

UnpackMess



CONCURSO 8.

Objetivo: Unpacking!

Packer: Themida v2.x

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 [.Net]

Agosto – 2012

Introducción

Quiero comenzar expresando que lamento enviar la solución al concurso tan tarde, esto ya se había resuelto en su mayor porcentaje, unos días después que empezó el concurso, pero por perdida del script que reparaba la IAT y el JMP Table (eso solo me pasa a mi :P), y luego el laburo se vino encima ya no pude rehacer el script hasta ahora.

Bien todo comenzó unos días después de que comenzara el concurso, mi amigo Eddy me había comentado sobre el unpackme del concurso del mes... Themida, encontré un tiempo y lo miré, para sorpresa mía todo me parecía familiar, es decir que las dos semanas de traceo en el anterior unpackme de Themida me sirvieron, pude sacar todos los puntos y tenía el script prácticamente hecho el mismo día.... Lo que pasó luego ya lo saben, perdí el Script, recuerdo haberme liado demasiado y modifiqué muchas cosas del antiguo Script que había hecho.

Al volver hacer el script, me basé en el anterior, para eso busqué – y es lo que me tomó más tiempo – el punto dónde obtiene las APIs buenas que me permita no modificar código del Script anterior, si no solamente los datos – puntos – tales como direcciones del OEP, IAT, JMP Table, etc.

Primero debemos hallar la forma de encontrar el OEP, la excelente herramienta de RDGMax, RDG Packer Detector nos dice que es un Borland Delphi v6.0 – v7.0 PACKER: ZProtect, pero no detecta el packer Themida por ningún lado, felizmente el creador del unpackme nos dio los datos de los packers que fueron usados.

Para hallar el OEP usaremos el Callback, esta manera de encontrar el OEP también puede ser usada para los VC++ (testado en las últimas versiones), Delphi y ASM empacados, la mayoría de las veces tiene éxito, y se describirá en el siguiente apartado.

Método del Callback.

Al correr dentro del debugger el target, ponemos un *Memory BreakPoint on Access* en la sección de código, en esa sección se encuentra el Callback que gestiona los mensajes de windows del formulario creado, también llamado *DlgProc* o *WndProc*.

Al crear un formulario se establece como uno de sus parámetros el puntero a la función *Callback* del formulario, encontrar dicha función y retroceder nos lleva a dónde empezó la ejecución del código, un lugar muy cercano al OEP.

Hacemos lo explicado con el target, comenzamos poniendo un BPM on Access en la sección de código y la depuración es pausada (si no pausa interactúen con el formulario, si de esa forma tampoco para, es porque la función *Callback* se encuentra en otra sección), en este target para en otra dirección que no es dónde rompe, pero eso fácilmente se puede solucionar revisando la subventana LOG, identificando la dirección dónde rompió realmente.

En este caso el desensamblado nos muestra otra dirección, y la subventana LOG nos muestra la dirección correcta.

Address	Hex dump	Disassembly	Comment
5B150000			Module C:\WINDOWS\system32\uxtheme.dll
7C91E4F4	C3	RETN	004CF21B Memory breakpoint when executing [004CF21B]
7C91E4F5	8DA424 00000000	LEA ESP, DWORD	Analysing dell.dll
7C91E4F6	00000000	LEA ESP, DWORD	1697 heuristical procedures

Si vamos a esa dirección e intentamos retroceder sería un buen lío, ya que no se encuentra ninguna referencia, para solucionar ese inconveniente observamos la pila e identificamos la dirección de retorno después de terminar el flujo de la rutina dónde está pausado el debugger.

Address	Hex dump	Disassembly
0044CCD0	> 59	XOR EAX, EAX
0044CCD1	. 5A	POP EDI
0044CCD2	. 59	POP ECX
0044CCD3	. 59	POP ECX
0044CCD4	. 64:8910	MOV DWORD PTR FS:[EAX], EDX
0044CCD5	. 68 EFC4400	PUSH de1de11.0044CCE7
0044CCD6	> 8D45 F8	LEA EAX, DWORD PTR SS:[EBP-10]
0044CCD7	. BR 02000000	MOV EDI, 2
0044CCD8	. E8 4D71FBFF	CALL de1de11.00403E34
0044CCD9	. C3	RETN
0044CCDA	. E9 276FBFFF	JMP de1de11.00403814
0044CCDB	. ^ EB EB	JMP SHORT de1de11.0044CCDA
0044CCDC	. 5F	POP EDI
0044CCDD	. 5E	POP ESI
0044CCDE	. 5B	POP EBX
0044CCDF	. 8BE5	MOV ESP, EBP
0044CE00	. 5D	POP EBP
0044CE01	. C3	RETN

Address	Value	Comment
0012FED0	02E10ADC	RETURN to 02E10ADC
0012FED4	0044CCD0	RETURN to de1de11.0044CCD0 from de1de11.0040662
0012FED8	0012FF2C	Pointer_ to next SEH record

La dirección 44CCCD pertenece a la función *CallBack* del formulario, a partir de esta rutina comenzaremos a retroceder, para eso nos dirigimos al inicio de la función y buscamos las referencias a la misma.

