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++ (testeado 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.

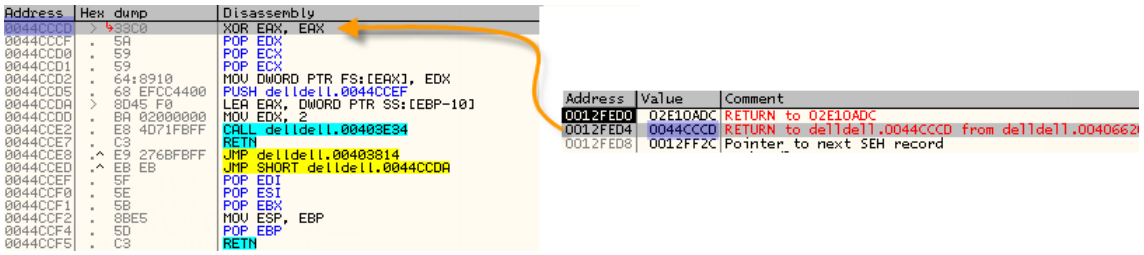
Al correr dentro del debugger el target, ponemos un *Memory BreakPoint on Access* en la sección de código, en esa sección se encuentra el CallBack que gestiona los mensajes de windows del formulario creado, también llamado *DlgProc* o *WndProc*.

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.

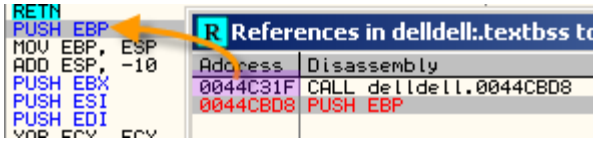
Hacemos lo explicado con el target, comenzamos poniendo un BPM on Access en la sección de código y la depuración es pausada (si no pausa interactúen con el formulario, si de esa forma tampoco para, es porque la función *CallBack* se encuentra en otra sección), en este target para en otra dirección que 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.

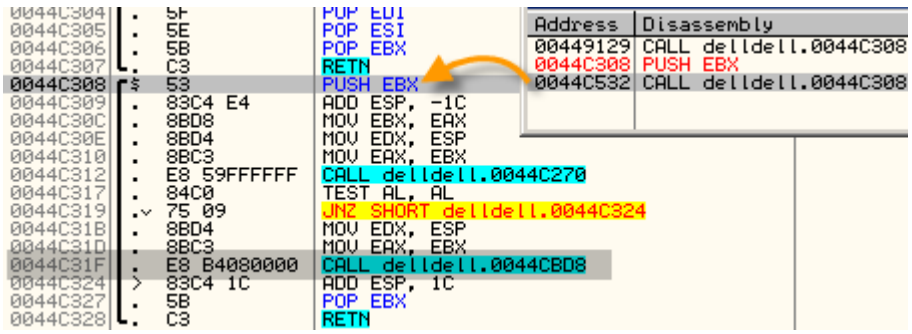
Si vamos a esa dirección e intentamos retroceder sería un buen lío, ya que no se encuentra ninguna referencia, para solucionar ese inconveniente observamos la pila e identificamos la dirección de retorno después de terminar el flujo de la rutina dónde está pausado el debugger.



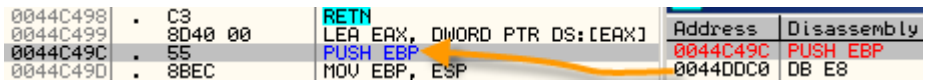
La dirección 44CCCD pertenece a la función *CallBack* del formulario, a partir de esta rutina comenzaremos a retroceder, para eso nos dirigimos al inicio de la función y buscamos las referencias a la misma.



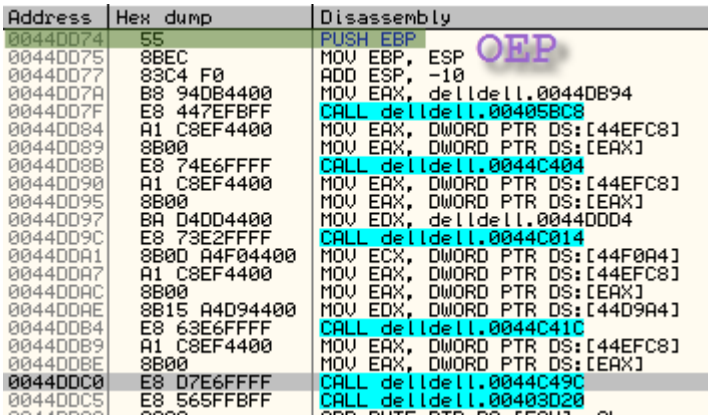
Hacemos el mismo procedimiento, nos dirigimos a la dirección 44C31F y buscamos el inicio de la rutina.



