

UNPACKME



CONCURSO 2.

OBJETIVO: Unpacking!

PACKER: Themida v2.1.8

HERRAMIENTAS: (OllyDBG v1.10, y PlugIns), ImportRec.

PLUGÍNS FUNDAMENTALES:

CommandBar v3.00.108

StrongOD v0.4.5.810

OllyDump v3.00.110

FOR: **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 en serio a analizar el packer, fue tanto lo que quise lograr, hasta que el propio packer 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.

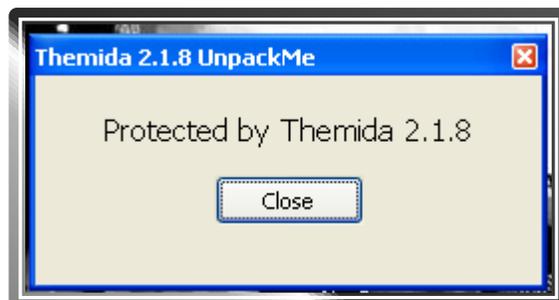
Ya que he visto bastante cosas, no especificaré todo los movimientos del OllyDBG que haga, es más, habrá lugares que sólo señalaré y diré el como llegar, más no mostraré la transición de llegada.

*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

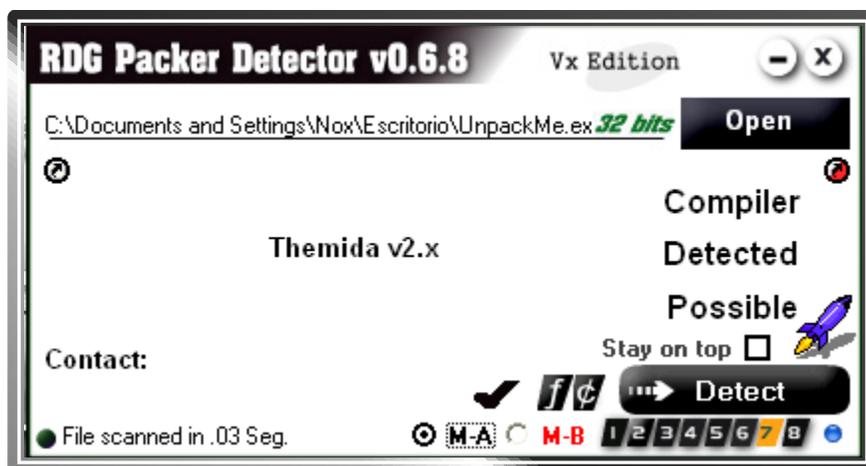
Ejecutamos.

Tarda algo de 5 segundos 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.

```
-= [ 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\UnpackMe.exe  
File Type : 32-Bit Exe (Subsystem : Win GUI / 2), Size : 2265600 (0229200h) Byte(s)  
[File Heuristics] -> Flag : 00000000000001001100000000110011 (0x0004C033)  
[!] Themida v2.0.1.0 - v2.1.8.0 (or newer) detected!  
[i] Hide PE Scanner Option used  
[CompilerDetect] -> Delphi 2007 (CodeGear RAD Studio 5.0)  
- Scan Took : 0.296 Second(s) [000000128h tick(s)]
```

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 ;).

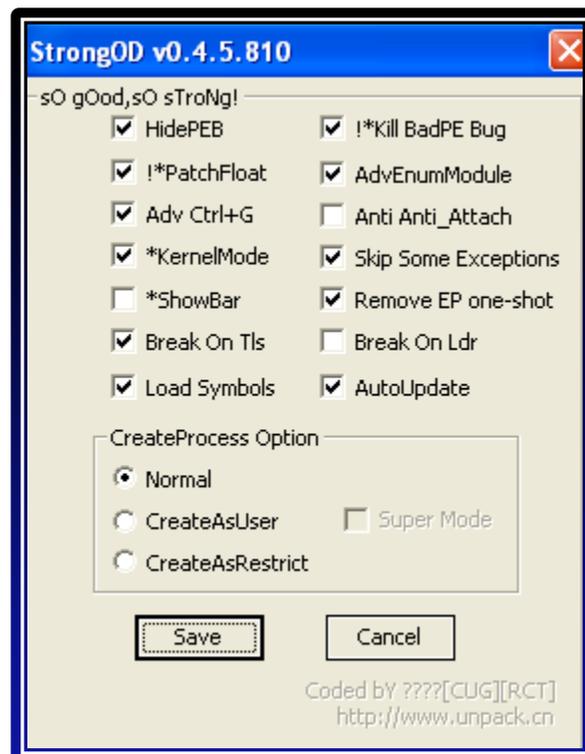
Esto se me ocurrió ahorita al escribir estas líneas, a las 11:40pm, abrir el RDG y usar su OEP detector, si se me hubiera ocurrido al principio, no hubiera sufrido con el OEP, (si, lo sé soy muy novato no?).



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...

Address	Hex dump	Disassembly
0085E000	83EC 04	SUB ESP, 4
0085E003	50	PUSH EAX
0085E004	53	PUSH EBX
0085E005	E8 01000000	CALL UnpackMe.0085E00B
0085E00A	CC	INT3
0085E00B	58	POP EAX
0085E00C	89C3	MOV EBX, EBX

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.

Address	Value	Comment
0012F0E0	7E3BD45D	CALL to GetModuleHandleW from user32.7E3BD457
0012F0E4	0012F0EC	LpModule = "C:\WINDOWS\system32\IMM32.DLL"
0012F0E8	7E3F11B8	ASCII "CJ"
0012F0EC	003A0043	
0012F0F0	0057005C	UnpackMe.0057005C
0012F0F4	004E0049	UnpackMe.004E0049
0012F0F8	004F0044	UnpackMe.004F0044
0012F0FC	00530057	UnpackMe.00530057
0012F100	0073005C	UnpackMe.0073005C
0012F104	00730079	UnpackMe.00730079
0012F108	00650074	UnpackMe.00650074
0012F10C	00330020	

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

Seguimos apretando F9, hasta que lleguemos aquí →

Address	Value	Comment
0012FF10	7C80B760	CALL to GetModuleHandleW from kernel32.7C80B75B
0012FF14	7FFDFC00	lpModule = "kernel32.dll"
0012FF18	0012FF38	
0012FF1C	00400137	RETURN to UnpackMe.00400137 from UnpackMe.00406828
0012FF20	00400164	ASCII "kernel32.dll"
0012FF24	00000008	
0012FF28	00457108	RETURN to UnpackMe.00457108 from UnpackMe.0040012C
0012FF2C	0012FF40	Pointer to next SEH record
0012FF30	0045711E	SE handler

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

Address	Value	Comment
0012FF1C	00400137	RETURN to UnpackMe.00400137 from UnpackMe.00406828
0012FF20	00400164	ASCII "kernel32.dll"

Por fin un retorno correcto, pasamos el RETN 4.

0040012C	53	PUSH EBX	
0040012D	68 64D14000	PUSH UnpackMe.00400164	ASCII "kernel32.dll"
00400132	E8 F196FFFF	CALL UnpackMe.00406828	
00400137	8BD8	MOV EBX, EAX	kernel32.7C800000
00400139	85DB	TEST EBX, EBX	
0040013B	74 10	JE SHORT UnpackMe.00400140	
0040013D	68 74D14000	PUSH UnpackMe.00400174	ASCII "GetDiskFreeSpaceExA"
00400142	53	PUSH EBX	
00400143	E8 E896FFFF	CALL UnpackMe.00406830	
00400148	A3 1C884500	MOV DWORD PTR DS:[45881C], EAX	
0040014D	833D 1C884500	CMP DWORD PTR DS:[45881C], 0	
00400154	75 0A	JNZ SHORT UnpackMe.00400160	
00400156	B8 548B4000	MOV EAX, UnpackMe.00408B54	
0040015B	A3 1C884500	MOV DWORD PTR DS:[45881C], EAX	
00400160	5B	POP EBX	
00400161	C3	RETN	

Pues... alguien sabe dónde estamos?

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

Address	Hex dump	Disassembly
00408200	55	PUSH EBP
00408201	8BEC	MOV EBP, ESP
0040820F	83C4 F0	ADD ESP, -10
00408212	B8 C4814000	MOV EAX, Keygen.004081C4
00408217	E8 F0C2FFFF	CALL Keygen.0040450C
0040821C	6A 40	PUSH 40

EP de un Delphi

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!).

