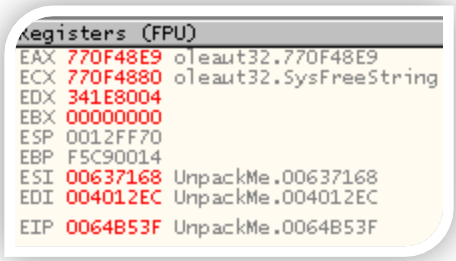
De esta forma en scripting:

BPWM 4011F4, 114

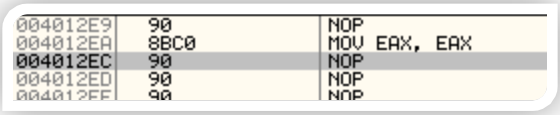
F9 RUN.



Los registros.

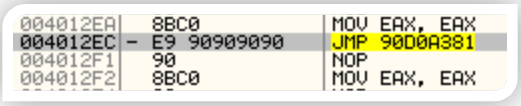


EDI en Disassembler.

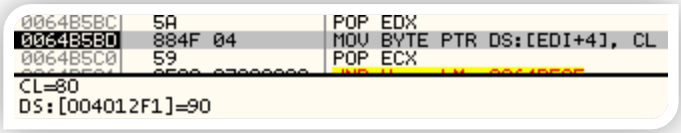


Empieza por el primer opcode, 0E9h que es el opcode del JMP relativo.

F7 Step Into, veamos como queda.

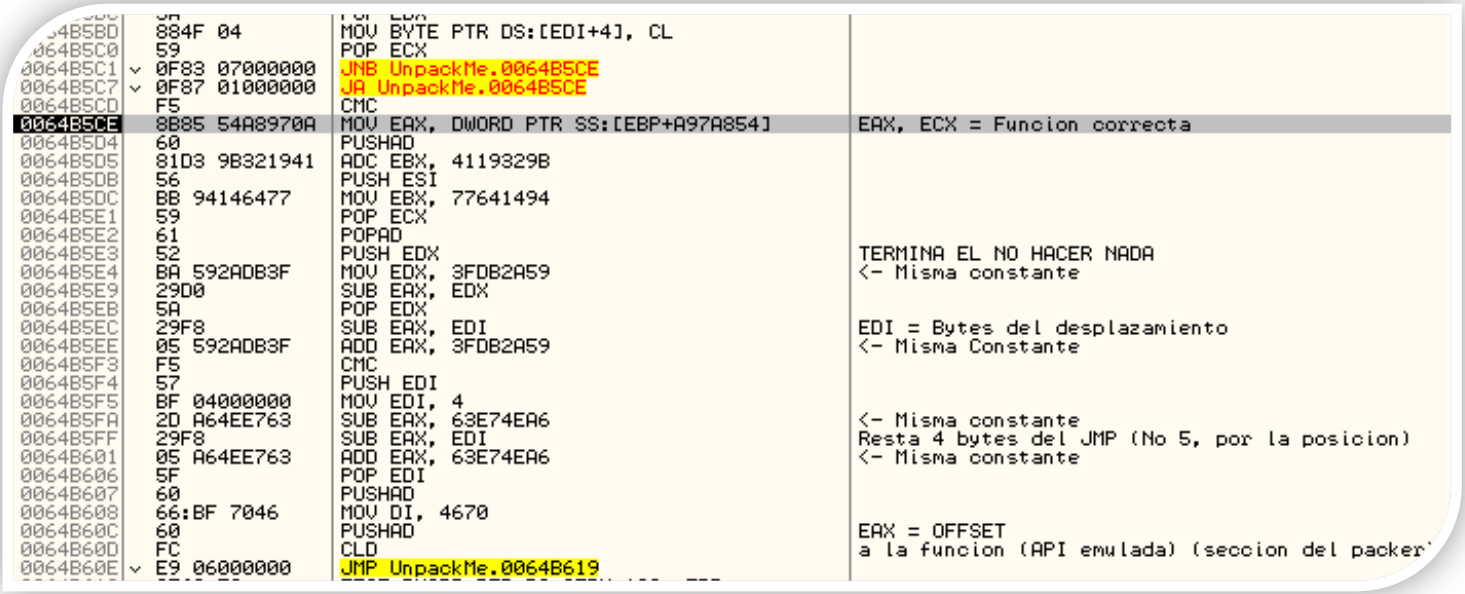


Otra vez F9 RUN, y llegamos aquí.



Aquí rellena de basura simplemente, el sexto byte, ya que al packer no le intereza =/.

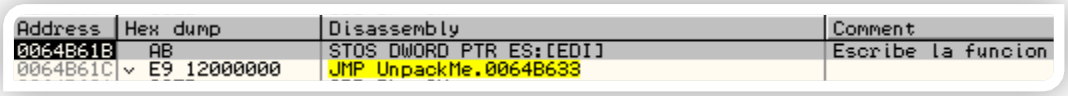
A partir de aquí podemos ir traceando para ver como calcula el offset, valor restante para completar el JMP.

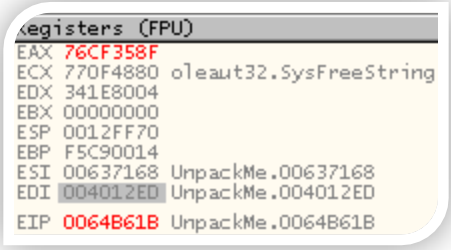


Yo ya lo tengo comentado, todo este juego de instrucciones mostrada en las imágenes no es nada más y nada menos, que, un pocas palabras una resta donde se encuentra la dirección del JMP (Origen), y la dirección dónde se encontraría la api emulada (en una sección creada por el packer) o hacia la api correcta, la diferencia de esta resta se obtendría el offset o desplazamiento.

En EAX tendríamos el offset.

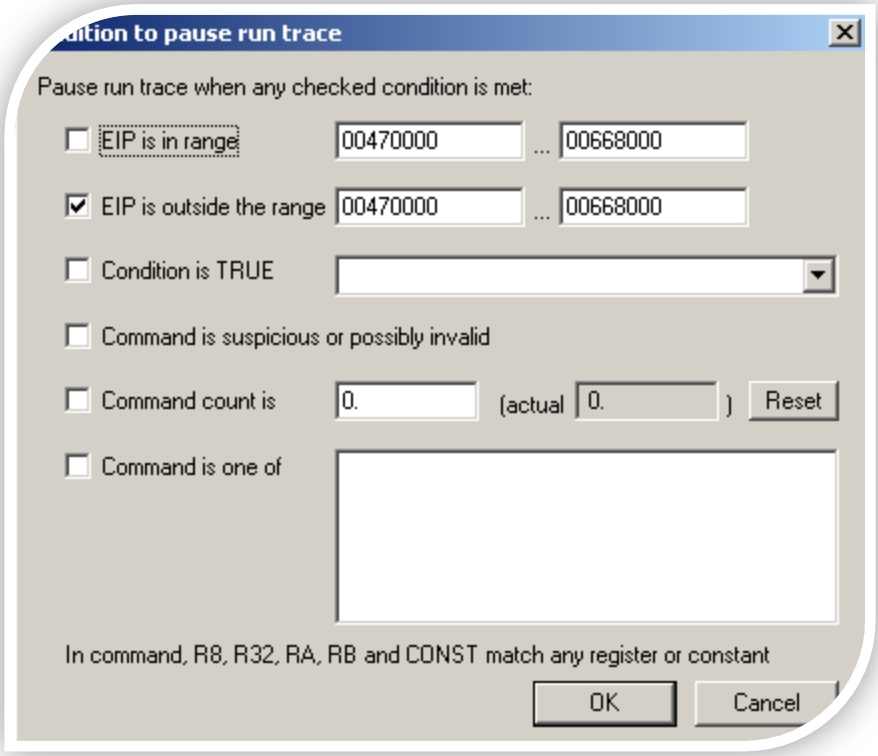
Damos RUN nuevamente.





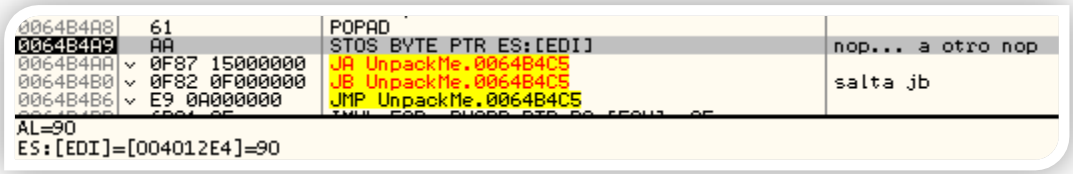
Como había mencionado, en EAX, tendríamos el desplazamiento, será escrita, y así pasará a escribir el otro JMP, hasta completarlos.

Pero como somos curiosos y hasta ahora tenemos solo es esto, hacemos un RUN TRACE.