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



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.

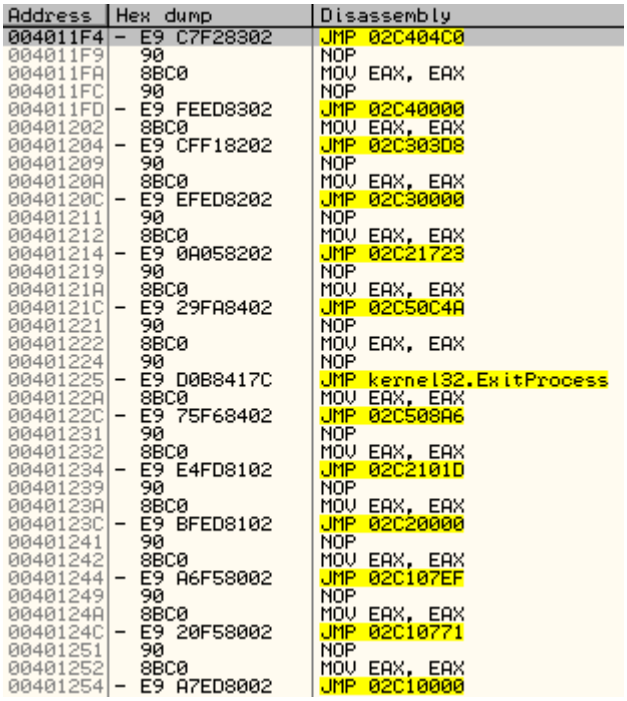


Y tenemos el OEP en la dirección 44DD74, ahora necesitamos ver en que estado aquedado la IAT.

Reparando la IAT y el JMP Table.

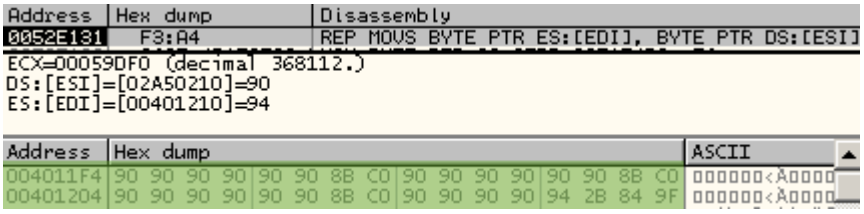
Aquí es dónde se encuentra el laburo y la mayor parte de este tute será escrito en este apartado.

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*...



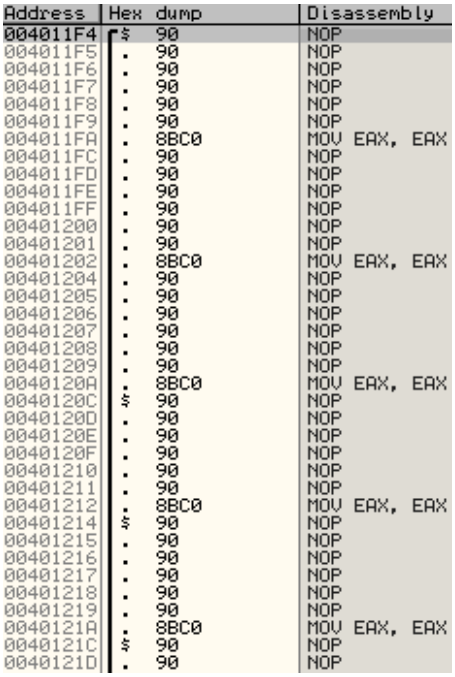
Los saltos son hacia una sección creada por el packer y un salto relativo a una API ExitProcess... NOPs que rellenan el byte extra de los 6 bytes que ocupa el JMP indirecto.

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.