Address	Value	Comment
0012FF54	7C80B760	CALL to GetModuleHandleW from kernel
0012FF58	7FFDDC00	pModule = "kernel32.dll"
0012FF5C	0012FF7C	
0012FF60	004078E3	RETURN to Keygen.004078E3 from <JMP.>
0012FF64	00407910	ASCII "kernel32.dll"
0012FF68	00000007	
0012FF6C	00407DE9	RETURN to Keygen.00407DE9 from Keygen
0012FF70	0012FF84	Pointer to next SEH record
0012FF74	00407DFC	SE handler

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

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

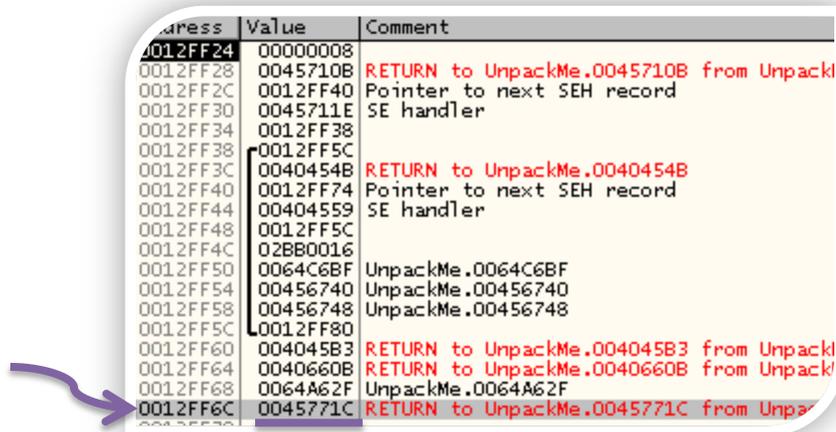
004078D8	53	PUSH EBX	
004078D9	68 10794000	PUSH Keygen.00407910	pModule = "kernel32.dll"
004078DE	E8 0DC0FFFF	CALL <JMP.&kernel32.GetModuleHandleA>	GetModuleHandleA
004078E3	8B08	MOV EBX, EAX	kernel32.7C800000
004078E5	850B	TEST EBX, EBX	
004078E7	74 10	JE SHORT Keygen.004078F9	
004078E9	68 20794000	PUSH Keygen.00407920	ProcNameOrOrdinal = "GetDiskFreeSpaceExA"
004078EE	53	PUSH EBX	hModule
004078EF	E8 04C0FFFF	CALL <JMP.&kernel32.GetProcAddress>	GetProcAddress
004078F4	A3 E8904000	MOV DWORD PTR DS:[4090E8], EAX	
004078F9	> 833D E8904000	CMP DWORD PTR DS:[4090E8], 0	
00407900	> 75 0A	JNZ SHORT Keygen.0040790C	
00407902	B8 FC554000	MOV EAX, Keygen.004055FC	
00407907	A3 E8904000	MOV DWORD PTR DS:[4090E8], EAX	
0040790C	> 5B	POP EBX	
0040790D	> C3	RETN	

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

Address	Value	Comment
0012FF68	00000007	
0012FF6C	00407DE9	RETURN to Keygen.00407DE9 f
0012FF70	0012FF84	Pointer to next SEH record
0012FF74	00407DFC	SE handler
0012FF78	0012FF7C	
0012FF7C	0012FF9C	
0012FF80	00403338	RETURN to Keygen.00403338
0012FF84	0012FFB4	Pointer to next SEH record
0012FF88	00403346	SE handler
0012FF8C	0012FF9C	
0012FF90	0012CE6C	
0012FF94	0012D41C	
0012FF98	004081C4	Keygen.004081C4
0012FF9C	0012FFC0	
0012FFA0	0040339F	RETURN to Keygen.0040339F f
0012FFA4	0040454B	RETURN to Keygen.0040454B f
0012FFA8	7FFDE000	
0012FFAC	0040821C	RETURN to Keygen.0040821C f
0012FFB0	7C91DCBA	RETURN to ntdll.7C91DCBA

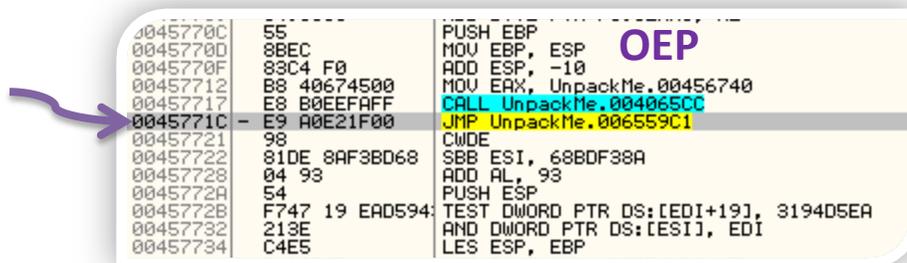
El RETURN es **0040821Ch**.

Regresamos al UnpackMe.exe, y miremos la pila.



Address	Value	Comment
0012FF24	00000008	
0012FF28	0045710B	RETURN to UnpackMe.0045710B from UnpackMe.0040821C
0012FF2C	0012FF40	Pointer to next SEH record
0012FF30	0045711E	SE handler
0012FF34	0012FF38	
0012FF38	0012FF5C	
0012FF3C	0040454B	RETURN to UnpackMe.0040454B from UnpackMe.0040821C
0012FF40	0012FF74	Pointer to next SEH record
0012FF44	00404559	SE handler
0012FF48	0012FF5C	
0012FF4C	02BB0016	
0012FF50	0064C6BF	UnpackMe.0064C6BF
0012FF54	00456740	UnpackMe.00456740
0012FF58	00456748	UnpackMe.00456748
0012FF5C	0012FF80	
0012FF60	004045B3	RETURN to UnpackMe.004045B3 from UnpackMe.0040821C
0012FF64	0040660B	RETURN to UnpackMe.0040660B from UnpackMe.0040821C
0012FF68	0064A62F	UnpackMe.0064A62F
0012FF6C	0045771C	RETURN to UnpackMe.0045771C from UnpackMe.0040821C

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



Address	Disassembly	Comment
0045770C	55	PUSH EBP
0045770D	8BEC	MOV EBP, ESP
0045770E	83C4 F0	ADD ESP, -10
00457710	B8 40674500	MOV EAX, UnpackMe.00456740
00457712	E8 B0EEFAFF	CALL UnpackMe.004065CC
00457714	E9 A0E21F00	JMP UnpackMe.006559C1
00457716	98	CWDE
00457718	81DE 8AF3BD68	SBB ESI, 68BDF38A
0045771A	04 93	ADD AL, 93
0045771C	54	PUSH ESP
0045771E	F747 19 EAD594	TEST DWORD PTR DS:[EDI+19], 3194D5EA
00457720	213E	AND DWORD PTR DS:[ESI], EDI
00457722	C4E5	LES ESP, EBP

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	CWD	
00457722	81DE 8AF3BD68	SBB ESI, 68BDF38A	

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

Intermodular Calls:

0045212C	CALL UnpackMe.00406000	
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	

00420E5A	CALL UnpackMe.00406958	gdi32.CreateDIBitmap
00420F11	CALL UnpackMe.00406958	gdi32.CreateDIBitmap
00420F6F	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
00421406	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.00406C88	
00422478	CALL UnpackMe.00406A88	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.

```

004011E0 . 11          DB 11
004011E1 . 54 49 6E 74 ASCII "TInterfacedObjec"
004011F1 . 74          ASCII "t"
004011F2 . 8BC0       MOV EAX, EAX
004011F4 $- E9 A7FB8A02 JMP 02CB0DA0
004011F9 . 90         NOP
004011FA . 8BC0       MOV EAX, EAX
004011FC . 90         NOP
004011FD .- E9 98F68A02 JMP 02CB089A
00401202 . 8BC0       MOV EAX, EAX
00401204 .- E9 F7ED8A02 JMP 02CB0000

```

Inicio_1: 004011F4h

```

004012EC $ 90         NOP
004012ED .- E9 8E35CF76 JMP oleaut32.SysFreeString
004012F2 . 8BC0       MOV EAX, EAX
004012F4 $- E9 C6F88002 JMP 02C10BBF
004012F9 . 90         NOP
004012FA . 8BC0       MOV EAX, EAX
004012FC $- E9 2CF88002 JMP 02C10B20
00401301 . 90         NOP
00401302 . 8BC0       MOV EAX, EAX
00401304 $- E9 0EF88002 JMP 02C10B17
00401309 . 90         NOP
0040130A . 8BC0       MOV EAX, EAX
0040130C $ 53        PUSH EBX