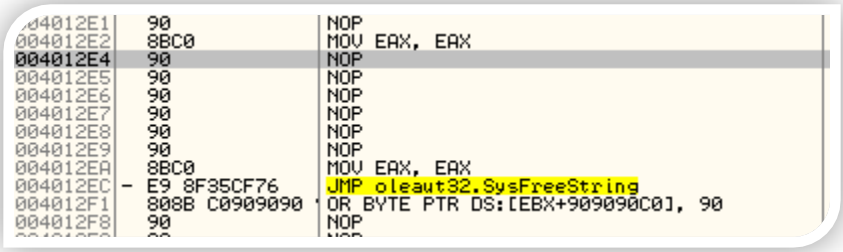


El rango le ponemos que no salga de la sección donde está. Ctrl + F11, Trace Into.

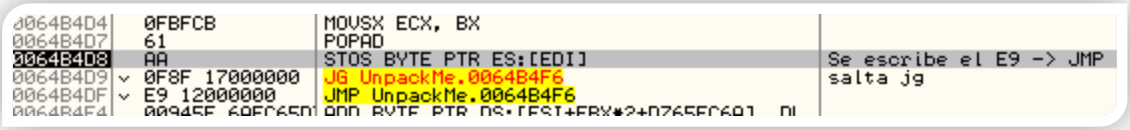
Aquí para.



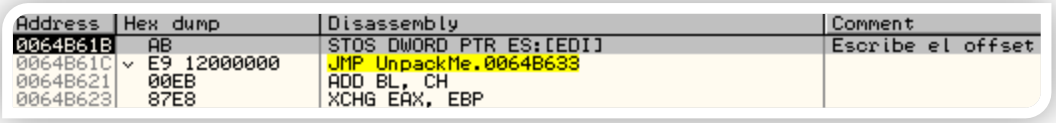
Nada de nada, lo único es que si hacemos un FOLLOW Disassembler a EDI.



El JMP relativo a la api SysFreeString que acaba de escribir, Ctrl + F11 otra vez!

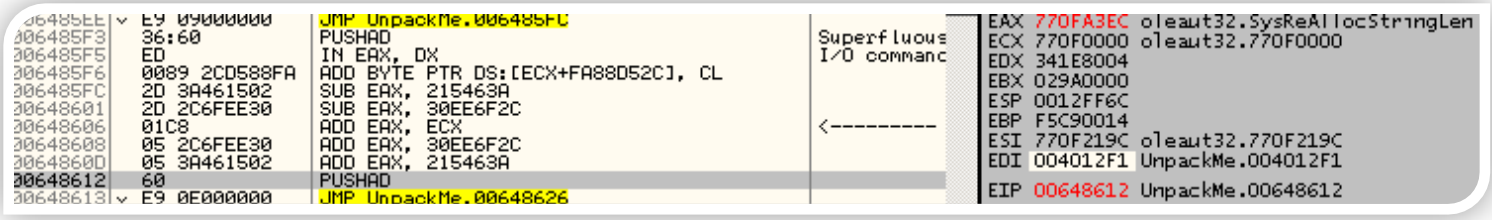


Para un poco más abajo, y escribe el opcode 0E9h del siguiente JMP a escribir. Ctrl + F11, de nuevo



Y lo mismo que acabamos de ver para el anterior JMP relativo.

Entonces retrocedamos hagamos "reversing" y vayamos para atras, a encontrar algo comprometedor.



El RUN TRACE no miente, aquí, lo que en realidad es una simple suma (solo que el packer hace operaciones de más, como todo ya sabemos) en EAX apunta a la api correcta, y a ECX a la librearía de la API.

Si loggeo en este punto a EAX y ECX, nos daremos cuenta de que en este lugar obtendremos todas las apis correctas y las librerías a las que pertenece ya con esto podemos hacer muchas cosas, pero sigamos un poco más con el análisis.

Logeamos EAX y ECX con el siguiente script:

```
WRT "Logeo.txt", "Logeo de los Registros EAX, ECX... Nox - CLS \r\n"
BPHWS 00648612
RUN

Bucle:
GN eax
EVAL "EAX: {eax}, ECX: {ecx}; Librería: {$RESULT_1}, API: {$RESULT_2}"
WRTA "Logeo.txt",$RESULT
RUN
JMP Bucle
```

Obtenemos algo así:

```
Logeo de los Registros EAX, ECX... Nox - CLS

EAX: 770F4880, ECX: 770F0000; Librería: oleaut32, API: SysFreeString
EAX: 770FA3EC, ECX: 770F0000; Librería: oleaut32, API: SysReAllocStringLen
EAX: 770F4B39, ECX: 770F0000; Librería: oleaut32, API: SysAllocStringLen
EAX: 77DA7AAB, ECX: 77DA0000; Librería: advapi32, API: RegQueryValueExA
EAX: 77DA7842, ECX: 77DA0000; Librería: advapi32, API: RegOpenKeyExA
EAX: 77DA6C17, ECX: 77DA0000; Librería: advapi32, API: RegCloseKey
EAX: 7E3B11DB, ECX: 7E390000; Librería: user32, API: GetKeyboardType
EAX: 7E3AB19C, ECX: 7E390000; Librería: user32, API: DestroyWindow
EAX: 7E3AC908, ECX: 7E390000; Librería: user32, API: LoadStringA
EAX: 7E3D07EA, ECX: 7E390000; Librería: user32, API: MessageBoxA
EAX: 7E3AC8B0, ECX: 7E390000; Librería: user32, API: CharNextA
EAX: 7C8099A5, ECX: 7C800000; Librería: kernel32, API: GetACP
EAX: 7C802446, ECX: 7C800000; Librería: kernel32, API: Sleep
EAX: 7C809B74, ECX: 7C800000; Librería: kernel32, API: VirtualFree
EAX: 7C809AE1, ECX: 7C800000; Librería: kernel32, API: VirtualAlloc
EAX: 7C8097B8, ECX: 7C800000; Librería: kernel32, API: GetCurrentThreadId
EAX: 7C80980A, ECX: 7C800000; Librería: kernel32, API: InterlockedDecrement
EAX: 7C8097F6, ECX: 7C800000; Librería: kernel32, API: InterlockedIncrement
EAX: 7C80BA61, ECX: 7C800000; Librería: kernel32, API: VirtualQuery
EAX: 7C80A164, ECX: 7C800000; Librería: kernel32, API: WideCharToMultiByte
EAX: 7C809C88, ECX: 7C800000; Librería: kernel32, API: MultiByteToWideChar
```

Una pequeña muestra.

Son todas las APIS que son usadas, esto reconfirma que el lugar es el correcto.

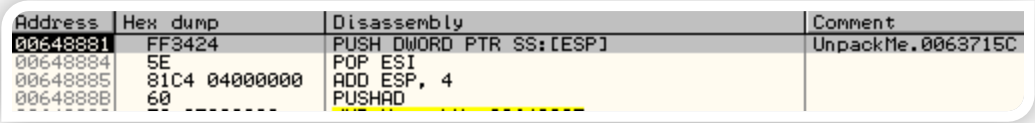
Dejamos el HE en 00648612h, reiniciamos y RUN!

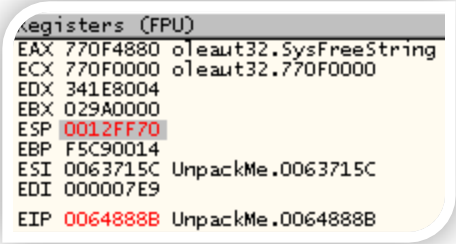
Comenzaremos a tracear, ver lo que hace primeramente las entradas buenas (es decir el camino que toma) de la IAT y los JMPs relativos, las 3 primeras son entradas buenas, luego viene la sucesión de comprobaciones y flag seteados a 1 o

0 dependiendo si es correcto o no, para eso también necesitamos encontrar la IAT, que es lo primero que debemos reparar.

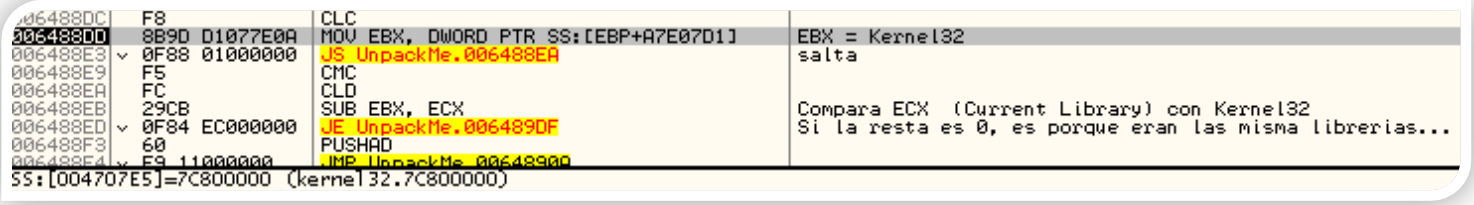
Parados por HE, comenzaremos a tracear con Step Into (almenos que indique lo contrario), y solo mostraré la información relevante, ya que hay mucha basura, ejeciones de instrucciones de más, en fin...