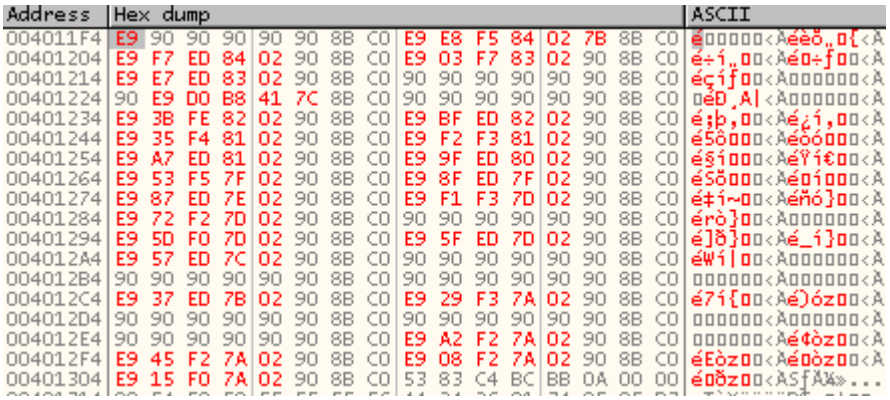


En esta parte se NOPea, una forma de limpiar el lugar dónde irá los JMPs relativos que crea el packer y sigue la siguiente forma.

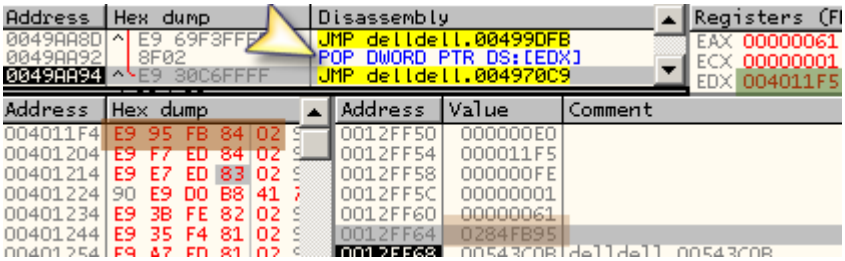
Queda como sigue:



Ejecutamos de nuevo, y en el dump podemos observar.



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

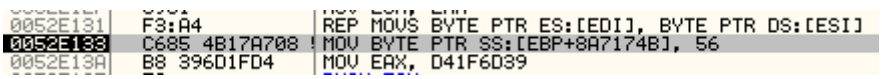


Usando la instrucción POP escribe el offset que se encontró en la pila, y en EDX la dirección dónde pondrá los 4 bytes restantes del JMP relativo.

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.



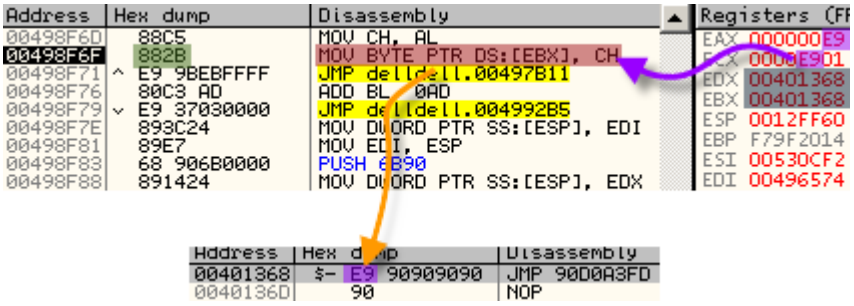
Cuando el registro EIP se encuentra en la dirección 0052E133h, ponemos un BPM on Write, en la dirección dónde comienza el *JMP Table*, 004011F4h abarcando con el tamaño de 20960h aproximadamente, coge todo los segmentos del *JMP Table*.

En OllyScript: BPWM 004011F4, 20960

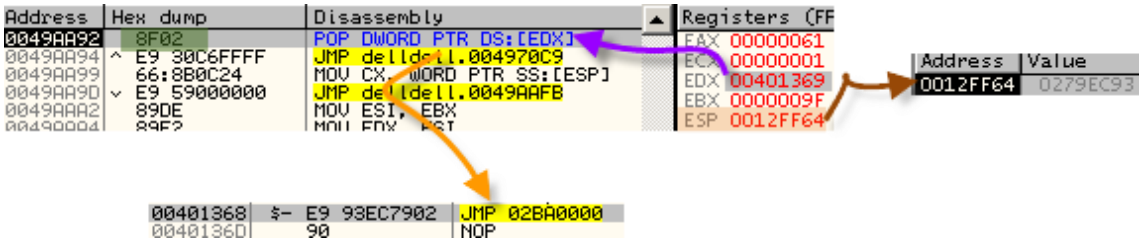
En el CommandBar: MW 004011F4, 00421B54

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

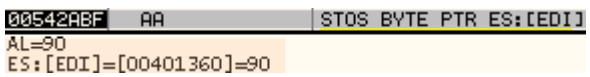
Al continuar la ejecución del Unpackme dentro del debugger rompe para escribir el *opcode* E9 del JMP relativo, en el registro EDX y EBX está la dirección dónde escribirá dicho *opcode*.