```

Final_1: 00401309h

```

0040132D L C3        RETN
0040132E . 8BC0       MOV EAX, EAX
00401330 $- E9 21F68002 JMP 02C10956
00401335 . 90         NOP
00401336 . 8BC0       MOV EAX, EAX
00401338 $- E9 28F48002 JMP 02C10765
0040133D . 90         NOP
0040133E . 8BC0       MOV EAX, EAX
00401340 $- E9 23F38002 JMP 02C10668
00401345 . 90         NOP
00401346 . 8BC0       MOV EAX, EAX
00401348 $- E9 3AF37F02 JMP 02C00687
0040134D . 90         NOP
0040134E . 8BC0       MOV EAX, EAX
00401350 . FFFFFFFF DD FFFFFFFF
00401354 . 57000000 DD 00000057

```

Inicio_2: 00401330h

Final_2: 0040134Dh

004037C1	: 6211	BOUND EDX, 0WORD
004037C3	: 35 AECA7BC3	XOR EAX, C37BCAE
004037C8	:- E9 7ECC7F02	JMP 02CC0044B
004037CD	: 90	NOP
004037CE	: 8BC0	MOV EAX, EAX
004037D0	: 53	PUSH FRX

Inicio_3: 004037C8h

Final_3: 004037CDh

00406504	: C3	RETN
00406505	: 8D40 00	LEA EAX, DWORD PTR DS:[EAX]
00406508	:- E9 09A48B02	JMP 02CC0916
0040650D	: 90	NOP
0040650E	: 8BC0	MOV EAX, EAX
00406510	:- E9 239F8B02	JMP 02CC0438
00406515	: 90	NOP
00406516	: 8BC0	MOV EAX, EAX
00406518	:- E9 CC9D8B02	JMP 02CC02E9
0040651D	: 90	NOP
0040651E	: 8BC0	MOV EAX, EAX
00406520	:- E9 DB9A8B02	JMP 02CC0000
00406525	: 90	NOP
00406526	: 8BC0	MOV EAX, EAX

Inicio_4: 00406508h

Final_4: 00406525h

00406742	: 7C664000	DD UnpackMe.0040667C
00406746	: 8BC0	MOV EAX, EAX
00406748	:- E9 DA9ABE02	JMP 02FF0227
0040674D	: 90	NOP
0040674E	: 8BC0	MOV EAX, EAX
00406750	:- E9 AB98BE02	JMP 02FF0000
00406755	: 90	NOP
00406756	: 8BC0	MOV EAX, EAX
00406758	:- E9 A398BD02	JMP 02FE0000
0040675D	: 90	NOP
0040675E	: 8BC0	MOV EAX, EAX
00406760	:- E9 9B98BC02	JMP 02FD0000

Inicio_5: 00406748h

```

00406FF0  $- E9 0B908C02  JMP 02CD0000
00406FF5      90             NOP
00406FF6      8BC0          MOV EAX, EAX
00406FF8  $- E9 C09D8B02  JMP 02CC0DB0
00406FFD      90             NOP
00406FFE      8BC0          MOV EAX, EAX
00407000  $- E9 799D8B02  JMP 02CC0D7E
00407005      90             NOP
00407006      8BC0          MOV EAX, EAX
00407008  $- E9 669A8B02  JMP 02CC0A73
0040700D      90             NOP
0040700E      8BC0          MOV EAX, EAX
00407010  r$ 55           PUSH EBP

```

Final_5: 0040700Dh

```

0040D02A  . 5B           POP EBX
0040D02B  . C3           RETN
0040D02C  $- E9 5737BE02  JMP 02FF0788
0040D031      90             NOP
0040D032      8BC0          MOV EAX, EAX

```

Inicio_6: 0040D02Ch

Final_6: 0040D0231h

```

0040DBA1  $ 90             NOP
0040DBA4  $- E9 A66DCE76  JMP oleaut32.VariantInit
0040DBAA      8BC0          MOV EAX, EAX
0040DBAC      90             NOP
0040DBAD  $- E9 3E6DCE76  JMP oleaut32.VariantClear
0040DBB2      8BC0          MOV EAX, EAX
0040DBB4  $- E9 4471CE76  JMP oleaut32.VariantCopy
0040DBB9      E3           DB E3
0040DBBA      8BC0          MOV EAX, EAX
0040DBBC      90             NOP
0040DBBD  $- E9 F98FCE76  JMP oleaut32.VariantChangeType
0040DBC2      8BC0          MOV EAX, EAX
0040DBC4  r. 55           PUSH EBP

```

Inicio_7: 0040DBA4h

Final_7: 0040DBC1h

0040E00B	90	NOP
0040E00C	\$- E9 44CACE76	JMP oleaut32.SafeArrayCreate
0040E011	74	DB 74
0040E012	8BC0	MOV EAX, EAX
0040E014	90	NOP
0040E015	.- E9 8C71CE76	JMP oleaut32.SafeArrayGetLBound
0040E01A	8BC0	MOV EAX, EAX
0040E01C	90	NOP
0040E01D	.- E9 3871CE76	JMP oleaut32.SafeArrayGetUBound
0040E022	8BC0	MOV EAX, EAX
0040E024	90	NOP
0040E025	.- E9 E6CACE76	JMP oleaut32.SafeArrayPtrOfIndex
0040E02A	8BC0	MOV EAX, EAX

Inicio_8: 0040E00Bh

Final_8: 0040E029h

0041CE7C	90	NOP
0041CE7D	.- E9 E4C4F976	JMP comctl32.ImageList_Create
0041CE82	8BC0	MOV EAX, EAX
0041CE84	90	NOP
0041CE85	.- E9 9582F976	JMP comctl32.ImageList_Destroy
0041CE8A	8BC0	MOV EAX, EAX
0041CE8C	\$- E9 CA82F976	JMP comctl32.ImageList_GetImageCount
0041CE91	D4	DB D4
0041CE92	8BC0	MOV EAX, EAX
0041CE94	90	NOP
0041CE95	.- E9 4783F976	JMP comctl32.ImageList_Add
0041CE9A	8BC0	MOV EAX, EAX
0041CE9C	\$- E9 CE83F976	JMP comctl32.ImageList_SetBkColor
0041CEA1	41	INC ECX
0041CEA2	8BC0	MOV EAX, EAX

Inicio_9: 0041CE7Ch

0041CF0A	8BC0	MOV EAX, EAX
0041CF0C	\$- E9 A287F976	JMP comctl32.ImageList_SetIconSize
0041CF11	0C	DB 0C
0041CF12	8BC0	MOV EAX, EAX
0041CF14	90	NOP
0041CF15	.- E9 B388F876	JMP comctl32._TrackMouseEvent
0041CF1A	8BC0	MOV EAX, EAX
0041CF1C	E4C74580	DB 0041CF1C

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 04 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 07 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	000000<A000000<P
00401204	90 90 90 90 90 90 8B C0 90 90 90 90 E9 1C 0C 01	000000<A000000<.c
00401214	C8 80 CA F7 D9 79 13 DF 90 DE E6 6C FB 8C 30 7C	EëE=Uydß0pælú00l
00401224	11 08 3C E8 83 0A C1 10 78 ED 13 31 16 8B 44 9D	00<èf.Aoxi0l0<Dc
00401234	2C 89 20 84 30 46 04 42 1B 60 4C 08 78 F5 8B CE	,% „OF0B0`L0x0<I
00401244	26 89 72 09 86 90 10 80 10 39 7C D1 F7 24 52 74	&W.r.t00009 N+&Rt
00401254	60 02 20 81 E9 A4 A7 70 45 FE CA 78 C1 02 03 0F	`o de&spEbExA00c
00401264	B6 D6 B8 FE 68 0B D3 C0 0C 21 04 95 24 2D 75 54	¶0 ,bho0A.!o.\$-ut
00401274	ED 10 89 63 D1 25 05 20 49 23 4C 07 81 EA 30 0B	ã0%cN%o I#L00èP

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

Quitamos 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.