Examinaremos la primera entrada buena, la API como bien muestra la imagen SysFreeString.



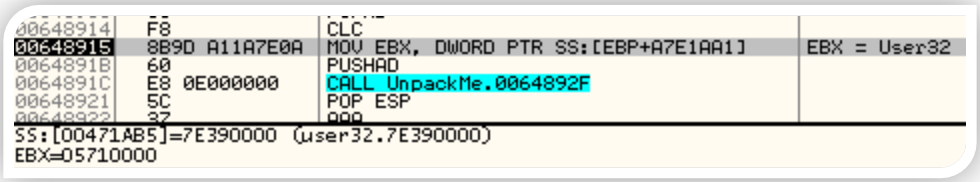


El primero cambio a los registros es en ESI, el valor que se está apunto de mover es muy importante, ya que este hace referencia a una tabla con la cual se calcula la dirección donde se escribirá el JMP relativo, y la dirección correspondiente a la entrada de la IAT de dicha API, sea correcta o no, un pequeño adelanto de lo que guarda este packer, je!



EBX tendrá el handle de Kernel32, para luego comparar si la API en cuestión pertenece a la librería Kernel32, si es así salta, hacia otro lugar de comprobaciones, en ese lugar es el último recurso, y ahí se decide si va hacer una entrada mala y JMP relativo a la sección creada por el packer, o una entrada buena y un JMP relativo a la api.

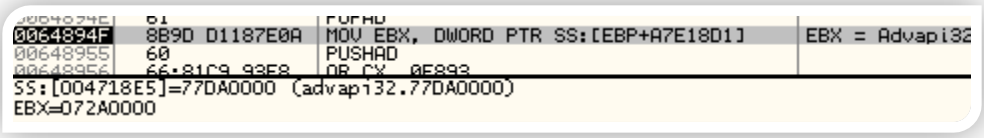
La resta como bien comenté enantes, es para saber si la diferencia es 0, si lo es, el packer se encarga de su gestión como bien yo lo mencioné. Como en esta ocasión no se cumple, así que no salta, (aunque tampoco es objetivo estudiar todos los caminos), llegando al JS que siempre saltará por el CLC anterior, sigamos.

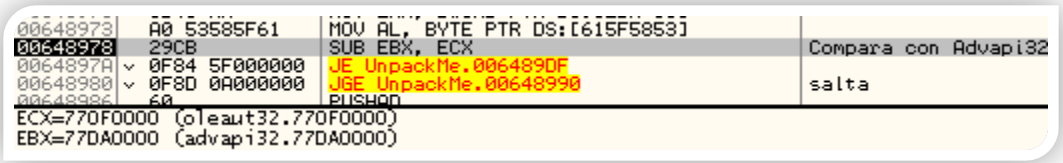


Ahora EBX tendrá el handle de User32.

Y se repite el mismo patrón anterior, como no se cumple la condición, no salta.

El turno de Advapi32.



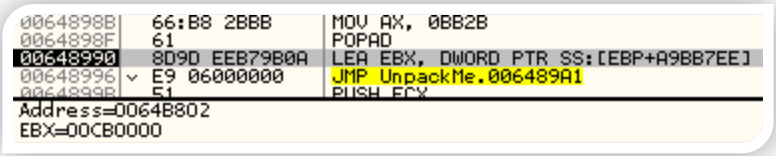


El packer comprueba si la librería a la que pertenece la API es de User32, Kernel32 o Advapi32, luego de esto son comparadas con APIs específicas de cada librearía, si la comprobación es correcta, posiblemente se podría escribir la entrada de la IAT correcta y el JMP relativo a la API.

Comentando un poco estas comprobaciones, si las comprobaciones son incorrectas y seguirán el camino bueno, pero si son correctas, es decir salta

porque la API actual le pertenece a alguna librería, Kernel32, User32 o Advapi32, pasa a otra tanda de comprobaciones, para APIs especificas que no irán por el camino bueno, uno de estos es la API ExitProcess, si fueron observadores, al llegar al OEP, en la tanda de JMPs que vimos antes, pudieron observar el JMP relativo a esta API, esta no es la única API, si no hay otras que son especificadas para que vayan por el camino bueno, pero (como todo), esto es porque en la tabla (que veremos específicamente más adelante), tiene el valor que al hacer las operaciones (del camino bueno) obtiene la dirección válida.

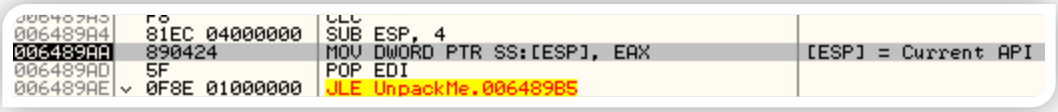
tabla[i] = Addy_X[i] + (Operaciones) = Addy[i] ;Dirección válida



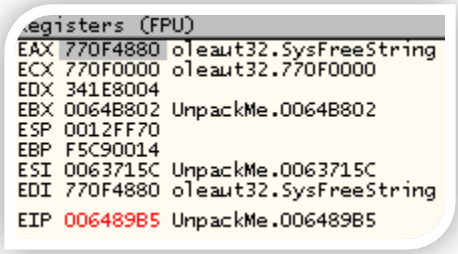
Usando a EBX como registro general, y preservando los demás registros.

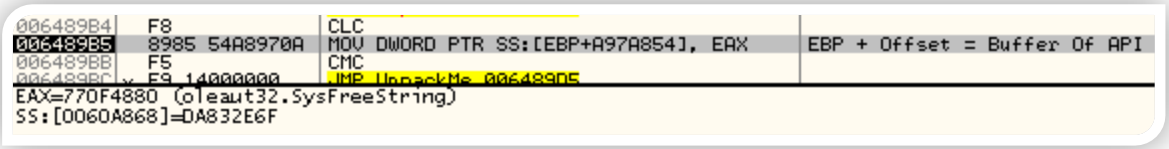
Siendo EBX = **0064B802h**

Pasamos esa CALL con F8, no hay nada interesante en la subrutina, para efectos del tute.

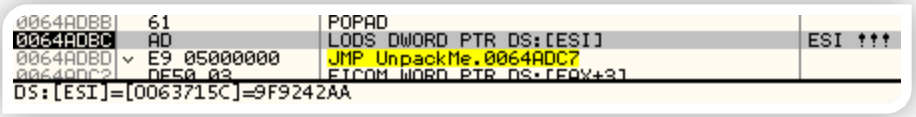


EDI = Actual API.

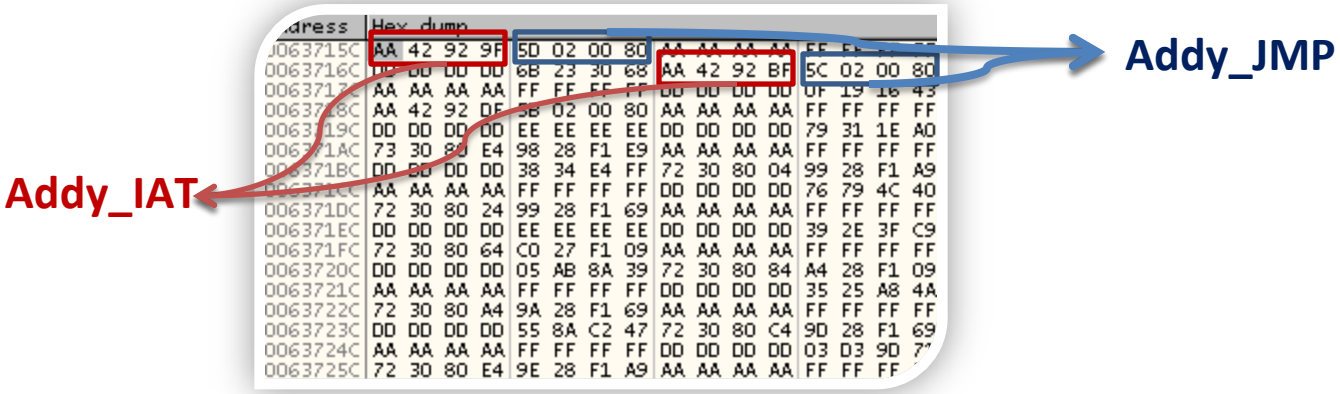




[**0060A868h**] = Actual API.



Recuerdan lo que mencioné sobre ESI?, volquemos en el dump para mayor claridad.