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.



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).



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.

Seguimos con el análisis, y ejecutamos tantas veces como sea factible para entender la siguiente imagen.

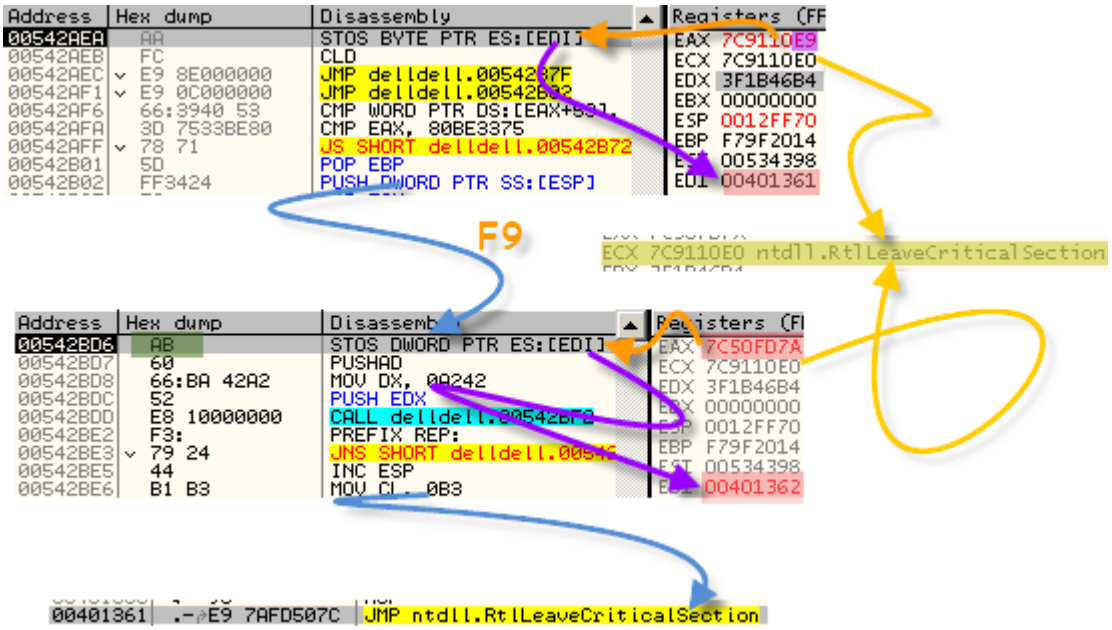


Imagen – JMP relativo hacia una API

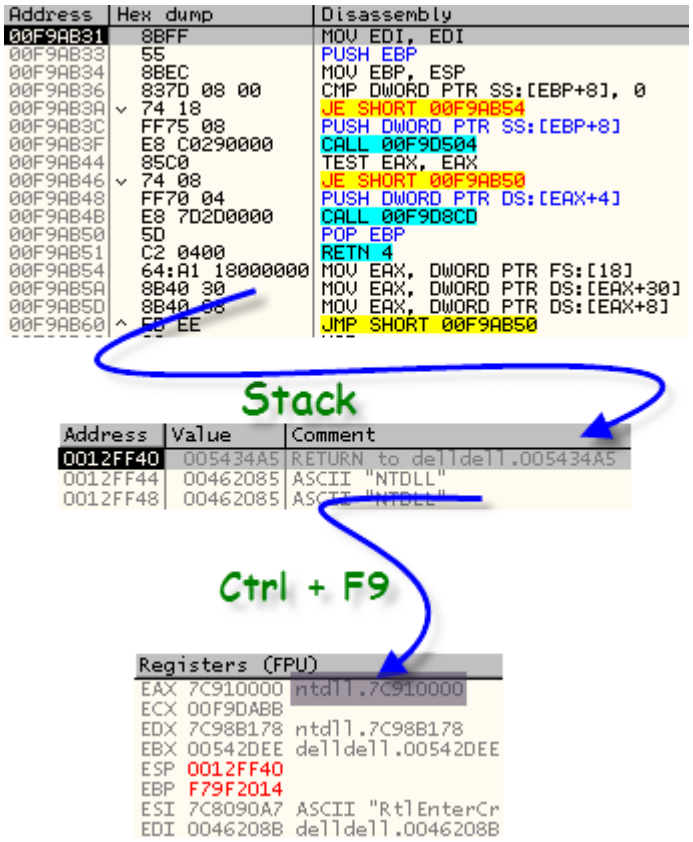
Algo que debemos tener en cuentra para el script que se desarrollará, son los opcodes, estos bytes serán usados para comprobar la ubicación en que está parado el debugger, saber si estamos escribiendo un JMP a una API o una sección del Packer.