0040129E	90	NOP
0040129F	90	NOP
004012A0	90	NOP
004012A1	90	NOP
004012A2	8BC0	MOV EAX, EAX
004012A4	90	NOP
004012A5	90	NOP
004012A6	90	NOP
004012A7	90	NOP
004012A8	90	NOP
004012A9	90	NOP
004012AA	8BC0	MOV EAX, EAX
004012AC	90	NOP
004012AD	90	NOP
004012AE	90	NOP
004012AF	90	NOP
004012B0	90	NOP
004012B1	90	NOP
004012B2	8BC0	MOV EAX, EAX
004012B4	90	NOP
004012B5	90	NOP
004012B6	90	NOP
004012B7	90	NOP
004012B8	90	NOP
004012B9	90	NOP
004012BA	8BC0	MOV EAX, EAX
004012BC	90	NOP
004012BD	90	NOP
004012BE	90	NOP
004012BF	90	NOP
004012C0	90	NOP
004012C1	90	NOP
004012C2	8BC0	MOV EAX, EAX
004012C4	90	NOP
004012C5	90	NOP
004012C6	90	NOP
004012C7	90	NOP
004012C8	90	NOP
004012C9	90	NOP
004012CA	8BC0	MOV EAX, EAX
004012CC	90	NOP
004012CD	90	NOP
004012CE	90	NOP
004012CF	90	NOP
004012D0	90	NOP
004012D1	90	NOP
004012D2	8BC0	MOV EAX, EAX
004012D4	90	NOP

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.

0064B53E	A4	MOV BYTE PTR ES:[EDI], BYTE PTR DS:[ESI]	
0064B53F	AA	STOS BYTE PTR ES:[EDI]	Escribe E9 = JMP
0064B540	60	PUSHAD	
0064B541	E9 11000000	JMP UnpackMe.0064B557	
0064B546	B7 06	MOV EB, 6	

Los registros.

```
Registers (FPU)
EAX 770F48E9 oleaut32.770F48E9
ECX 770F4880 oleaut32.SysFreeString
EDX 341E8004
EBX 00000000
ESP 0012FF70
EBP F5C90014
ESI 00637168 UnpackMe.00637168
EDI 004012EC UnpackMe.004012EC
EIP 0064B53F UnpackMe.0064B53F
```

EDI en Disassembler.

```
004012E9  90      NOP
004012EA  8BC0   MOV EAX, EAX
004012EC  90      NOP
004012ED  90      NOP
004012FE  90      NOP
```

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

F7 Step Into, veamos como queda.

```
004012EA  8BC0   MOV EAX, EAX
004012EC  E9 90909090 JMP 9000A381
004012F1  90      NOP
004012F2  8BC0   MOV EAX, EAX
```

Otra vez F9 RUN, y llegamos aquí.

```
0064B5BC  5A      POP EDX
0064B5BD  884F 04  MOV BYTE PTR DS:[EDI+4], CL
0064B5C0  59      POP ECX
CL=80
DS:[004012F1]=90
```

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

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

0064B5BD	884F 04	MOV BYTE PTR DS:[EDI+4], CL	
0064B5C0	59	POP ECX	
0064B5C1	0F83 07000000	JNB UnpackMe.0064B5CE	
0064B5C7	0F87 01000000	JA UnpackMe.0064B5CE	
0064B5CD	F5	CMC	
0064B5CE	8B85 54A8970A	MOV EAX, DWORD PTR SS:[EBP+A97A854]	EAX, ECX = Funcion correcta
0064B5D4	60	PUSHAD	
0064B5D5	81D3 9B321941	ADC EBX, 4119329B	
0064B5DB	56	PUSH ESI	
0064B5DC	BB 94146477	MOV EBX, 77641494	
0064B5E1	59	POP ECX	
0064B5E2	61	POPAD	
0064B5E3	52	PUSH EDX	TERMINA EL NO HACER NADA
0064B5E4	BA 592ADB3F	MOV EDX, 3FDB2A59	<- Misma constante
0064B5E9	29D0	SUB EAX, EDX	
0064B5EB	5A	POP EDX	
0064B5EC	29F8	SUB EAX, EDI	EDI = Bytes del desplazamiento
0064B5EE	05 592ADB3F	ADD EAX, 3FDB2A59	<- Misma Constante
0064B5F3	F5	CMC	
0064B5F4	57	PUSH EDI	
0064B5F5	BF 04000000	MOV EDI, 4	
0064B5FA	2D A64EE763	SUB EAX, 63E74EA6	<- Misma constante
0064B5FF	29F8	SUB EAX, EDI	Resta 4 bytes del JMP (No 5, por la posicion)
0064B601	05 A64EE763	ADD EAX, 63E74EA6	<- Misma constante
0064B606	5F	POP EDI	
0064B607	60	PUSHAD	
0064B608	66:BF 7046	MOV DI, 4670	
0064B60C	60	PUSHAD	EAX = OFFSET
0064B60D	FC	CLD	a la funcion (API emulada) (seccion del packer)
0064B60E	E9 06000000	JMP UnpackMe.0064B619	

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.

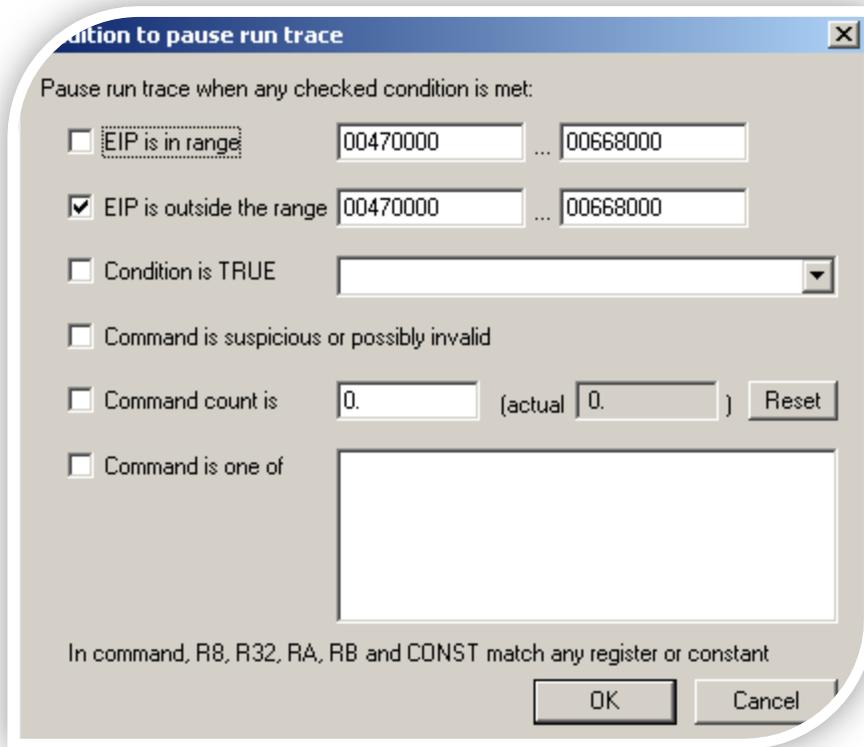
Damos RUN nuevamente.

Address	Hex dump	Disassembly	Comment
0064B61B	AB	STOS DWORD PTR ES:[EDI]	Escribe la funcion
0064B61C	E9 12000000	JMP UnpackMe.0064B633	

Registers (FPU)	
EAX	76CF358F
ECX	770F4880 oleaut32.SysFreeString
EDX	341E8004
EBX	00000000
ESP	0012FF70
EBP	F5C90014
ESI	00637168 UnpackMe.00637168
EDI	004012ED UnpackMe.004012ED
EIP	0064B61B UnpackMe.0064B61B

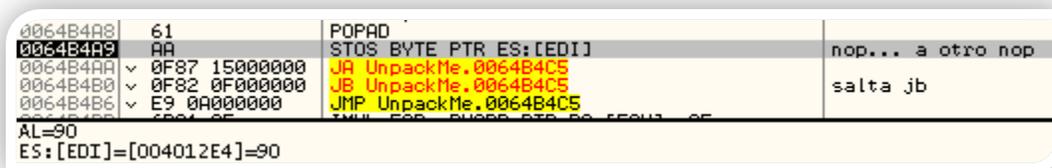
Como había mencionado, en EAX, tendríamos el desplazamiento, será escrita, y así pasará a escribir el otro JMP, hasta completarlos.

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.