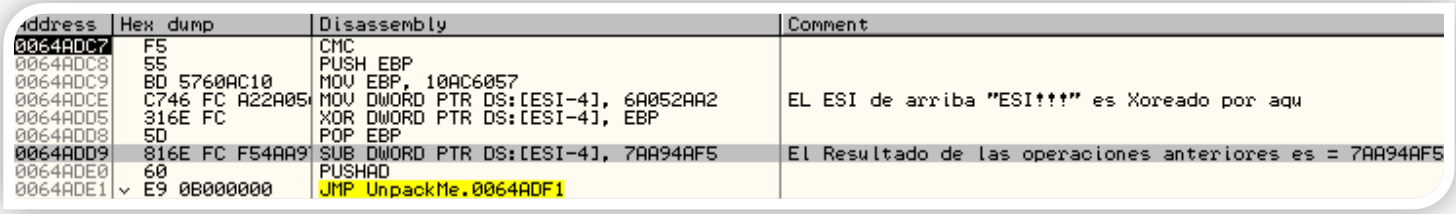
Esta es la tabla a la que me refería, lo que está en **guinda**, hace referencia a la address de la IAT dónde se escribirá la entrada respectiva, lo de **azul** indica la address dónde se escribirá el JMP relativo correspondiente.

Este mismo patrón se repite para todas, sea la entrada correcta a la IAT o no, sea el JMP relativo a la API o a una sección creada por el packer.

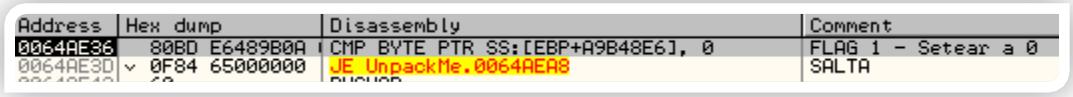
Se toma estos dos valores, se hacen ciertas operaciones correspondientes a la entrada de la IAT (correcta o incorrecta) y al JMP relativo (a la sección o a la API), en cualquiera de los casos, son operaciones independientes únicas (al menos hasta dónde he visto) para cada caso. Es decir si la entrada es reedirecionada a una address de la sección creada del packer, quiere decir que también el JMP relativo será hacia una address a una sección creada por el packer, estos tienen un algoritmo que toma las entradas de la tabla, señaladas en las imágenes y obtienen las direcciones válidas, de igual manera en caso contrario.

De estas operaciones obtenemos la address correcta, dónde empezará a escribir, por eso dije que era muy importante, en mi intento por tratar de que el packer ponga las entradas correctas en la IAT vi por encima como llenaban esta tabla, pero como saben la enfermedad me quitó mucho tiempo… en fin, sigamos.

Entonces EAX Obtendrá el valor de [ESI].



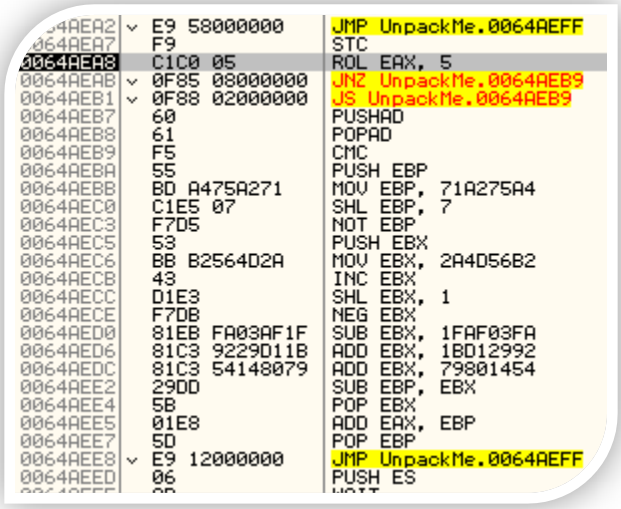
Este es el lugar donde borran los datos la tabla.



Este es otro de los flags que indican si va por el camino bueno o malo, si está en 0 camino bueno, si no, por el camino malo, creo que mencionar que cambiando solo esto, no se repara nada, es uno de los tantos FLAGs que usa el packer, si no es 0, es comparado con otro FLAG más….

Como dije… hay muchas cosas por ver que no se mostrará en este tute…

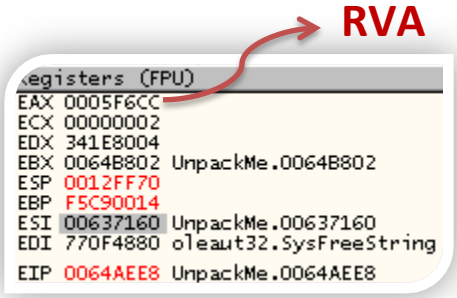
Este camino es el bueno, así que salta…

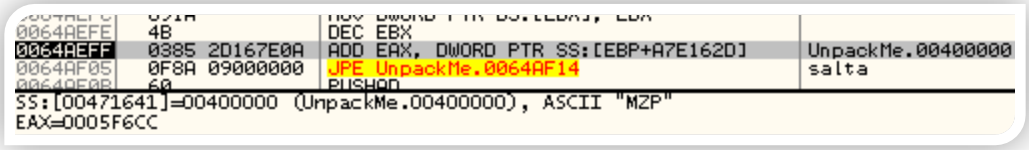


Este es el algoritmo que mencionaba, la verdad me hubiera encantando ahondar más en esto, pero ya saben porque no pude ☹, y la verdad… no sé si pueda.

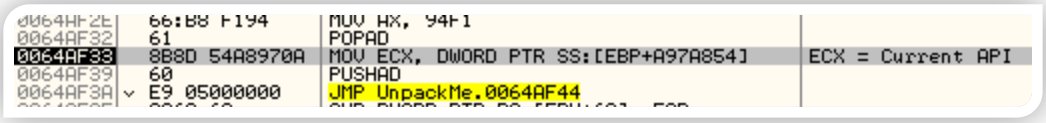
EAX, contiene el valor de la tabla Addy_IAT (como lo mencioné), EAX obtendría la RVA de la dirección válida de la Addy_IAT[0].

Addy_IAT[0] = Value + Operaciones = RVA.

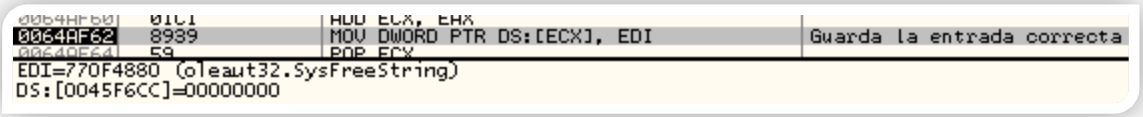




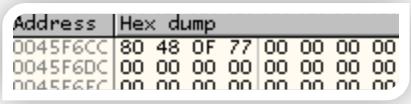
Y como es RVA…



ECX = API actual.



Para estos instantes, ECX tiene la dirección válida, y EDI la dirección de la API actual.



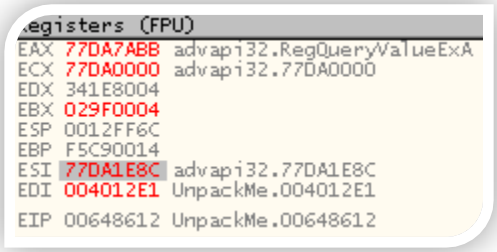
Ahí está la IAT, comienza en la 0045F6BCC

Como no es necesario para este tute especificar las operaciones ni el camino que toma para resolver la dirección válida de la ADDY_JMP[i].

Ya antes vimos que en EDI (para el camino bueno) con la instrucción STOS, la dirección donde se escribirá el JMP relativo y el lugar dónde escribe el desplazamiento son cruciales en el desarrollo del script.

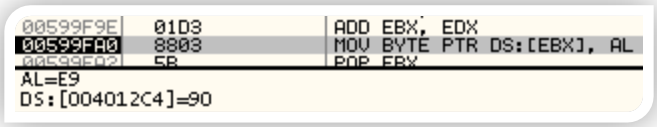
Lo segundo sería ver que instrucción y que registro se usa para las entradas malas, así que recordando lo anterior de poner un BPM como la anterior vez.

Antes de eso, pasemos las 3 primeras APIs, las cuales son del camino bueno, y llegaremos a la cuarta que es del camino malo, es decir aquí →

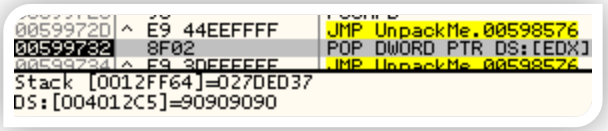


Estamos parados en la dirección donde se obtiene las todas las APIS, y el valor de EAX, muestra la API, esta es la del camino malo, ahora pongamos el BPM.