Luego toca encontrar el punto mágico, este punto es dónde el packer obtiene todas las APIs que va a usar, existen muchos puntos dónde se puede ver obtener las APIs, pero al menos solo uno, de todos los que encontré da todas las APIs usadas por el target, para esto podemos hacer un run trace, así evitamos tracear un buena parte de código, ofuscación y basura de por medio. Para empezar podemos poner una condición para que rompa si el EIP está fuera de la sección de código.

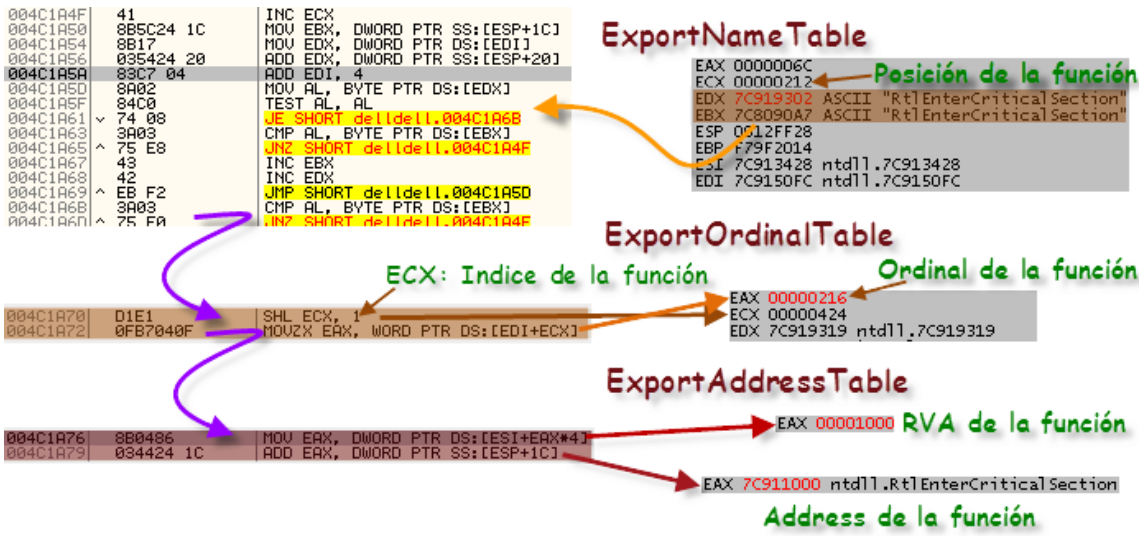
Y con Ctrl + F11, activamos el Trace Into, hasta que pare un par de veces, en una de ellas sale de la sección, si eso pasa, apretamos la combinación Ctrl + F9 hasta regresar a la sección de código, luego volver a Ctrl + F11 para que siga logeando los trace hasta que lleguemos a la dirección dónde escribe el siguiente JMP relativo como muestra la imagen de nombre “Imagen – JMP relativo hacia una API”. Las siguientes imágenes muestran el análisis basado en el run trace hecho.

Rompe en una función – EIP fuera de la sección de código –, por ser una función muy usada, se puede dar cuenta que es GetModuleHandle, el

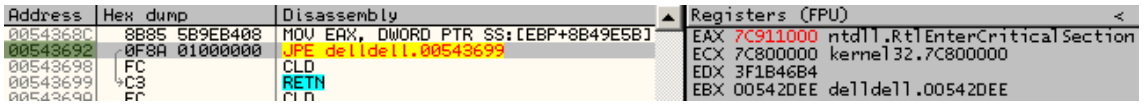
parámetro es la cadena “NTDLL” y en el registro ESI, un puntero a la cadena “RtlEnterCriticalSection”.