004012E1	90	NOP
004012E2	8BC0	MOV EAX, EAX
004012E4	90	NOP
004012E5	90	NOP
004012E6	90	NOP
004012E7	90	NOP
004012E8	90	NOP
004012E9	90	NOP
004012EA	8BC0	MOV EAX, EAX
004012EC	E9 8F35CF76	JMP oleaut32.SysFreeString
004012F1	808B C0909090	OR BYTE PTR DS:[EBX+909090C0], 90
004012F8	90	NOP

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

0064B4D4	0FBFCB	MOVSX ECX, BX	
0064B4D7	61	POPAD	
0064B4D8	AA	STOS BYTE PTR ES:[EDI]	Se escribe el E9 -> JMP
0064B4D9	0F8F 17000000	JG UnpackMe.0064B4F6	salta jg
0064B4DF	E9 12000000	JMP UnpackMe.0064B4F6	
0064B4F4	00945F 60F65F01	OR BYTE PTR DS:[ESI+FRY*2+0765F601], 01	

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

Address	Hex dump	Disassembly	Comment
0064B61B	AB	STOS DWORD PTR ES:[EDI]	Escribe el offset
0064B61C	E9 12000000	JMP UnpackMe.0064B633	
0064B621	00EB	ADD BL, CH	
0064B623	87E8	XCHG EAX, EBP	

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

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

006485E1	E9 09000000	JMP UnpackMe.006485F0			EAX 770FA3EC oleaut32.SysReAllocStringLen
006485F3	36:60	PUSHAD		Superfluous I/O command	ECX 770F0000 oleaut32.770F0000
006485F5	ED	IN EAX, DX			EDX 341E8004
006485F6	0089 2CD588FA	ADD BYTE PTR DS:[ECX+FA88D52C], CL			EBX 029A0000
006485FC	2D 3A461502	SUB EAX, 215463A			ESP 0012FF6C
00648601	2D 2C6FEE30	SUB EAX, 30EE6F2C			EBP F5C90014
00648606	01C8	ADD EAX, ECX		<-----	ESI 770F219C oleaut32.770F219C
00648608	05 2C6FEE30	ADD EAX, 30EE6F2C			EDI 004012F1 UnpackMe.004012F1
0064860D	05 3A461502	ADD EAX, 215463A			EIP 00648612 UnpackMe.00648612
00648612	60	PUSHAD			
00648613	E9 0E000000	JMP UnpackMe.00648626			

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 librería 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, ejecuciones de instrucciones de más, en fin...

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

Address	Hex dump	Disassembly	Comment
00648881	FF3424	PUSH DWORD PTR SS:[ESP]	UnpackMe.0063715C
00648884	5E	POP ESI	
00648885	81C4 04000000	ADD ESP, 4	
00648888	60	PUSHAD	

Registers (FPU)		
EAX	770F4880	oleaut32.SysFreeString
ECX	770F0000	oleaut32.770F0000
EDX	341E8004	
EBX	029A0000	
ESP	0012FF70	
EBP	F5C90014	
ESI	0063715C	UnpackMe.0063715C
EDI	000007E9	
EIP	0064888B	UnpackMe.0064888B

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!

006488DC	F8	CLC	
006488DD	8B90 D1077E0A	MOV EBX, DWORD PTR SS:[EBP+A7E07D1]	EBX = Kernel32
006488E3	0F88 01000000	JNS UnpackMe.006488EA	salta
006488E9	F5	CMC	
006488EA	FC	CLD	
006488EB	29CB	SUB EBX, ECX	Compara ECX (Current Library) con Kernel32
006488ED	0F84 EC000000	JBE UnpackMe.006489DF	Si la resta es 0, es porque eran las misma librerias...
006488F3	60	PUSHAD	
006488F4	F9 11000000	JMP UnpackMe.00648900	

SS:[004707E5]=7C800000 (kernel32.7C800000)

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.

```

00648914  F8          CLC
00648915  8B9D A11A7E0A MOV EBX, DWORD PTR SS:[EBP+A7E1AA1] EBX = User32
0064891B  60          PUSHAD
0064891C  E8 0E000000 CALL UnpackMe.0064892F
00648921  5C          POP ESP
00648922  37          JNB
SS:[00471AB5]=7E390000 (user32.7E390000)
EBX=05710000

```

Ahora EBX tendrá el handle de User32.

```

00648935  B1          JNB
00648936  29CB       SUB EBX, ECX
00648938  0F84 A1000000 JE UnpackMe.006489DF
0064893E  60          PUSHAD
ECX=770F0000 (oleaut32.770F0000)
EBX=7E390000 (user32.7E390000)

```

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

El turno de Advapi32.

```

0064894F  8B9D D1187E0A MOV EBX, DWORD PTR SS:[EBP+A7E18D1] EBX = Advapi32
00648955  60          PUSHAD
00648956  66 81C9 93E8 DB CX, 0E893
SS:[004718E5]=77DA0000 (advapi32.77DA0000)
EBX=072A0000

```

```

00648973  A0 53585F61 MOV AL, BYTE PTR DS:[615F5853]
00648978  29CB       SUB EBX, ECX
0064897A  0F84 5F000000 JE UnpackMe.0064899F
00648980  0F8D 0A000000 JGE UnpackMe.00648990
00648986  60          PUSHAD
ECX=770F0000 (oleaut32.770F0000)
EBX=77DA0000 (advapi32.77DA0000)

```

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 librería, 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 específicas 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

```

0064898B 66:B8 2BBB MOV AX, 0BB2B
0064898F 61 POPAD
00648990 8D9D EEB79B0A LEA EBX, DWORD PTR SS:[EBP+A9BB7EE]
00648996 E9 06000000 JMP UnpackMe.006489A1
0064899B 51 PUSH ECX
Address=0064B802
EBX=00CB0000

```

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

Address	Hex dump	Disassembly
006489A1	FFD3	CALL EBX

Siendo EBX = **0064B802h**

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

```

006489A3 F8 CALL
006489A4 81EC 04000000 SUB ESP, 4
006489AA 890424 MOV DWORD PTR SS:[ESP], EAX [ESP] = Current API
006489AD 5F POP EDI
006489AE 0F8E 01000000 JLE UnpackMe.006489B5

```

EDI = Actual API.

```

Registers (FPU)
EAX 770F4880 oleaut32.SysFreeString
ECX 770F0000 oleaut32.770F0000
EDX 341E8004
EBX 0064B802 UnpackMe.0064B802
ESP 0012FF70
EBP F5C90014
ESI 0063715C UnpackMe.0063715C
EDI 770F4880 oleaut32.SysFreeString
EIP 006489B5 UnpackMe.006489B5

```

```

006489B4  F8          CLC
006489B5  8985 54A8970A  MOV DWORD PTR SS:[EBP+A97A854], EAX  EBP + Offset = Buffer Of API
006489BB  F5          CMC
006489BC  F9 14000000   JMP UnpackMe.006489DE
EAX=770F4880 (oleaut32.SysFreeString)
SS:[0060A868]=0A832E6F

```

[0060A868h] = Actual API.

```

0064A0BB  61          POPAD
0064A0BC  AD         LODS DWORD PTR DS:[ESI]             ESI !!!
0064A0BD  E9 05000000  JMP UnpackMe.0064A0D7
0064A0C2  DF58 03     FICOM WORD PTR DS:[ESI+3]
DS:[ESI]=[0063715C]=9F9242AA

```

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

Address	Hex dump
0063715C	AA 42 92 9F 5D 02 00 80 AA AA AA AA FF FF FF FF
0063716C	DD DD DD DD 6B 23 30 68 AA 42 92 BF 5C 02 00 80
0063717C	AA AA AA AA FF FF FF FF DD DD DD DD 0F 19 18 43
0063718C	AA 42 92 0F 5B 02 00 80 AA AA AA AA FF FF FF FF
0063719C	DD DD DD DD EE EE EE EE DD DD DD DD 79 31 1E A0
006371AC	73 30 80 E4 98 28 F1 E9 AA AA AA AA FF FF FF FF
006371BC	DD DD DD DD 38 34 E4 FF 72 30 80 04 99 28 F1 A9
006371CC	AA AA AA AA FF FF FF FF DD DD DD DD 76 79 4C 40
006371DC	72 30 80 24 99 28 F1 69 AA AA AA AA FF FF FF FF
006371EC	DD DD DD DD EE EE EE EE DD DD DD DD 39 2E 3F C9
006371FC	72 30 80 64 C0 27 F1 09 AA AA AA AA FF FF FF FF
0063720C	DD DD DD DD 05 AB 8A 39 72 30 80 84 A4 28 F1 09
0063721C	AA AA AA AA FF FF FF FF DD DD DD DD 35 25 A8 4A
0063722C	72 30 80 A4 9A 28 F1 69 AA AA AA AA FF FF FF FF
0063723C	DD DD DD DD 55 8A C2 47 72 30 80 C4 9D 28 F1 69
0063724C	AA AA AA AA FF FF FF FF DD DD DD DD 03 D3 9D 7F
0063725C	72 30 80 E4 9E 28 F1 A9 AA AA AA AA FF FF FF FF

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 reedireccionada 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].

address	Hex dump	Disassembly	Comment
0064ADC7	F5	CMC	
0064ADC8	55	PUSH EBP	
0064ADC9	BD 5760AC10	MOV EBP, 10AC6057	
0064ADCE	C746 FC A22A05	MOV DWORD PTR DS:[ESI-4], 6A052AA2	EL ESI de arriba "ESI!!!" es Xoreado por aqu
0064ADD5	316E FC	XOR DWORD PTR DS:[ESI-4], EBP	
0064ADD8	5D	POP EBP	
0064ADD9	816E FC F54AA9	SUB DWORD PTR DS:[ESI-4], 7AA94AF5	El Resultado de las operaciones anteriores es = 7AA94AF5
0064ADE0	60	PUSHAD	
0064ADE1	E9 0B000000	JMP UnpackMe.0064ADF1	

Este es el lugar donde borran los datos la tabla.