BPM on Write, Address: 4011F4h, Size: 114h



Ahí está la dirección, se escribe el E9 del JMP relativo, pero no es lo que queremos, F9 otra vez.



Registro EDX, tiene la dirección del JMP a escribir, y en estos momentos se escribe el offset.

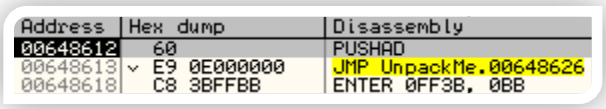
Scriptiando

Después de todo lo visto podemos codificar el script en OllyScript.

```
BPHWC
VAR hLib
VAR lpAPI
VAR IAT
VAR AddyJMP
VAR Offset
```

```
//OEP
BPHWS 45770C
// INICIO DE LA IAT
MOV IAT, 0045F6CC
// EAX = API.LIBRERIA
// ECX = LIBRERIA
BPHWS 648612 // ADDY DONDE OBTIENE LAS APIS
RUN
```

Luego de poner declarar las variables a usar en el script, inicializar la variable del comienzo de la IAT, HE en el OEP y en el Address donde se obtiene las APIs, damos RUN.



Ponemos un BPM on Write, Address: 4011F4h, Size: 1BD25.

Guardamos el valor de librería actual de la API en la variable hLib, para su comprobación posterior y guardamos el address de la API actual en la variable lpAPI.

```
BPRM 4011F4, 1BD25 //<- COMIENZA JMPS RELATIVOS - IAT
BPWM 4011F4, 1BD25
MOV hLib, ecx
```

Comparamos el valor que contiene la variable hLib es igual al registros ECX que contiene el valor de la librearía actual, si lo son, salta a la etiqueta "Comprobar". Si no son iguales, quiere decir que debemos darle un espacio de 0s (DWORD) para especificar que comienza otra tanda de APIs de diferente librería.

EP:

```
MOV lpAPI, eax
CMP hLib, ecx
JZ Comprobar

MOV [IAT], 0
ADD IAT, 4
MOV hLib, ecx
```

Comprobar:

```
RUN
CMP eip, 0045770C
JZ Salir

CMP [eip], 0AB, 1
JZ Bueno

CMP [eip], 028F, 2
JZ Malo
JMP Comprobar
```

Parados en la etiqueta "Comprobar".

Se comprueba si son los opcodes de la instrucción que se usa para escribir el offset del JMP relativo, si la comprobación es correcta para uno de estos dos, salta a su respectiva etiqueta, si es correcta para la comprobación del OEP quiere decir que ya se terminó de escribir la IAT y JMPs indirectos y que debe salir, saltando a la etiqueta "Salir".

Malo:

```
MOV AddyJMP, edx
JMP Escribir
```

Bueno:

```
MOV AddyJMP, edi
```

Al estar en las etiquetas "Bueno", "Malo", se preserva el valor del registro EDI, EDX, respectivamente para su posterior gestión.

Escribir:

```
STI
CMP [IAT], lpAPI
JZ Seguir

MOV [IAT], lpAPI
```

Recordamos que si la API va por el camino correcto, en la IAT se escribe la entrada válida, para esto es la etiqueta "Escribir", si la IAT ya es válida no hay necesidad de escribir la entrada, pero si no lo es, escribe la válida.

Seguir:

```
CMP [AddyJMP - 2], 90, 1
JNZ Continua
DEC AddyJMP
```

En la etiqueta “Seguir”, buscamos algún NOP anterior del JMP, si lo encontramos, comenzamos a escribir a partir de ese NOP los JMPs indirectos, cosa de estética, pero para los que piensen que los CALLs relativos (las que hacen referencia a los JMP indirectos), no hacen referencia a los NOPs si no al JMP, se equivocan, si hace referencia a los NOPs, es decir de estos CALLs no lo toca para nada!.

Continua:

```
MOV [AddyJMP - 1], 25FF, 2
MOV [AddyJMP + 1], IAT

ADD IAT, 4

RUN

JMP lala
```

La etiqueta "Continua" lo que hace es simplemente escribir el JMP indirecto hacia la el address de la IAT con la entrada válida, se le suma un DWORD para la siguiente entrada y vuelve al bucle.

lala:

```
CMP eip, 00648612
JZ EP

CMP eip, 0045770C
JZ Salir

RUN

JMP lala
```

Si está parado en el OEP, por supuesto, sale, y se termina el Script.

Veamos como lo dejó, mandemos el Script.

```
004011F4  $- FF25 78F7450( JMP DWORD PTR DS:[45F778]          kernel32.GetStdHandle
004011FA     8BC0         MOV EAX, EAX
004011FC  .- FF25 74F7450( JMP DWORD PTR DS:[45F774]          kernel32.RaiseException
00401202     8BC0         MOV EAX, EAX
00401204  .- FF25 70F7450( JMP DWORD PTR DS:[45F770]          kernel32.RtlUnwind
0040120A     8BC0         MOV EAX, EAX
0040120C  $- FF25 6CF7450( JMP DWORD PTR DS:[45F76C]          kernel32.UnhandledExceptionFilt
00401212     8BC0         MOV EAX, EAX
00401214  $- FF25 68F7450( JMP DWORD PTR DS:[45F768]          kernel32.WriteFile
0040121A     8BC0         MOV EAX, EAX
0040121C  $- FF25 FCF6450( JMP DWORD PTR DS:[45F6FC]          user32.CharNextA
00401222     8BC0         MOV EAX, EAX
00401224  $- FF25 64F7450( JMP DWORD PTR DS:[45F764]          kernel32.CompareStringA
0040122A     8BC0         MOV EAX, EAX
0040122C     90           NOP
0040122D  .- FF25 60F7450( JMP DWORD PTR DS:[45F760]          kernel32.ExitProcess
00401233     C0           DB C0
00401234  $- FF25 F8F6450( JMP DWORD PTR DS:[45F6F8]          user32.MessageBoxA
0040123A     8BC0         MOV EAX, EAX
0040123C  $- FF25 5CF7450( JMP DWORD PTR DS:[45F75C]          kernel32.FindClose
00401242     8BC0         MOV EAX, EAX
00401244  $- FF25 58F7450( JMP DWORD PTR DS:[45F758]          kernel32.FindFirstFileA
0040124A     8BC0         MOV EAX, EAX
0040124C  $- FF25 54F7450( JMP DWORD PTR DS:[45F754]          kernel32.FreeLibrary
00401252     8BC0         MOV EAX, EAX
00401254  $- FF25 50F7450( JMP DWORD PTR DS:[45F750]          kernel32.GetCommandLineA
0040125A     8BC0         MOV EAX, EAX
0040125C  $- FF25 4CF7450( JMP DWORD PTR DS:[45F74C]          kernel32.GetLocaleInfoA
00401262     8BC0         MOV EAX, EAX
00401264  $- FF25 48F7450( JMP DWORD PTR DS:[45F748]          kernel32.GetModuleFileNameA
0040126A     8BC0         MOV EAX, EAX
0040126C  $- FF25 44F7450( JMP DWORD PTR DS:[45F744]          kernel32.GetModuleHandleA
00401272     8BC0         MOV EAX, EAX
00401274  $- FF25 40F7450( JMP DWORD PTR DS:[45F740]          kernel32.GetProcAddress
0040127A     8BC0         MOV EAX, EAX
0040127C  $- FF25 3CF7450( JMP DWORD PTR DS:[45F73C]          kernel32.GetStartupInfoA
00401282     8BC0         MOV EAX, EAX
00401284  $- FF25 38F7450( JMP DWORD PTR DS:[45F738]          kernel32.GetThreadLocale
0040128A     8BC0         MOV EAX, EAX
0040128C  $- FF25 34F7450( JMP DWORD PTR DS:[45F734]          kernel32.LoadLibraryExA
00401292     8BC0         MOV EAX, EAX
00401294  $- FF25 F4F6450( JMP DWORD PTR DS:[45F6F4]          user32.LoadStringA
0040129A     8BC0         MOV EAX, EAX
0040129C  $- FF25 30F7450( JMP DWORD PTR DS:[45F730]          kernel32.lstrcpynA
004012A2     8BC0         MOV EAX, EAX
004012A4  $- FF25 2CF7450( JMP DWORD PTR DS:[45F72C]          kernel32.lstrlenA
004012AA     8BC0         MOV EAX, EAX
004012AC  $- FF25 28F7450( JMP DWORD PTR DS:[45F728]          kernel32.MultiByteToWideChar
004012B2     8BC0         MOV EAX, EAX
004012B4  $- FF25 E4F6450( JMP DWORD PTR DS:[45F6E4]          advapi32.RegCloseKey
004012BA     8BC0         MOV EAX, EAX
```

Un pequeño ejemplo, revisando la los JMPs me di cuenta de algo.