Seguimos analizando el run trace y luego de obtener la dirección de la librería que usará para obtener la API “RtlEnterCritialSection”. Recorre la ExportTable de la librería NTDLL. Comenzando con la ExporNameTable, 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.



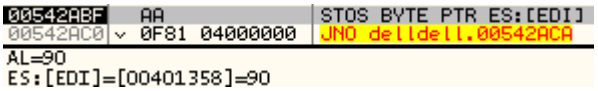
Muchos pensarán que la dirección 4C1A76, es el punto mágico, pero esta parte es la del camino bueno por así decirlo, dónde crea el JMP relativo a la API. Si seguimos analizando, luego de pasar ofuscación encontramos un cambio de flujo, si el EIP está en la dirección 543692, el registro EAX tiene la API 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.



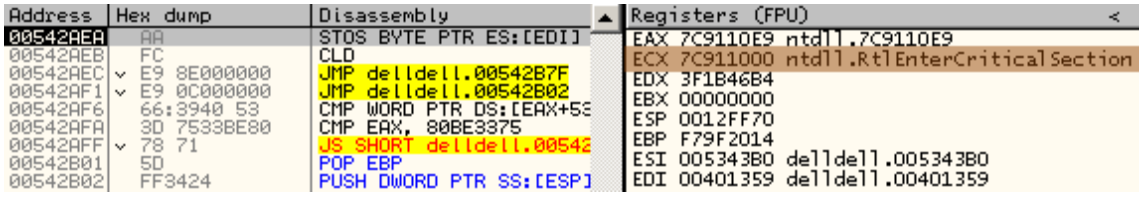
Luego sigue una serie de comparaciones de APIs y Flags, para los que leyeron mi anterior unpacking de themida lo expliqué en parte, y es que no hay mucha ciencia, comienza a comparar si la API en EAX, está definida por el packer para que el JMP sea a la API (como en este caso), si no se encuentra, el 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



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



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

Opcode 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

Opcode 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 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, 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

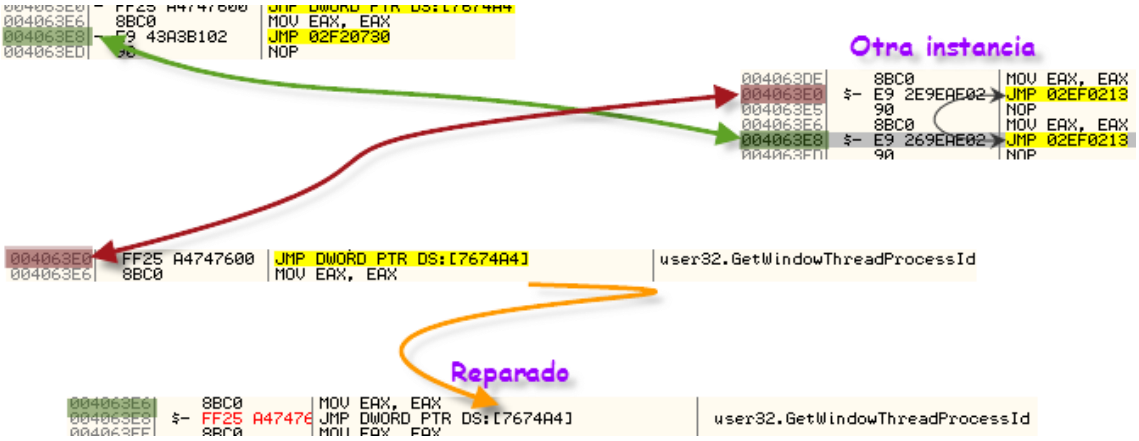
La etiqueta "Continua" lo que hace es simplemente escribir el JMP indirecto hacia la el address de la IAT con la entrada válida, se le suma un DWORD para la siguiente entrada y vuelve al bucle.