Address	Disassembly
RETN	
PUSH EBP	
MOV EBP, ESP	
ADD ESP, -10	
PUSH EBX	
PUSH ESI	
PUSH EDI	
VBP EBP EBP	

Address	Disassembly
0044C31F	CALL de1de11.0044CB08
0044CB08	PUSH EBP

Hacemos el mismo procedimiento, nos dirigimos a la dirección 44C31F y buscamos el inicio de la rutina.

0044C304	. 5F	PUP EDI
0044C305	. 5E	POP ESI
0044C306	. 5B	POP EBX
0044C307	. C3	RETN
0044C308	. 53	PUSH EBX
0044C309	. 83C4 E4	ADD ESP, -1C
0044C30C	. 8BD8	MOV EBX, EAX
0044C30E	. 8BD4	MOV EDX, ESP
0044C310	. 8BC3	MOV EAX, EBX
0044C312	. E8 59FFFFFF	CALL de1de11.0044C270
0044C317	. 84C0	TEST AL, AL
0044C319	. 75 09	JNZ SHORT de1de11.0044C324
0044C31B	. 8BD4	MOV EDX, ESP
0044C31D	. 8BC3	MOV EAX, EBX
0044C31F	. E8 B4080000	CALL de1de11.0044CB08
0044C324	> 83C4 1C	ADD ESP, 1C
0044C327	. 5B	POP EBX
0044C328	. C3	RETN

Address	Disassembly
00449129	CALL de1de11.0044C308
0044C308	PUSH EBX
0044C532	CALL de1de11.0044C308

Nos dirigimos a la dirección 44C532, luego al irnos al inicio de la rutina encontramos las referencias a ella.

0044C498	. C3	RETN
0044C499	. 8D40 00	LEA EAX, DWORD PTR DS:[EAX]
0044C49C	. 55	PUSH EBP
0044C49D	. 8BEC	MOV EBP, ESP

Address	Disassembly
0044C49C	PUSH EBP
0044DDC0	DB E8

DB E8?, problemas del Olly y su análisis, pero E8 es el *opcode* de un CALL relativo, y el OEP de un Delphi está copado de CALLs, pos nos vamos a esa dirección.

Address	Hex dump	Disassembly
0044DD74	55	PUSH EBP
0044DD75	8BEC	MOV EBP, ESP
0044DD77	83C4 F0	ADD ESP, -10
0044DD7A	B8 94DB4400	MOV EAX, dellhell.0044DB94
0044DD7F	E8 447EFBFF	CALL dellhell.00405BC8
0044DD84	A1 C8EF4400	MOV EAX, DWORD PTR DS:[44EFC8]
0044DD89	8B00	MOV EAX, DWORD PTR DS:[EAX]
0044DD8B	E8 74E6FFFF	CALL dellhell.0044C404
0044DD90	A1 C8EF4400	MOV EAX, DWORD PTR DS:[44EFC8]
0044DD95	8B00	MOV EAX, DWORD PTR DS:[EAX]
0044DD97	BA 04DD4400	MOV EDX, dellhell.0044DD04
0044DD9C	E8 73E2FFFF	CALL dellhell.0044C014
0044DDA1	8B00 A4F04400	MOV ECX, DWORD PTR DS:[44F0A4]
0044DDA7	A1 C8EF4400	MOV EAX, DWORD PTR DS:[44EFC8]
0044DDAC	8B00	MOV EAX, DWORD PTR DS:[EAX]
0044DDAE	8B15 A4D94400	MOV EDX, DWORD PTR DS:[44D9A4]
0044DDB4	E8 63E6FFFF	CALL dellhell.0044C41C
0044DDB9	A1 C8EF4400	MOV EAX, DWORD PTR DS:[44EFC8]
0044DDBE	8B00	MOV EAX, DWORD PTR DS:[EAX]
0044DDC0	E8 07E6FFFF	CALL dellhell.0044C49C
0044DDC5	E8 565FFBFF	CALL dellhell.00403D20

Y tenemos el OEP en la dirección 44DD74, ahora necesitamos ver en que estado quedado la IAT.

Reparando la IAT y el JMP Table.

Aquí es dónde se encuentra el laburo y la mayor parte de este tute será escrito en este apartado.

Si miramos los intermodular calls, nos daremos cuenta de que existen entradas buenas, y luego nos dirigimos al desensamblado para mirar el estado del *JMP Table*...