Address	Hex dump	Disassembly	Comment
0064AE36	80BD E6489B0A	CMP BYTE PTR SS:[EBP+A9B48E6], 0	FLAG 1 - Setear a 0
0064AE3D	0F84 65000000	JE UnpackMe.0064AE88	SALTA

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...

```

0064AE82  v E9 58000000 JMP UnpackMe.0064AEFF
0064AE87  F9 STC
0064AE88  C1C0 05 ROL EAX, 5
0064AE8B  v 0F85 03000000 JNZ UnpackMe.0064AEB9
0064AE8E  v 0F88 02000000 JS UnpackMe.0064AEB9
0064AE97  60 PUSHAD
0064AE98  61 POPAD
0064AE99  F5 CMC
0064AE9A  55 PUSH EBP
0064AE9B  BD A475A271 MOV EBP, 71A275A4
0064AE9C  C1E5 07 SHL EBP, 7
0064AE9D  F7D5 NOT EBP
0064AE9E  53 PUSH EBX
0064AE9F  BB B2564D2A MOV EBX, 2A4D56B2
0064AEE0  43 INC EBX
0064AEE1  D1E3 SHL EBX, 1
0064AEE2  F7D5 NEG EBX
0064AEE3  81EB FA03AF1F SUB EBX, 1FAF03FA
0064AEE4  81C3 9229D11B ADD EBX, 1BD12992
0064AEE5  81C3 54148079 ADD EBX, 79801454
0064AEE6  29D0 SUB EBP, EBX
0064AEE7  5B POP EBX
0064AEE8  01E8 ADD EAX, EBP
0064AEE9  5D POP EBP
0064AEEA  v E9 12000000 JMP UnpackMe.0064AEFF
0064AEEB  06 PUSH ES
0064AEEC  06

```

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.

RVA

```

Registers (FPU)
EAX 0005F6CC
ECX 00000002
EDX 341E8004
EBX 0064B802 UnpackMe.0064B802
ESP 0012FF70
EBP F5C90014
ESI 00637160 UnpackMe.00637160
EDI 770F4880 oleaut32.SysFreeString
EIP 0064AEE8 UnpackMe.0064AEE8

```

```

0064AEFF  4B DEC EBX
0064AF00  0385 2D167E0A ADD EAX, DWORD PTR SS:[EBP+A7E162D] UnpackMe.00400000
0064AF05  0F8A 09000000 JPE UnpackMe.0064AF14 salta
0064AF08  60 PUSHAD
SS:[00471641]=00400000 (UnpackMe.00400000), ASCII "M2P"
EAX=0005F6CC

```

Y como es RVA...

0064AF2E	66:B8 F194	MOV HX, 94+1	
0064AF32	61	POPAD	
0064AF33	8B8D 54A8970A	MOV ECX, DWORD PTR SS:[EBP+A97A854]	ECX = Current API
0064AF39	60	PUSHAD	
0064AF3A	E9 05000000	JMP UnpackMe.0064AF44	

ECX = API actual.

0064AF60	01C1	ADD ECX, EAX	
0064AF62	8939	MOV DWORD PTR DS:[ECX], EDI	Guarda la entrada correcta
0064AF64	59	POP ECX	
EDI=770F4880 (oleaut32.SysFreeString)			
DS:[0045F6CC]=00000000			

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

Address	Hex dump
0045F6CC	80 48 0F 77 00 00 00 00
0045F6DC	00 00 00 00 00 00 00 00
0045F6FC	00 00 00 00 00 00 00 00

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 →

```

Registers (FPU)
EAX 77DA7ABB advapi32.RegQueryValueExA
ECX 77DA0000 advapi32.77DA0000
EDX 341E8004
EBX 029F0004
ESP 0012FF6C
EBP F5C90014
ESI 77DA1E8C advapi32.77DA1E8C
EDI 004012E1 UnpackMe.004012E1
EIP 00648612 UnpackMe.00648612

```

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

```

00599F9E 0103 ADD EBX, EDX
00599FA0 8803 MOV BYTE PTR DS:[EBX], AL
00599FA2 5B POP EBX
AL=E9
DS:[004012C4]=90

```

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

```

00599720 ^ E9 44EEFFFF JMP UnpackMe.00598576
00599722 8F02 POP DWORD PTR DS:[EDX]
00599724 ^ E9 30FFFFFF JMP UnpackMe.00598576
Stack [0012FF64]=027DED37
DS:[004012C5]=90909090

```

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.

Address	Hex dump	Disassembly
00648612	60	PUSHAD
00648613	E9 0E000000	JMP UnpackMe.00648626
00648618	C8 3BFFBB	ENTER 0FF3B, 0BB

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 librería 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** Continúa**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.GetStorageHandle
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.UnhandledExceptionFilter
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	DB 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.lstrcpyA
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	
004012BC	FF25 E0F6450	JMP DWORD PTR DS:[45F6E0]	advapi32.RegOpenKeyExA

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  FF25 A79D9902 JMP 02DA0B04
00406D5D  90      NOP
00406D5F  8BC0      MOV EAX, EAX

```

```

00406D96  8BC0      MOV EAX, EAX
00406D98  FF25 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  FF25 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...

00406D55	90	NOP
00406D56	8BC0	MOV EAX, EAX
00406D58	\$- E9 F89E9A02	JMP 02DB0C55
00406D5D	90	NOP
00406D5E	8BC0	MOV EAX, EAX
Address	Hex dump	Disassembly
00406CF8	\$- E9 589F9A02	JMP 02DB0C55
00406CFD	90	NOP

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...

00406D8D	90	NOP
00406D8E	8BC0	MOV EAX, EAX
00406D90	\$- E9 DB9C9702	JMP 02D80A70
00406D95	90	NOP
00406D96	8BC0	MOV EAX, EAX
00406D98	\$- E9 D39C9702	JMP 02D80A70
00406D9D	90	NOP
00406D9E	8BC0	MOV EAX, EAX

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

Address	Hex dump	Disassembly
00406EE8	\$- E9 13918D02	JMP 02CE0000
00406EED	90	NOP
00406EEE	8BC0	MOV EAX, EAX
00406EF0	\$- E9 0B918D02	JMP 02CE0000
00406EFA	90	NOP
00406FF6	8BC0	MOV EAX, EAX

Igual que el anterior, ahora toca repararlos.

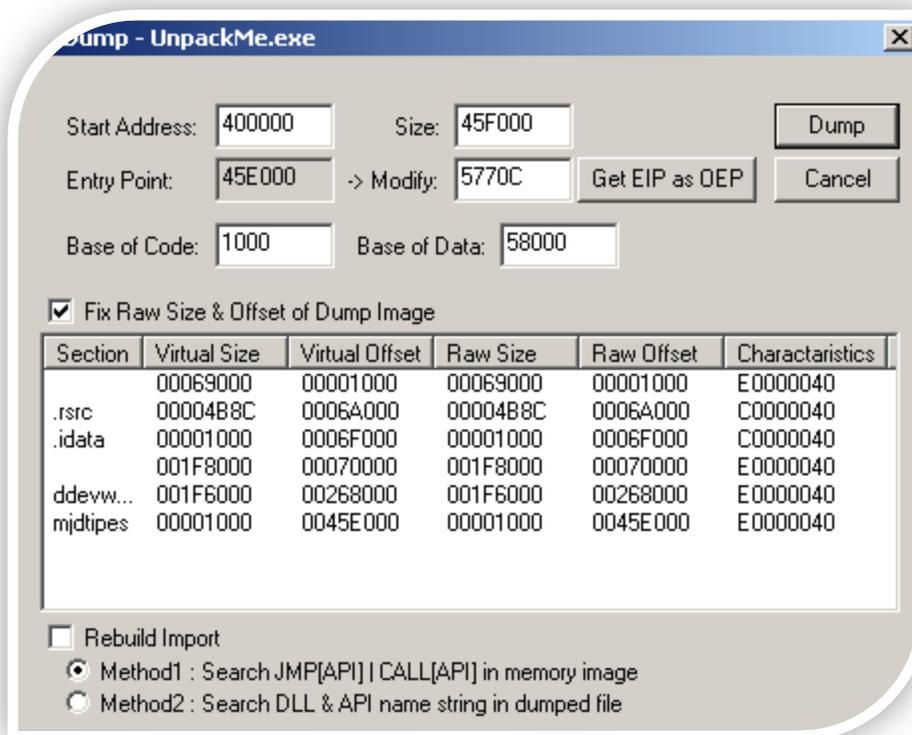
Nos dirigimos a nuestro OllyDBG con el progie packeado.