Salir:
BPHWC
BPMC
RET

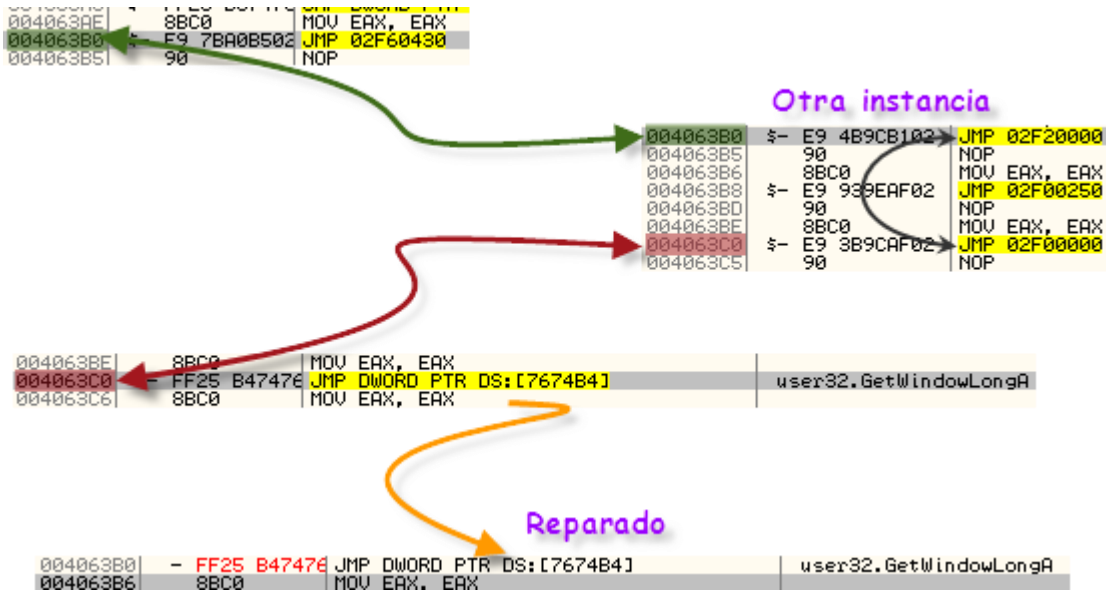
lala:
CMP eip, 00648612
JZ EP
CMP eip, 0045770C
JZ Salir

RUN
JMP lala

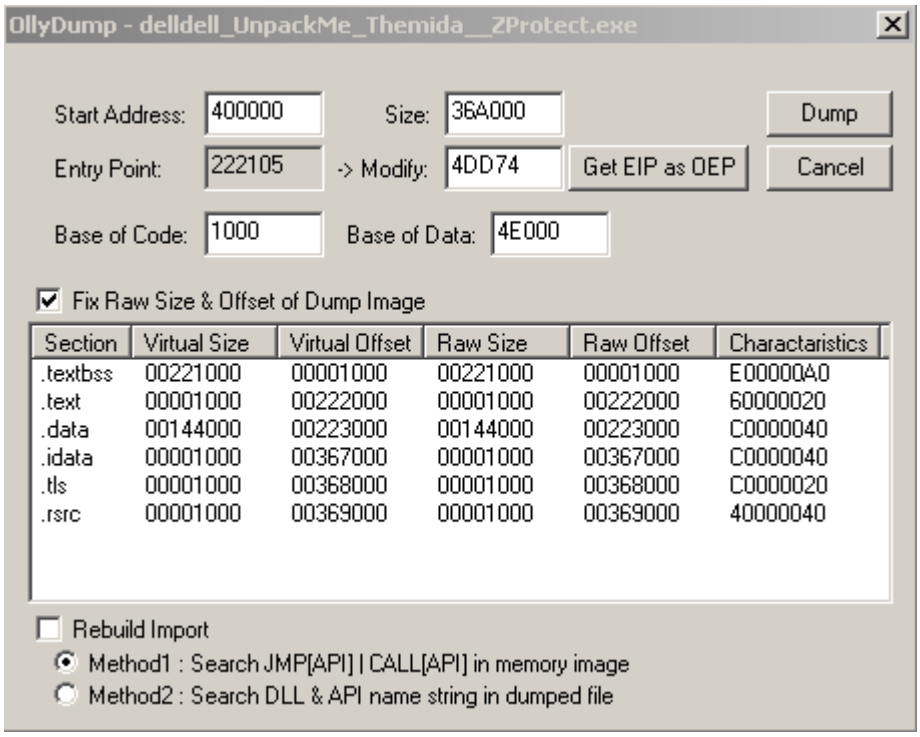
Cuando ejecutamos el script quedan sin resolver dos JMPs, la respuesta es que, al usar la misma API, el packer ya no vuelve a obtener la dirección de la API correspondiente, entonces ya no pasa por el punto mágico, y el script obvia dicha entrada. Para identificar que API saltará el JMP indirecto, podemos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