Address	Hex dump	Disassembly
004011F4	- E9 C7F28302	JMP 02C404C0
004011F9	90	NOP
004011FA	8BC0	MOV EAX, EAX
004011FC	90	NOP
004011FD	- E9 FEED8302	JMP 02C40000
00401202	8BC0	MOV EAX, EAX
00401204	- E9 CFF18202	JMP 02C30308
00401209	90	NOP
0040120A	8BC0	MOV EAX, EAX
0040120C	- E9 EFED8202	JMP 02C30000
00401211	90	NOP
00401212	8BC0	MOV EAX, EAX
00401214	- E9 0A058202	JMP 02C21723
00401219	90	NOP
0040121A	8BC0	MOV EAX, EAX
0040121C	- E9 29FA8402	JMP 02C50C4A
00401221	90	NOP
00401222	8BC0	MOV EAX, EAX
00401224	90	NOP
00401225	- E9 D0B8417C	JMP kernel32.ExitProcess
0040122A	8BC0	MOV EAX, EAX
0040122C	- E9 75F68402	JMP 02C508A6
00401231	90	NOP
00401232	8BC0	MOV EAX, EAX
00401234	- E9 E4FD8102	JMP 02C21010
00401239	90	NOP
0040123A	8BC0	MOV EAX, EAX
0040123C	- E9 BFE8102	JMP 02C20000
00401241	90	NOP
00401242	8BC0	MOV EAX, EAX
00401244	- E9 A6F58002	JMP 02C107EF
00401249	90	NOP
0040124A	8BC0	MOV EAX, EAX
0040124C	- E9 20F58002	JMP 02C10771
00401251	90	NOP
00401252	8BC0	MOV EAX, EAX
00401254	- E9 A7ED8002	JMP 02C10000

Los saltos son hacia una sección creada por el packer y un salto relativo a una API ExitProcess... NOPs que rellenan el byte extra de los 6 bytes que ocupa el JMP indirecto.

Para empezar a reparar, debemos ver en que momento escribe los saltos, calcula el desplazamiento a una API correcta o a una sección del packer, para esto ponemos un Hardware BreakPoint on Write en la dirección 4011F4 - inicio del *JMP Table*- y asegurarnos que ocupe 4 bytes, reiniciamos el debugger, luego RUN hasta que rompa como se muestra en la siguiente imagen.

Address	Hex dump	Disassembly
0052E131	F3:A4	REP MOVS BYTE PTR ES:[EDI], BYTE PTR DS:[ESI]
ECX=00059DF0 (decimal 368112.)		
DS:[ESI]=[02A50210]=90		
ES:[EDI]=[00401210]=94		
Address	Hex dump	ASCII
004011F4	90 90 90 90 90 90 8B C0	000000<A0000
00401204	90 90 90 90 90 90 8B C0 90 90 90 90 94 2B 84 9F	000000<A0000

En esta parte se NOPea, una forma de limpiar el lugar dónde irá los JMPs relativos que crea el packer y sigue la siguiente forma.

Queda como sigue:

Address	Hex dump	Disassembly
004011F4	90	NOP
004011F5	90	NOP
004011F6	90	NOP
004011F7	90	NOP
004011F8	90	NOP
004011F9	90	NOP
004011FA	8BC0	MOV EAX, EAX
004011FC	90	NOP
004011FD	90	NOP
004011FE	90	NOP
004011FF	90	NOP
00401200	90	NOP
00401201	90	NOP
00401202	8BC0	MOV EAX, EAX
00401204	90	NOP
00401205	90	NOP
00401206	90	NOP
00401207	90	NOP
00401208	90	NOP
00401209	90	NOP
0040120A	8BC0	MOV EAX, EAX
0040120C	90	NOP
0040120D	90	NOP
0040120E	90	NOP
0040120F	90	NOP
00401210	90	NOP
00401211	90	NOP
00401212	8BC0	MOV EAX, EAX
00401214	90	NOP
00401215	90	NOP
00401216	90	NOP
00401217	90	NOP
00401218	90	NOP
00401219	90	NOP
0040121A	8BC0	MOV EAX, EAX
0040121C	90	NOP
0040121D	90	NOP

Ejecutamos de nuevo, y en el dump podemos observar.

Address	Hex dump	ASCII
004011F4	E9 90 90 90 90 90 8B C0	é00000<Aeè0_0<A
00401204	E9 F7 ED 84 02 90 8B C0	e+i_00<Ae0+f00<A
00401214	E9 E7 ED 83 02 90 8B C0	éçif00<A000000<A
00401224	90 E9 00 B8 41 7C 8B C0	0é0_A <A000000<A
00401234	E9 3B FE 82 02 90 8B C0	é;b_00<Aé;i_00<A
00401244	E9 35 F4 81 02 90 8B C0	e50000<Ae00000<A
00401254	E9 A7 ED 81 02 90 8B C0	é5i000<AeYi000<A
00401264	E9 53 F5 7F 02 90 8B C0	é50000<Ae0i000<A
00401274	E9 87 ED 7E 02 90 8B C0	é+i~00<AéH0}00<A
00401284	E9 72 F2 7D 02 90 8B C0	ér0}00<A000000<A
00401294	E9 5D F0 7D 02 90 8B C0	é]0}00<Ae_i}00<A
004012A4	E9 57 ED 7C 02 90 8B C0	éW 00<A000000<A
004012B4	90 90 90 90 90 90 8B C0	000000<A000000<A
004012C4	E9 37 ED 7B 02 90 8B C0	é7i{00<Ae}0z00<A
004012D4	90 90 90 90 90 90 8B C0	000000<A000000<A
004012E4	90 90 90 90 90 90 8B C0	000000<Ae40z00<A
004012F4	E9 45 F2 7A 02 90 8B C0	éE0z00<Ae00z00<A
00401304	E9 15 F0 7A 02 90 8B C0	e00z00<A5fÅx»...