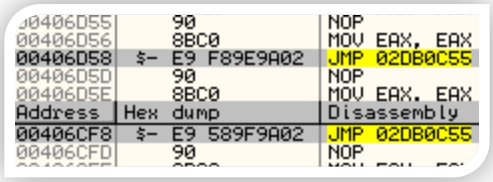
```
00406D50  $- FF25 E0F8450( JMP DWORD PTR DS:[45F8E0]
00406D56     8BC0         MOV EAX, EAX
00406D58  $- E9 A79D9902  JMP 02DA0B04
00406D5D     90           NOP
```

```
00406D96     8BC0         MOV EAX, EAX
00406D98  $- E9 4B999602  JMP 02D706E8
00406D9D     90           NOP
00406D9E     8BC0         MOV EAX, EAX
00406DA0  $- FF25 C0F8450( JMP DWORD PTR DS:[45F8C0]
```

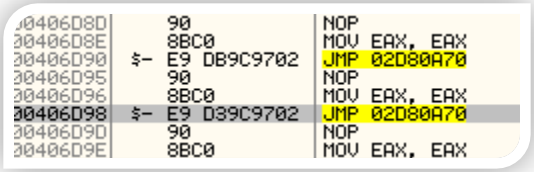
```
00406EE6     8BC0         MOV EAX, EAX
00406EE8  $- FF25 1CF8450( JMP DWORD PTR DS:[45F81C]
00406EEE     8BC0         MOV EAX, EAX
00406EF0  $- E9 0B918C02  JMP 02CD0000
00406EF5     90           NOP
```

3 JMPs que no fueron corregidos, revisando el orden de la IAT como va guardando, vi que omitía la entrada para estos saltos, pensé "muy raro", y comencé a buscar el problema, pensé que estaba en la programación del script, así que comencé a tracer y poner muchos bps para ver en que momento "fallaba" sin llegar a nada, pensé que el error estaba en la capa 8 (jeje, sí, entre el monitor y la silla U_U"), hasta que en las últimas abrí otra instancia del olly revisando como deja el packer estas entradas, y me di con la sorpresa que...

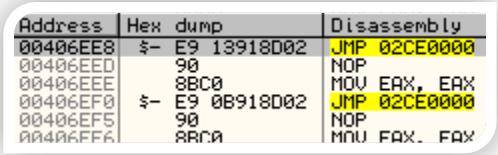
A la “pensé” mucho xD...



Luego de un Ctrl + F, Search for ---> Command... Se repiten!, por no hace referencia a la IAT, ni tampoco para en el address dónde se obtiene esta entrada, para escribir este JMP, como dije muchas cosas que ver de este packer...

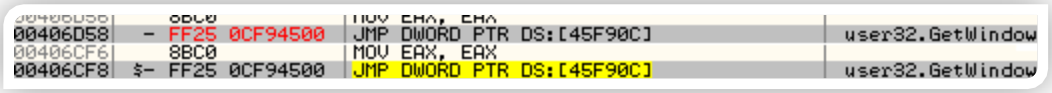


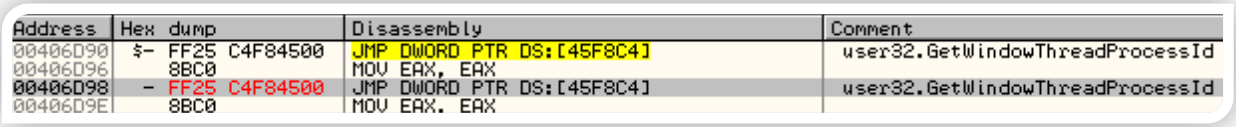
Para esto no fue necesario el Search for --> Command, veamos el último JMP sobrante.

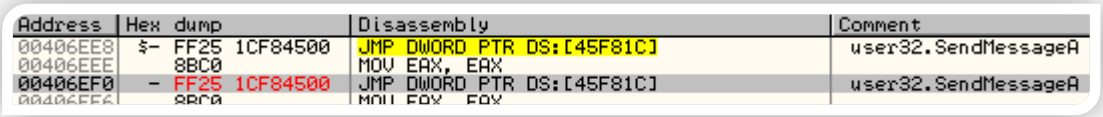


Igual que el anterior, ahora toca repararlos.

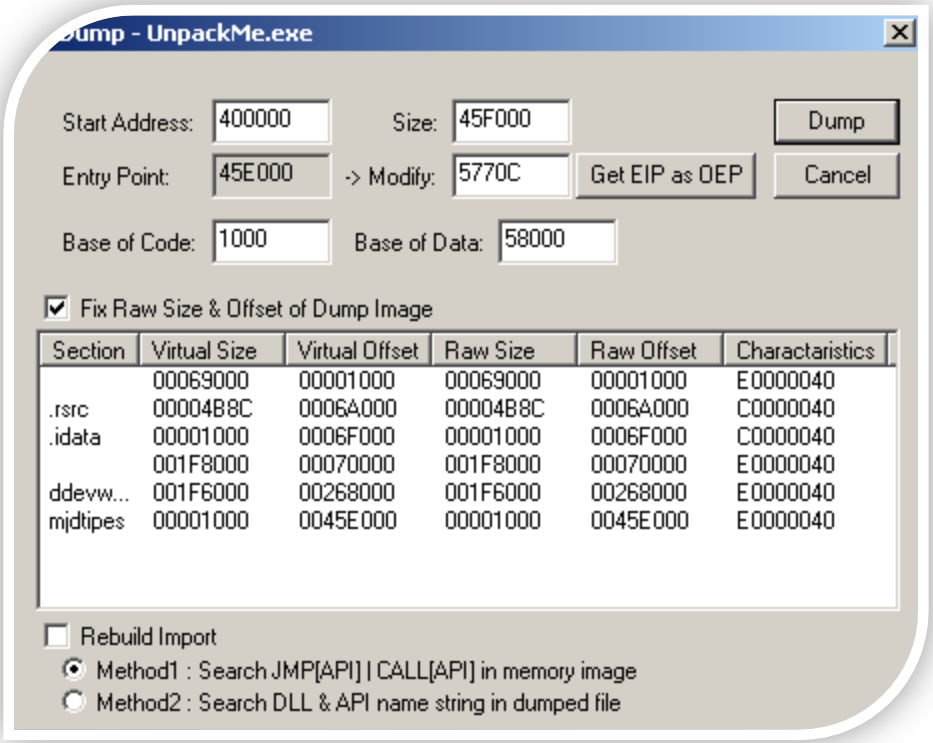
Nos dirigimos a nuestro OllyDBG con el proggie packeado.







Nos disponemos a hacer el DUMP con el OllyDUMP.

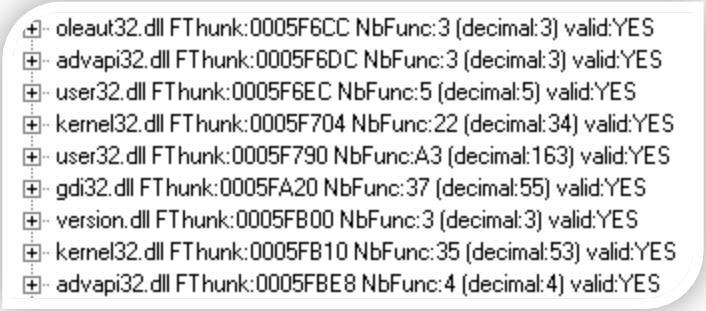


Lo guardo com dump.exe, y luego usamos el ImportRec.

Ponemos el RVA del OEP y le damos a "IAT AutoSearch" →

Nos sacó la IAT.

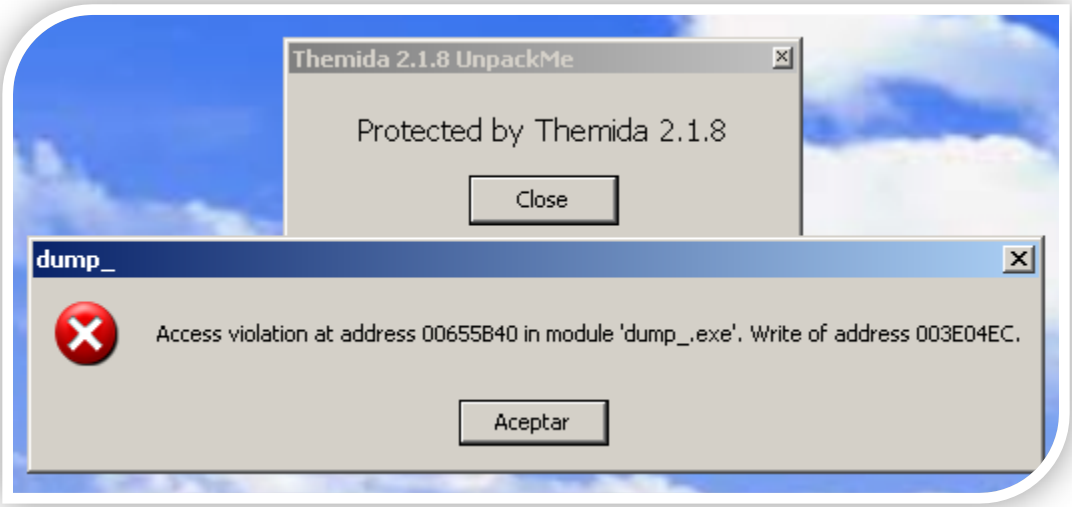
Clic a "Get Imports".