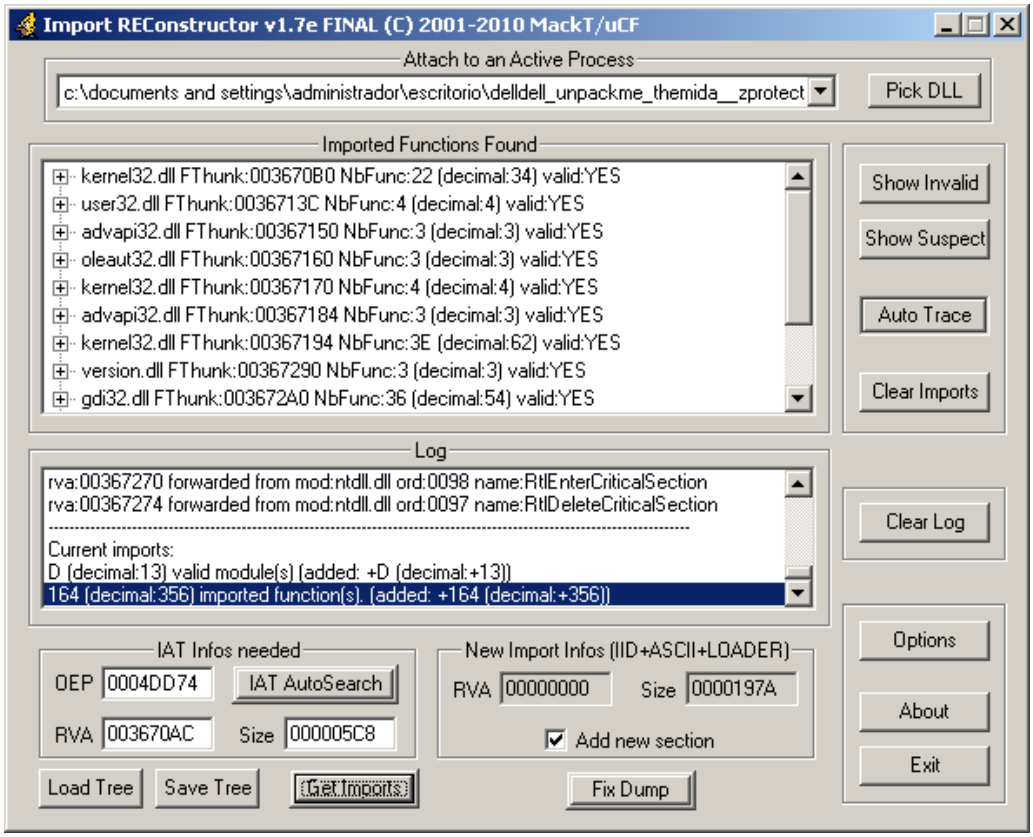
La API es GetWindowThreadProcessId, lo reparamos y pasamos al siguiente JMP no reparado.



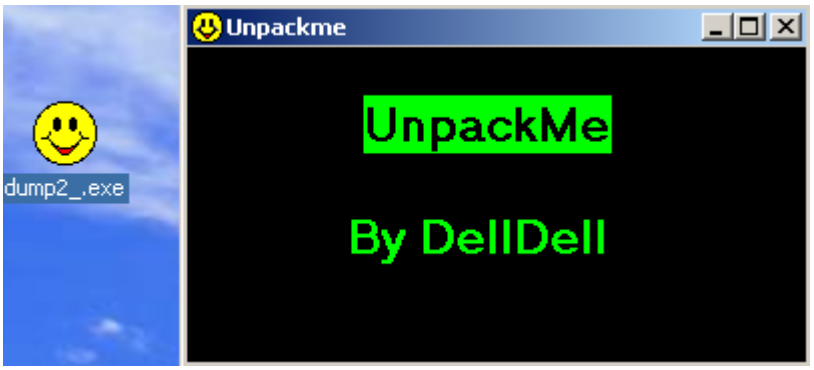
Luego de reparar los dos JMPs pasamos a dumpear como se muestra en la siguiente imagen.



Luego toca reparar la IT, 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.