F9 de nuevo, para ejecutar el debugger, así encontrar en que momento escribe el desplazamiento.

Address	Hex dump	Disassembly	Registers (F)
0049A980	E9 69F3FFF	JMP dell dell.00499DFB	EAX 00000061
0049A992	8F02	POP DWORD PTR DS:[EDX]	ECX 00000001
0049A994	E9 30C6FFFF	JMP dell dell.004970C9	EDX 004011F5

Address	Hex dump	Address	Value	Comment
004011F4	E9 95 FB 84 02	0012FF50	000000E0	
00401204	E9 F7 ED 84 02	0012FF54	000011F5	
00401214	E9 E7 ED 83 02	0012FF58	000000FE	
00401224	90 E9 00 B8 41 7C	0012FF5C	00000001	
00401234	E9 3B FE 82 02	0012FF60	00000061	
00401244	E9 35 F4 81 02	0012FF64	0284FB95	
00401254	E9 87 ED 7E 02	0012FF68	00543008	dell dell 00543008

Usando la instrucción POP escribe el offset que se encontró en la pila, y en EDX la dirección dónde pondrá los 4 bytes restantes del JMP relativo.

El offset que está escribiendo es de una entrada mala, y lo que se mostró en la imagen anterior, es la instrucción que se usa para escribir el JMP a una API emulada.

Para empezar un análisis más completo de las entradas buenas y malas, como las resuelve y el punto dónde obtiene todas las APIs. El *JMP Table* como en el anterior escrito que hice están dispersas, es decir encuentras un segmento de esta tabla en la dirección 4011F4, la siguiente está en la dirección 00405B04, otra en 00405D88, etc.

Para comenzar analizar como resuelve la *JMP Table*, podemos poner un BPM on Write, y ver a partir lo que muestre el OllyDBG al pausar. Un inconveniente es que Themida no las deja tan fácil, debemos encontrar un punto dónde no detecte los Memory BreakPoint, para eso reiniciamos y llegamos a la dirección 0052E131 dónde nopea la *JMP Table*, dejándola lista para escribir como se le venga en gana al packer.

```

0052E131 F3:A4 REP MOVSB BYTE PTR ES:[EDI], BYTE PTR DS:[ESI]
0052E133 C685 4B17A708 MOV BYTE PTR SS:[EBP+8A7174B], 56
0052E13A B8 396D1FD4 MOV EAX, D41F6D39

```

Cuando el registro EIP se encuentra en la dirección 0052E133h, ponemos un BPM on Write, en la dirección dónde comienza el *JMP Table*, 004011F4h abarcando con el tamaño de 20960h aproximadamente, coge todo los segmentos del *JMP Table*.

En OllyScript: BPWM 004011F4, 20960

En el CommandBar: MW 004011F4, 00421B54

El tamaño del BPM excede al *JMP Table* total, pero esto no afecta en ninguna manera el análisis.

Al continuar la ejecución del Unpackme dentro del debugger rompe para escribir el *opcode* E9 del JMP relativo, en el registro EDX y EBX está la dirección dónde escribirá dicho *opcode*.

Address	Hex dump	Disassembly	Registers (FF)
00498F60	88C5	MOV CH, AL	EAX 000000E9
00498F6F	882B	MOV BYTE PTR DS:[EBX], CH	ECX 0000E901
00498F71	E9 9BEBFFFF	JMP delldell.00497B11	EDX 00401368
00498F76	80C3 AD	ADD BL, 0AD	EBX 00401368
00498F79	E9 37030000	JMP delldell.004992B5	ESP 0012FF60
00498F7E	893C24	MOV DWORD PTR SS:[ESP], EDI	EBP F79F2014
00498F81	89E7	MOV EDI, ESP	ESI 00530CF2
00498F83	68 906B0000	PUSH EB90	EDI 00496574
00498F88	891424	MOV DWORD PTR SS:[ESP], EDX	

Address	Hex dump	Disassembly
00401368	-\$- E9 90909090	JMP 9000A3FD
0040136D	90	NOP

Ejecutamos el Unpackme dentro del debugger, y el mnemónico POP se encarga de escribir el offset a una sección dónde emula la API correspondiente, el opcode de esta instrucción es 8F02, en el registro EDX se encuentra la dirección dónde se escribirá el offset hacia una sección.