Ninguna entrada mala :)

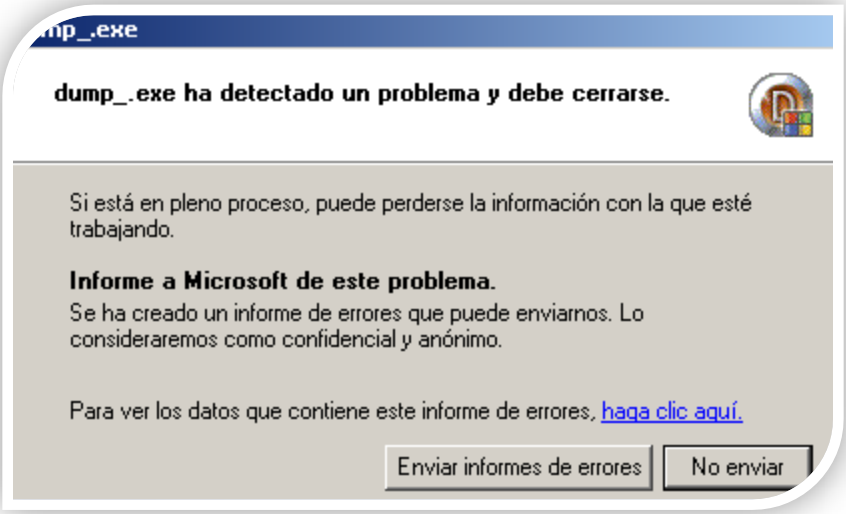
"Fix Dump", escogemos nuestro dump y lo fixiamos.

Ejecutamos, y damos clic al botón.



Upps…

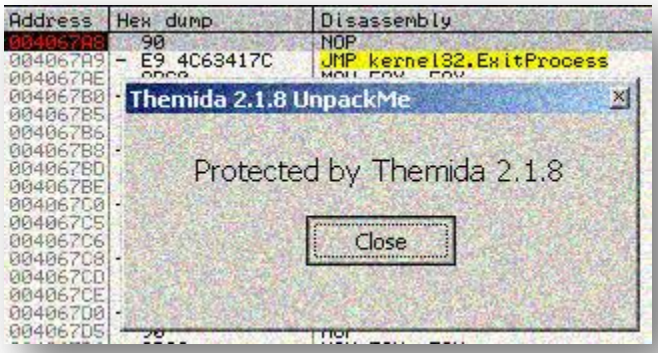
Si lo cerramos.



Para comenzar necesitamos saber el lugar que debemos fixiar, para el primer problema, el botón.

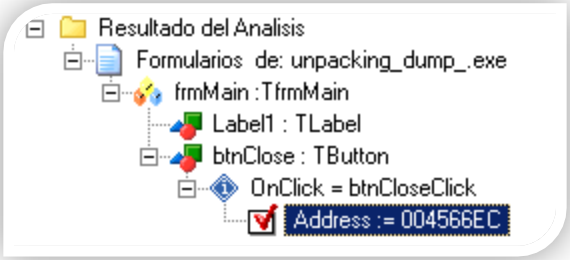
El evento onClick(), llama a ExitProcess para terminar el proceso, el que desea lo puede comprobar poniendo un BP a ExitProcess, o también en los JMPs indirectos.

Tan solo abre una instancia del proggie empaquetado, da RUN, y en ese momento pone los BPs, clic al botón.



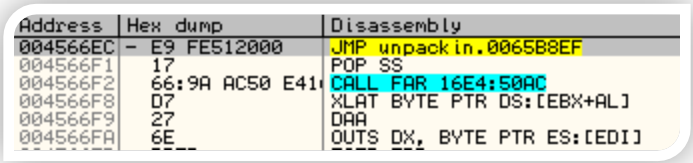
A partir de aquí, en la dirección que aparece el error al apretar el botón podemos ir hacia atras y encontraremos el lugar que debemos fixiar.

Para facilidad nuestra, usaremos la herramienta de Guan de Dio E2A.

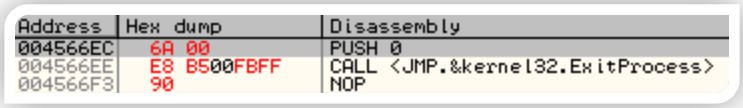


De igual manera puedes obtener la dirección del evento onClick(), poniendo BPs en los puntos mágicos para Delphi, que están muy bien documentados en la lista, para eso tienes el buscador de la web del maestro Ricardo.

Ahí está el address **004566ECh**.



El salto a la sección creada por el packer, si alguien desea lo puede seguir en el empaquetado pero se volverán viejos jejee, o tan solo puede poner bps en el lugar de la excepción, en el salto, en fin...



Guardamos los cambios.

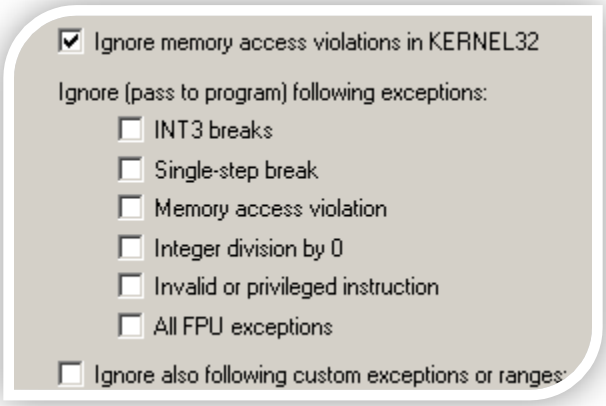
Primer error fixiado, ahora falta la X.

Este para mi fue el más problemático, si ven en el formulario (la imagen del E2A) no hay ningún evento onClose(), este evento no es gestionado, entonces el que se encarga aquí es el compilador, como lo hace el mismo delphi.

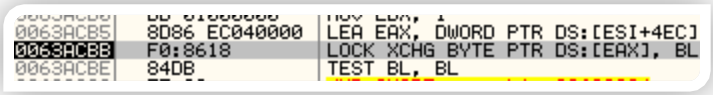
Al apretar la X también llama a la función ExitProcess, veamos en la instancia del Olly con el proggie empaquetado.

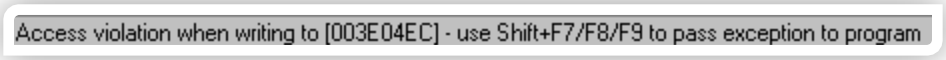
Puse el BP aprete la X para salir.

Ahora sin plugin del StrongOD, destiladamos que ignore todas las excepciones, excepto la primera.



RUN, y para aquí:





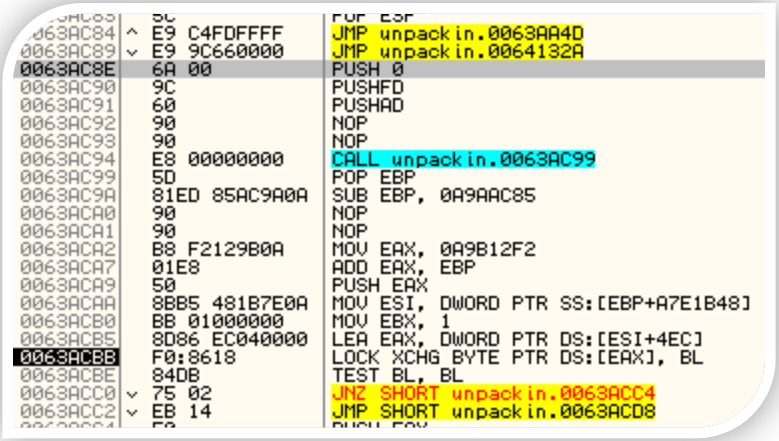
Esa es la excepción.

Conversando con Eddy y Guan de Dio, los dos me dijeron lo mismo, pon un BP en ExitProcess y hacia atras para ver en que parte debes fixiar.

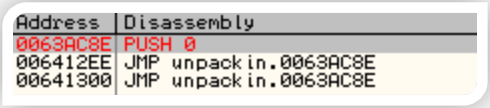
Comenzemos!!!

Observando, vemos que estamos en la sección del packer, entonces para poder fixiar debemos encontrar de que lugar de la sección .code salta y llega hasta esa dirección.