00406D58	OPC0	MOV EAX, EAX	
00406D58	- FF25 0CF94500	JMP DWORD PTR DS:[45F90C]	user32.GetWindow
00406CF6	8BC0	MOV EAX, EAX	
00406CF8	-\$- FF25 0CF94500	JMP DWORD PTR DS:[45F90C]	user32.GetWindow

Address	Hex dump	Disassembly	Comment
00406D90	-\$- FF25 C4F84500	JMP DWORD PTR DS:[45F8C4]	user32.GetWindowThreadProcessId
00406D96	8BC0	MOV EAX, EAX	
00406D98	- FF25 C4F84500	JMP DWORD PTR DS:[45F8C4]	user32.GetWindowThreadProcessId
00406D9E	8BC0	MOV EAX, EAX	

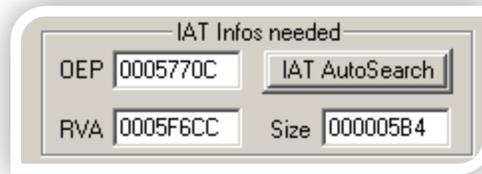
Address	Hex dump	Disassembly	Comment
00406EE8	-\$- FF25 1CF84500	JMP DWORD PTR DS:[45F81C]	user32.SendMessageA
00406EEE	8BC0	MOV EAX, EAX	
00406EF0	- FF25 1CF84500	JMP DWORD PTR DS:[45F81C]	user32.SendMessageA
00406FF4	8BC0	MOV EAX, EAX	

Nos disponemos a hacer el DUMP con el OllyDUMP.



Lo guardo con 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".

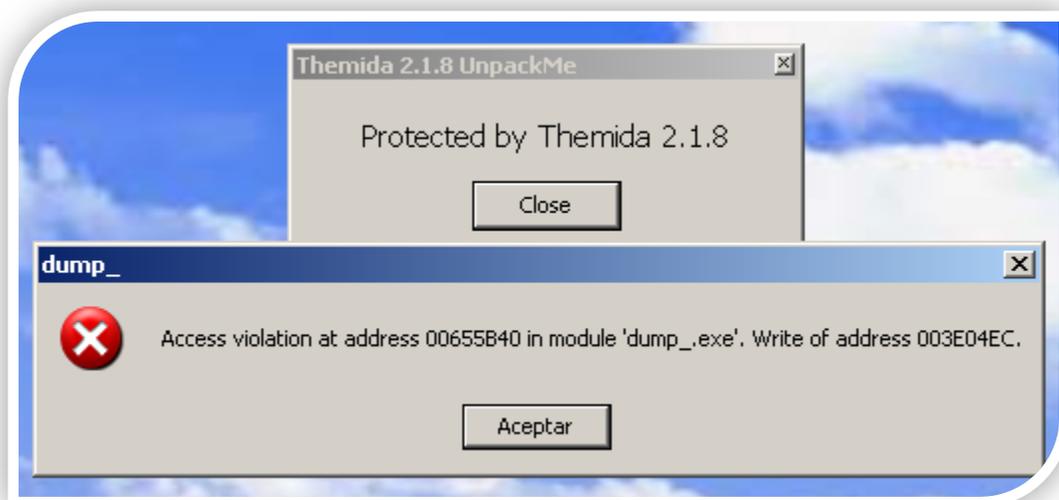
```
oleaut32.dll FTThunk:0005F6CC NbFunc:3 (decimal:3) valid:YES
advapi32.dll FTThunk:0005F6DC NbFunc:3 (decimal:3) valid:YES
user32.dll FTThunk:0005F6EC NbFunc:5 (decimal:5) valid:YES
kernel32.dll FTThunk:0005F704 NbFunc:22 (decimal:34) valid:YES
user32.dll FTThunk:0005F790 NbFunc:A3 (decimal:163) valid:YES
gdi32.dll FTThunk:0005FA20 NbFunc:37 (decimal:55) valid:YES
version.dll FTThunk:0005FB00 NbFunc:3 (decimal:3) valid:YES
kernel32.dll FTThunk:0005FB10 NbFunc:35 (decimal:53) valid:YES
advapi32.dll FTThunk:0005FBE8 NbFunc:4 (decimal:4) valid:YES
```

Ninguna entrada mala :)

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

```
Fixing a dumped file...
C (decimal:12) module(s)
161 (decimal:353) imported function(s).
*** New section added successfully. RVA:0045F000 SIZE:00002000
Image Import Descriptor size: F0; Total length: 1910
C:\Documents and Settings\Administrador\Escritorio\dump.exe saved successfully.
```

Ejecutamos, y damos clic al botón.



Upps...

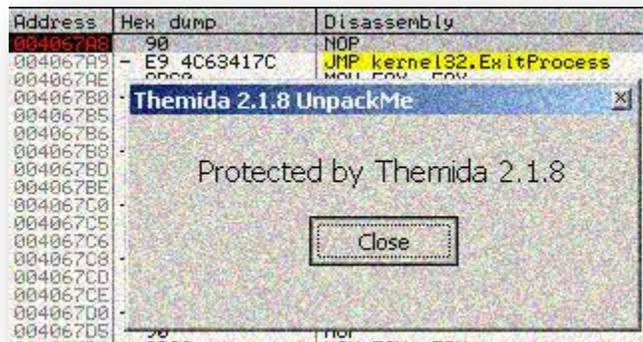
Si lo cerramos.



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

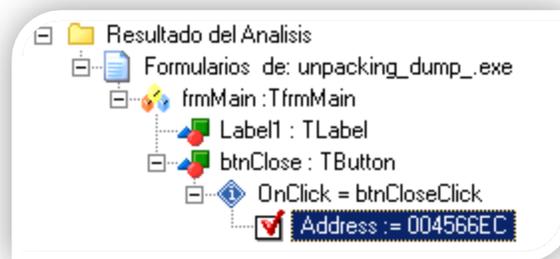
El evento `onClick()`, llama a `ExitProcess` para terminar el proceso, el que desea lo puede comprobar poniendo un BP a `ExitProcess`, o también en los Jumps indirectos.

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



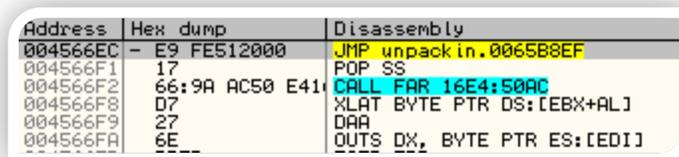
A partir de aquí, en la dirección que aparece el error al apretar el botón podemos ir hacia atrás y encontraremos el lugar que debemos fixiar.

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...

Address	Hex dump	Disassembly
004566EC	6A 00	PUSH 0
004566EE	E8 B500FBFF	CALL <JMP.&kernel32.ExitProcess>
004566F3	90	NOP

Guardamos los cambios.

Primer error fixiado, ahora falta la X.

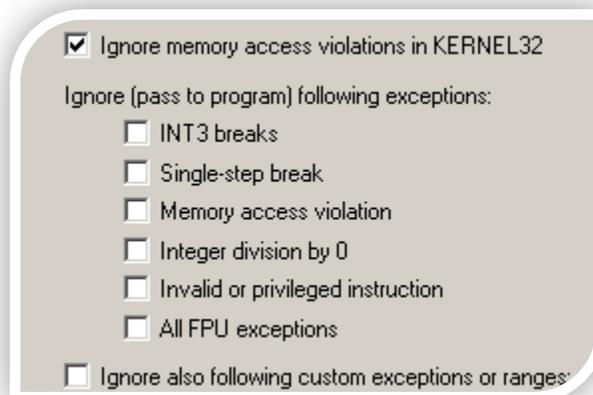
Este para mi fue el más problemático, si ven en el formulario (la imagen del E2A) no hay ningún evento onClose(), este evento no es gestionado, entonces el que se encarga aquí es el compilador, como lo hace el mismo delphi.