Address	Hex dump	Disassembly	Registers (FF)
0049AA92	8F02	POP DWORD PTR DS:[EDX]	EAX 00000061
0049AA94	E9 30C6FFFF	JMP delldell.004970C9	ECX 00000001
0049AA99	66:8B0C24	MOV CX, WORD PTR SS:[ESP]	EDX 00401369
0049AA9D	E9 59000000	JMP delldell.0049A9FB	EBX 0000009F
0049AAA2	89DE	MOV ESI, EBX	ESP 0012FF64
0049AA04	89F2	MOV ESI, ESI	

Address	Value
0012FF64	0279EC93

Address	Hex dump	Disassembly
00401368	-\$- E9 93EC7902	JMP 02BA0000
0040136D	90	NOP

Apretamos F9 para ejecutar, y ver el próximo JMP a escribir, rompe en un NOP a NOP, je, pero vemos como rellena con basura (en otros momentos pone cualquier valor).

```

00542ABF AA STOS BYTE PTR ES:[EDI]
AL=90
ES:[EDI]=[00401360]=90

```

El mnemónico STOS incremente EDI en uno, de esta manera la dirección dónde el packer comenzaría el escribir el opcode E9 del JMP, sería en 401361, dejando de lado un byte de los 6 bytes que ocupaba el JMP originalmente.

parámetro es la cadena “NTDLL” y en el registro ESI, un puntero a la cadena “RtlEnterCriticalSection”.

Address	Hex dump	Disassembly
00F9AB31	8BFF	MOV EDI, EDI
00F9AB33	55	PUSH EBP
00F9AB34	8BEC	MOV EBP, ESP
00F9AB36	837D 08 00	CMP DWORD PTR SS:[EBP+8], 0
00F9AB3A	74 18	JE SHORT 00F9AB54
00F9AB3C	FF75 08	PUSH DWORD PTR SS:[EBP+8]
00F9AB3F	E8 C0290000	CALL 00F9D504
00F9AB44	85C0	TEST EAX, EAX
00F9AB46	74 08	JE SHORT 00F9AB50
00F9AB48	FF70 04	PUSH DWORD PTR DS:[EAX+4]
00F9AB4B	E8 7D2D0000	CALL 00F908C0
00F9AB50	5D	POP EBP
00F9AB51	C2 0400	RETN 4
00F9AB54	64:A1 18000000	MOV EAX, DWORD PTR FS:[18]
00F9AB5A	8B40 30	MOV EAX, DWORD PTR DS:[EAX+30]
00F9AB5D	8B40 38	MOV EAX, DWORD PTR DS:[EAX+38]
00F9AB60	EB EE	JMP SHORT 00F9AB50

Stack

Address	Value	Comment
0012FF40	005434A5	RETURN to dellde11.005434A5
0012FF44	00462085	ASCII "NTDLL"
0012FF48	00462085	ASCII "NTDLL"

Ctrl + F9

Registers (FPU)		
EAX	7C910000	ntdll.7C910000
ECX	00F9DABB	
EDX	7C98B178	ntdll.7C98B178
EBX	00542DEE	dellde11.00542DEE
ESP	0012FF40	
EBP	F79F2014	
ESI	7C8090A7	ASCII "RtlEnterCr
EDI	0046208B	dellde11.0046208B

Seguimos analizando el run trace y luego de obtener la dirección de la librería que usará para obtener la API “RtlEnterCriticalSection”. Recorre la ExportTable de la librería NTDLL. Comenzando con la ExportNameTable, obtiene la posición de la función que se guarda en el registro contador ECX, luego con la instrucción SHL ECX, 1, multiplica por dos el valor de ECX, siendo necesario ya que la ExportOrdinalTable es una tabla de WORDs, con el índice de esta tabla podemos obtener el Ordinal de la función, la ExportAddressTable es una tabla de DWORDs, usando el Ordinal de la función como índice se obtiene el RVA de la función, y finalmente se le suma la ImageBase de la librería, en este caso la NTDLL para obtener la dirección de la función que será usado por el target.

ExportNameTable

Posición de la función

ExportOrdinalTable

ECX: Índice de la función

Ordinal de la función

ExportAddressTable

RVA de la función

Address de la función

Muchos pensarán que la dirección 4C1A76, es el punto mágico, pero esta parte es la del camino bueno por así decirlo, dónde crea el JMP relativo a la API. Si seguimos analizando, luego de pasar ofuscación encontramos un cambio de flujo, si el EIP está en la dirección 543692, el registro EAX tiene la API correcta, en ECX tenemos la librería que le pertenece a esa API, en este caso no tenemos problemas, porque en Kernel32 existe la declaración de esa API y de otras.

Address	Hex dump	Disassembly	Registers (FPU)
0054368C	8B85 5B9EB408	MOV EAX, DWORD PTR SS:[EBP+8B49E5B]	EAX 7C911000 ntdll.RtlEnterCriticalSection
00543692	0F8A 01000000	JPE delldell.00543699	ECX 7C800000 kernel32.7C800000
00543698	FC	CLD	EDX 3F1B46B4
00543699	C3	RETN	EBX 00542DEE delldell.00542DEE
0054369D	FC	CLD	

Luego sigue una serie de comparaciones de APIs y Flags, para los que leyeron mi anterior unpacking de themida lo expliqué en parte, y es que no hay mucha ciencia, comienza a comparar si la API en EAX, está definida por el packer para que el JMP sea a la API (como en este caso), si no se encuentra, el packer escribe un JMP a la dirección dónde tiene la API correspondiente emulada. Otras comparaciones son de librerías, si es Kernel32 o User32, si se cumple la condición los JMPs son a APIs emuladas, que contienen ofuscación. Después hay flags que como en el anterior unpacking de themida, que identifican si finalmente se escribirá un JMP a una API o una API emulada –como dicen mis amigos argentinos, todo un quilombo –.