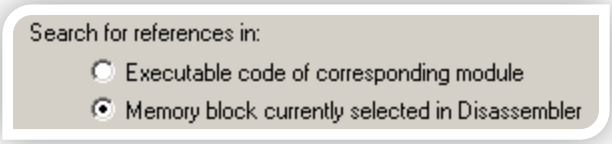
Así que vayamos hacia atras!! :)



Estamos en medio de JMPs así que esto nos da idea de que en 0063AC8E es donde comienza a ejecutarse este juego de instrucciones, para confirmar esto... hacemos un Ctrl + R para ver las referencias.

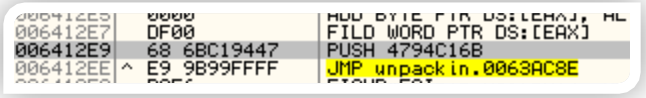


Aquí es dónde me estanqué (otra vez), unos cuantos minutos, había dos JMPs pero a ninguno de ellos nadie hacía referencia hasta que recordé la siguiente opción del Olly -->

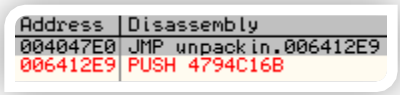


Estaba en Memory Block Currently.... pero tiene que saltar desde la sección .code, así que lo cambiamos el Radio Button a Executablecode of corresponding module.

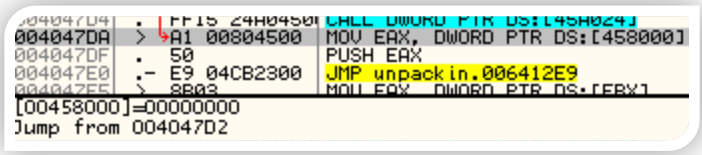
Luego nos vamos aquí -->



Ctrl + R.

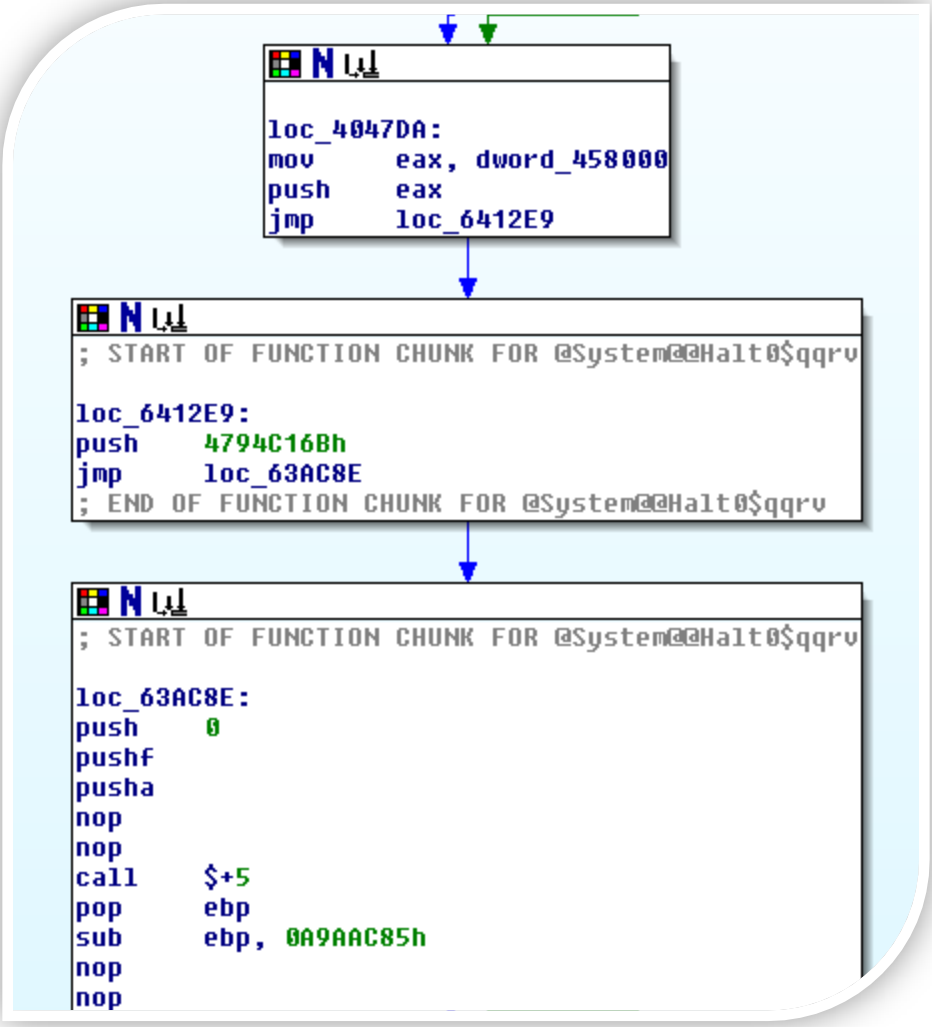


Y ahí estamos vemos como mueven EAX el valor 0, y hacen un push del mismo, muy obvio no?



Este es el punto que debemos de fixiar, la verdad que aquí para hacer esto es donde sufrí, pero gracias Eddy, Guan jee siempre refrescandome la memoria.

Escribiendo el tute, en estos momentos, recuerdo lo que había conversado con Guan, y habiá una palabrita que resaltaba, usa IDA!!!!!

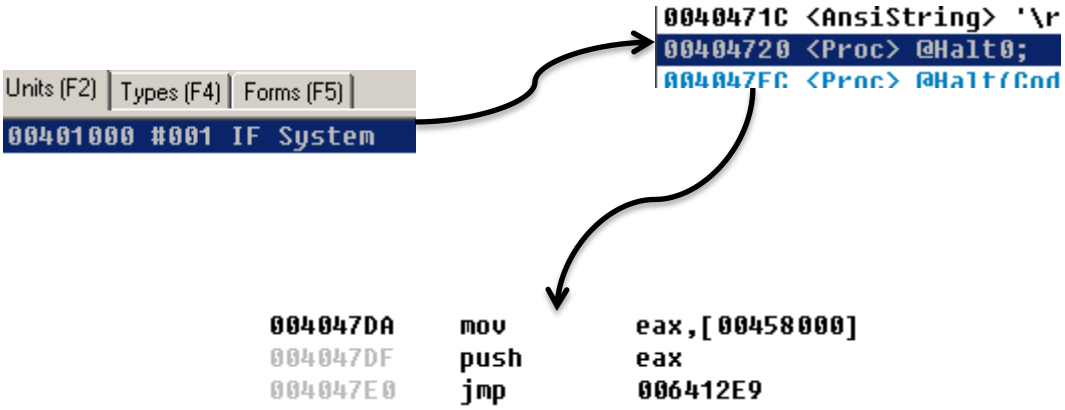


Vaya, me hubiera ahorrado mucho tiempo, si hubiera pensado en esto jeje, bueno es la primera vez que uso IDA en un tute XD...

Ahí las signatures de IDA nos dicen @System@@Halt0

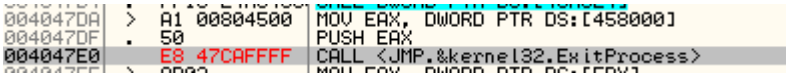
Esto ya es muy evidente.

De la misma manera, si abrimos el IDR



Deducimos que…. para estos casos IDA es el indicado.

Solo queda fixiar, cambiar el JMP por el CALL a ExitProcess.



Guardamos los cambios.

Ejecutamos, cliquiamos donde teníamos problemas… y sin ningún error todo corre normal :)

Anécdota:
El unpacked lo tenía listo un día antes de que terminara el concurso, el script también lo tenía listo, el lunes que empieza la semana, como todo al trabajo, y llegando nos envían un mail dando aviso de un nuevo malware y con el link correspondiente, al bajarlo y pasarle el PiD, me di con la sorpresa de que era Themida!, así es… la misma versión, recordé que tenía el script en el pendrive, rápidamente lo abrí con el Olly y en 10 min tenía todos elementales, para que el script hecho para este tute, funcionara para el malware, cambie todo, ejecuté el script… y que creen? Funcionó!!!, yo tardé más de dos semanas en poder hacerle unpackear este themida, entre webin y analizando en serio, cosa que ahorré mucho, pero mucho tiempo, tan solo 10 a 15min malware desprotegido y totalmente vulnerable en mis manos para análisis.

Muchos me animaron a escribir este tute, gracias Ema, Apo (trolleandome todos los días), Eddy, Elix, y a la gente que me dijo amanécete y escríbelo, jaja.

Se adjunta en la descarga,

ScriptThemida v2.1.8.txt
scriptlog.txt
Logeo.txt
UnpackMe.exe (empacado)
Unpackme_dump_fixed.exe (desempacado + fix)
Y este Tute!

Sábado, 17 de Marzo de 2012
Nox.