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

0040122A	8BC0	MOV EAX, EAX
0040122B	90	NOP
0040122D	E9 C8B8417C	JMP kernel32.ExitProcess
00401232	8BC0	MOV EAX, EAX

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:

0063ACB5	8D86 EC040000	LEA EAX, DWORD PTR DS:[ESI+4EC]
0063ACB8	F0:8618	LOCK XCHG BYTE PTR DS:[EAX], BL
0063ACBE	84DB	TEST BL, BL

Access violation when writing to [003E04EC] - use Shift+F7/F8/F9 to pass exception to program

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 fijiar.

Comenzemos!!!

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

Así que vayamos hacia atras!! :)

0063AC84	E9 C4FDFFFF	JMP unpack.in.0063AA40
0063AC89	E9 9C660000	JMP unpack.in.0064132A
0063AC8E	6A 00	PUSH 0
0063AC90	9C	PUSHFD
0063AC91	60	PUSHAD
0063AC92	90	NOP
0063AC93	90	NOP
0063AC94	E8 00000000	CALL unpack.in.0063AC99
0063AC99	5D	POP EBP
0063AC9A	81ED 85AC9A0A	SUB EBP, 0A9AC85
0063ACA0	90	NOP
0063ACA1	90	NOP
0063ACA2	B8 F2129B0A	MOV EAX, 0A9B12F2
0063ACA7	01E8	ADD EAX, EBP
0063ACA9	50	PUSH EAX
0063ACAA	8BB5 481B7E0A	MOV ESI, DWORD PTR SS:[EBP+A7E1B48]
0063ACB0	BB 01000000	MOV EBX, 1
0063ACB5	8D86 EC040000	LEA EAX, DWORD PTR DS:[ESI+4EC]
0063ACB8	F0:8618	LOCK XCHG BYTE PTR DS:[EAX], BL
0063ACBE	84DB	TEST BL, BL
0063ACC0	75 02	JNZ SHORT unpack.in.0063ACC4
0063ACC2	EB 14	JMP SHORT unpack.in.0063ACD8

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.

Address	Disassembly
0063AC8E	PUSH 0
006412EE	JMP unpack in.0063AC8E
00641300	JMP unpack in.0063AC8E

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

Search for references in:

Executable code of corresponding module

Memory block currently selected in Disassembler

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 -->

006412E0	0000	MOV BYTE PTR DS:[EAX], HL
006412E7	DF00	FILD WORD PTR DS:[EAX]
006412E9	68 6BC19447	PUSH 4794C16B
006412EE	E9 9B99FFFF	JMP unpack in.0063AC8E

Ctrl + R.

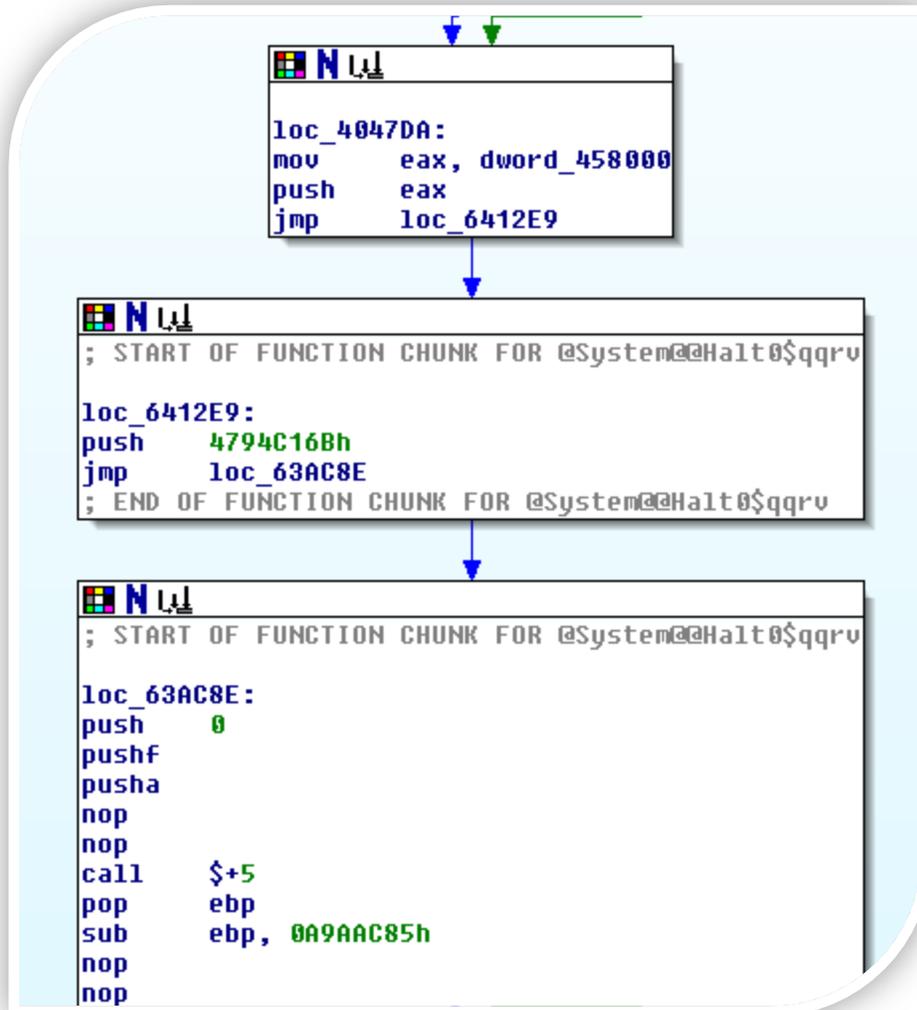
Address	Disassembly
004047E0	JMP unpack in.006412E9
006412E9	PUSH 4794C16B

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

004047D9	FF 15 24004501	CALL DWORD PTR DS:[450024]
004047DA	> 91 00804500	MOV EAX, DWORD PTR DS:[458000]
004047DF	. 50	PUSH EAX
004047E0	.- E9 04CB2300	JMP unpack in.006412E9
004047E5	> 8B03	MOV EAX, DWORD PTR DS:[EBX]
[00458000]=00000000		
Jump from 004047D2		

Este es el punto que debemos de fijar, 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 había una palabrita que resaltaba, usa IDA!!!!

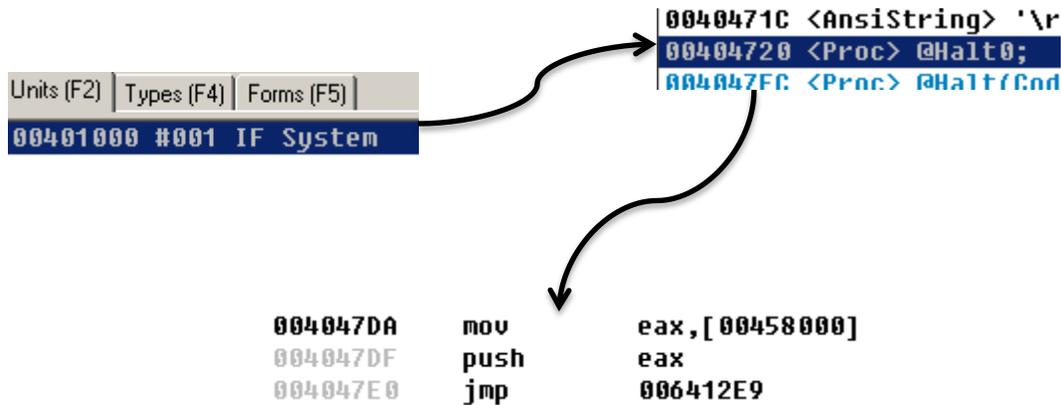


Vaya, me hubiera ahorrado mucho tiempo, si hubiera pensado en esto jeje, bueno es la primera vez que uso IDA en un tute XD...

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.

```

004047DA    > A1 00804500 MOV EAX, DWORD PTR DS:[458000]
004047DF    . 50          PUSH EAX
004047E0    \ E8 47CAFFFF CALL <JMP.&kernel32.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.