Si ponemos un BP Log en la dirección 543692, observamos que están todas las APIs a comparación del punto 4C1A76 y otros que encontré en el transcurso del análisis, y que no mostraré en este tutorial, ya que no son de importancia.

Luego de este pequeño análisis, la segunda vez que rompe es la dirección 542ABF, esta parte ya la comenté antes, es un NOP a NOP, y de esta forma comienza la escritura del JMP relativo a una API

```

00542ABF AA STOS BYTE PTR ES:[EDI]
00542AC0 0F81 04000000 JNO dellde11.00542ACA
AL=90
ES:[EDI]=[00401358]=90

```

Ctrl + F11 otra vez, hasta que rompió, el registro ECX queda la dirección de la API que habíamos visto su string antes, "RtlEnterCriticalSection".

Address	Hex dump	Disassembly	Registers (FPU)
00542AEA	AA	STOS BYTE PTR ES:[EDI]	EAX 7C9110E9 ntdll.7C9110E9
00542AEB	FC	CLD	ECX 7C911000 ntdll.RtlEnterCriticalSection
00542AEC	E9 8E000000	JMP dellde11.00542B7F	EDX 3F1B46B4
00542AF1	E9 0C000000	JMP dellde11.00542B02	EBX 00000000
00542AF6	66:3940 53	CMP WORD PTR DS:[EAX+53]	ESP 0012FF70
00542AFA	3D 7533BE80	CMP EAX, 80BE3375	EBP F79F2014
00542AFF	78 71	JS SHORT dellde11.00542B01	ESI 005343B0 dellde11.005343B0
00542B01	5D	POP EBP	EDI 00401359 dellde11.00401359
00542B02	FF3424	PUSH DWORD PTR SS:[ESP]	

Algo que me olvidé comentar, es que esta versión de themida – así como la anterior que desempaqué – tiene IAT, cuando comencé a repararla con las entradas correctas, el packer las vuelve a pisar, veía entradas correctas e incorrectas, entonces la solución más rápida era encontrar un espacio donde poner la IAT correcta, esta se tomó de la sección de importaciones .idata, exactamente desde la dirección 007670B0.

Programando el Script

Para comenzar a programar el script tenemos los siguientes datos:

OEP: 0044DD74

IAT: 007670B0

Punto Mágico: 00543692

En el registro EAX está la dirección de la API.

En el registro ECX está la dirección de la librería al que pertenece la API.

Inicio del JMP Table: 4011F4, Final: 00421B54, Tamaño: 20960

Opcod del JMP a API: 0AB (opcode de la instrucción que se usa para escribir el offset del JMP a la API).

En EDI está la dirección dónde escribirá el offset del JMP

Opcod del JMP a API ofuscada: 8F02 (opcode de la instrucción que usa el packer para escribir el offset del JMP hacia la dirección que pertenece a una sección creada por el packer en runtime).

Script:

```
BPHWC  
VAR hLib  
VAR lpAPI  
VAR IAT  
VAR AddyJMP  
VAR Offset  
//OEP  
BPHWS 0044DD74  
// INICIO DE LA IAT  
MOV IAT, 0045F6CC  
// EAX = API.LIBRERIA  
// ECX = LIBRERIA  
BPHWS 007670B0 // 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, Tamaño: 20960.

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, 20960 //<- COMIENZA JMPS RELATIVOS - IAT  
BPWM 4011F4, 20960  
MOV hLib, ecx
```

Comparamos el valor que contiene la variable hLib, con el registro 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, 0044DD74
```

```
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, estas contienen la dirección a escribir el offset del JMP Relativo.

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 + 4], 90, 1

JZ Continua

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.

Salir:

BPHWC

BPMC

RET

lala:

CMP eip, 00648612

JZ EP

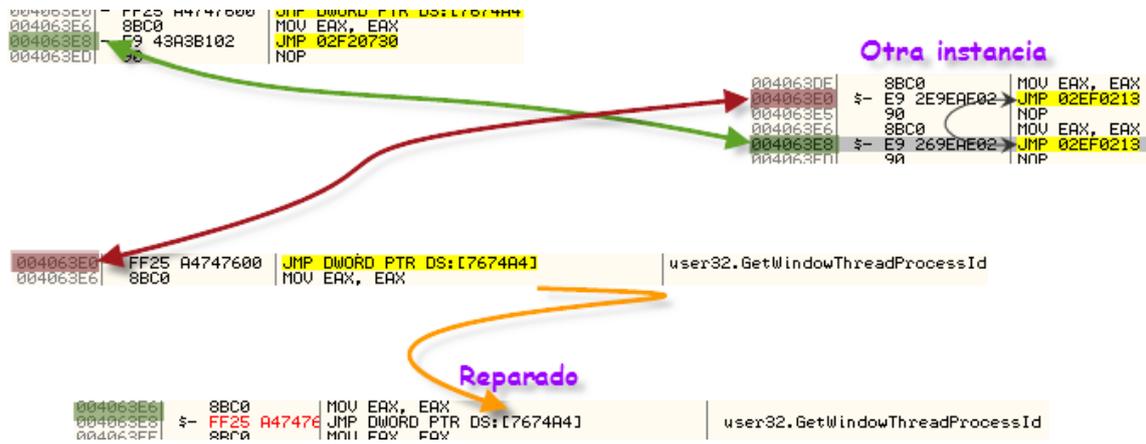
CMP eip, 0045770C

JZ Salir

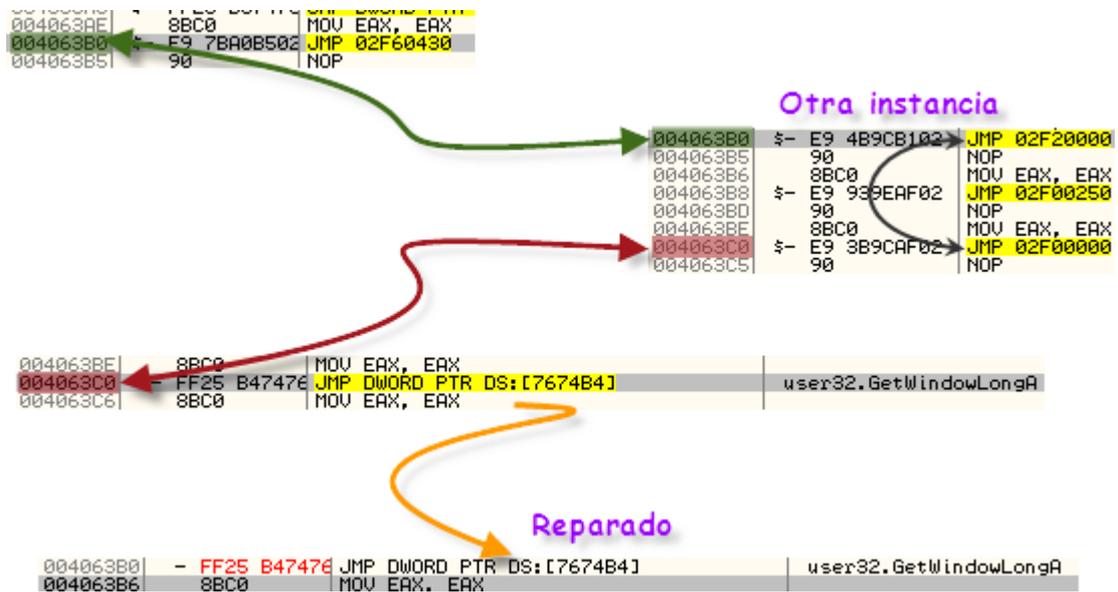
RUN

JMP lala

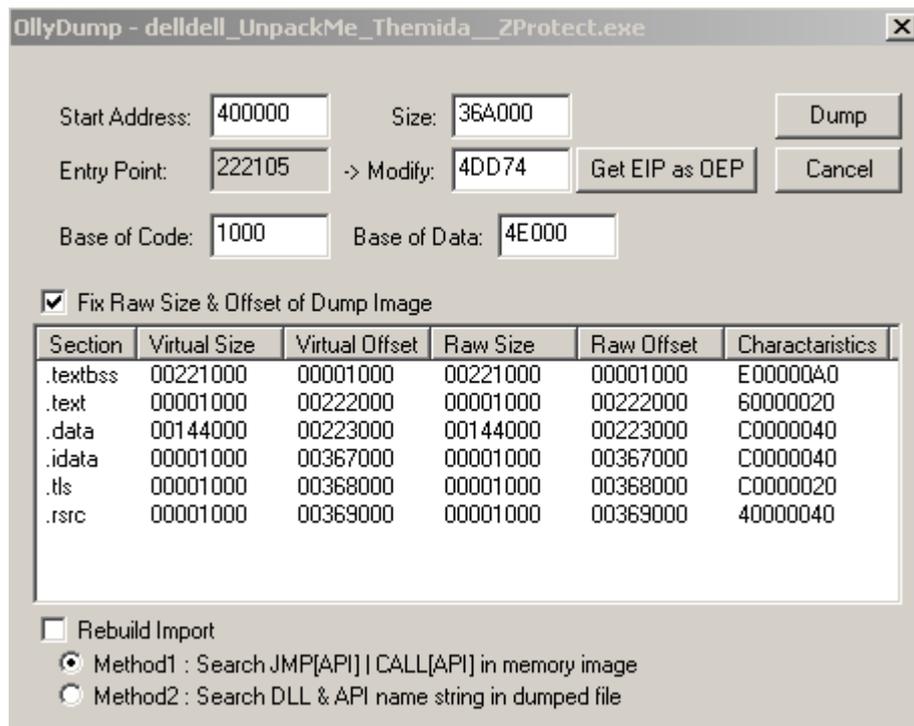
Cuando ejecutamos el script quedan sin resolver dos JMPs, la respuesta es que, al usar la misma API, el packer ya no vuelve a obtener la dirección de la API correspondiente, entonces ya no pasa por el punto mágico, y el script obvia dicha entrada. Para identificar que API saltará el JMP indirecto, podemos abrir instancia, ejecutar, buscar la misma dirección y encontrar en que dirección se repite el mismo salto.



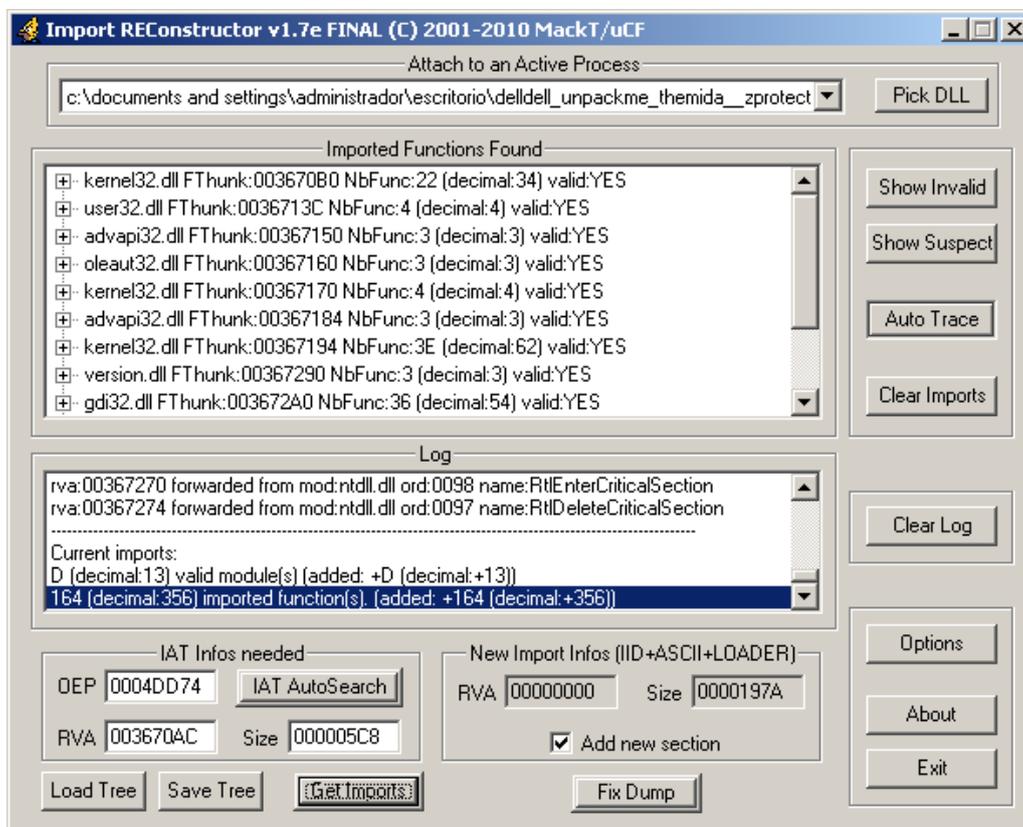
La API es GetWindowThreadProcessId, lo reparamos y pasamos al siguiente JMP no reparado.



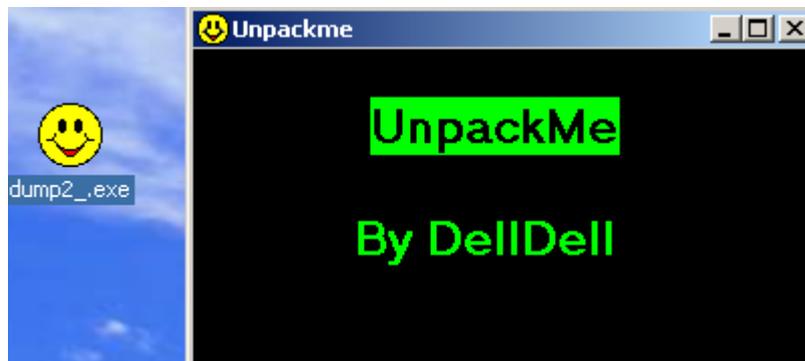
Luego de reparar los dos JMPs pasamos a dumpear como se muestra en la siguiente imagen.



Luego toca reparar la IAT, para eso abrimos el ImportRec, ponemos los datos del OEP: 4DD74, y clic a "IAT AutoSearch", encuentra toda la IAT correcta, después clic a "GetImports".



Todo está correcto, ahora reparamos el dump cliqueando en el botón “Fix Dump”, el ImportRec nos dice que ha sido reparado satisfactoriamente y que ah creado otro binario con la IT reparado.



Para que no crean que recién lo acabo de terminar, sólo estoy poniendo estas últimas palabras, lamento mandar tan pero tan tarde este tutorial, y muchas gracias a todos que han llegado hasta aquí, han leído todo el tute de unpackme, y un gran saludo a todos los de CLS Argentina que conocí en la EKO, así como a toda la lista en general, espero comenzar a escribir más seguido, como lo hacía antes, hasta el próximo escrito, que está bien pronto jeje.

Saludos,
Nox.