Inline patching themida 2.1.6.0

March 5

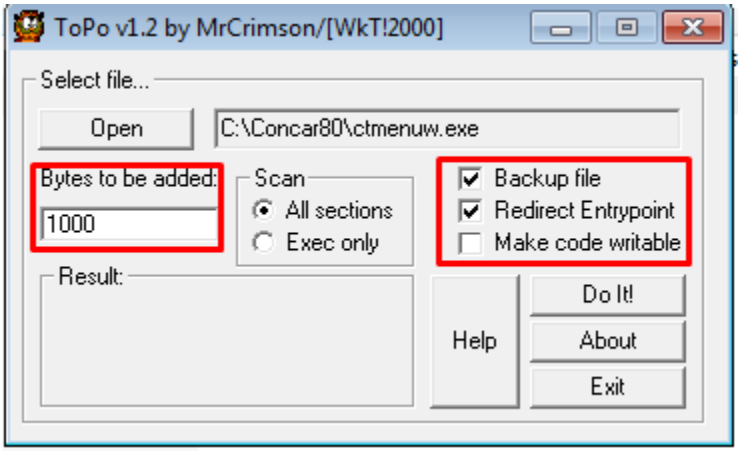
2014

torrescrack.blogspot.com

La idea de este escrito, es mostrar un método que espero e intento pueda ser “general” (sigo trabajando en ello) para aplicar inline-patching a binarios cifrados con themida, hace unas horas mande a ricardo un escrito que postee en el blog de hackingmexico, hablando sobre este mismo tema solo que aplicado en otro software y con menos explicación, le comente que escribiría uno especial para la lista mejor explicado y con mi código adjunto (al final) para hacer algunas pruebas , por si a alguien le sirve.

Hace un tiempo Ivinson escribió acerca de unpack/themida de un software hecho en VB, del cual yo anteriormente había escrito solo que en esos tiempos estaba empacado con visual protect según recuerdo, probé con ese mismo software el método de inline – patching que venía trabajando y vaya que me sirvió para hacer pruebas y mejorar algunos detalles que se me habían pasado, pero no los aburro y vayamos con eso.

Bien lo primero que haremos para inlinear nuestro binario, será crear una sección de código vacia dentro del mismo, existen varias herramientas para ello, pero en este caso usaremos “TOPO”, recuerden que topo creas secciones nuevas y tiene opciones de buscar huecos y usarlos, pero nosotros crearemos una sección nueva, también tildamos algunas casillas que me facilitan el labor:



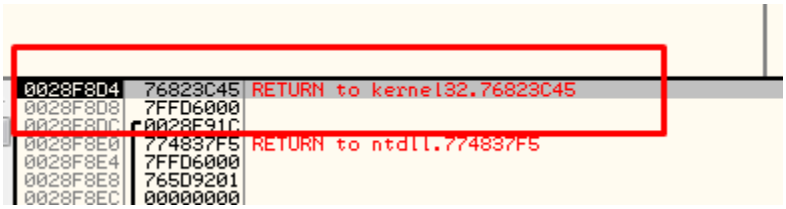
Es esencial tildar que nos guarde un backup por si hacemos algo mal y que nos rediriga el EP para que inicie desde nuestro injerto..

Les quiero comentar que el themida no tenía chequeos CRC, en caso de tenerlos existe un script que parcheo esos chequeos CRC en themida.

Ahora pausemos un rato todo y pasare a explicarles un poco mas de lo que se hará:

Como sabrán, cuando escribimos código dentro de un ejecutable este ya tiene decidido lo que cargara y las API's que usara.. El compilador es el que se encarga de crear una tabla IAT (Import Address Table), que el sistema operativo se encargara de llenar con valores que correspondan a direcciones de las API's; por lo tanto un programa ya compilado tiene su propia tabla lo que nosotros debemos hacer es intentar encontrar la dirección de kernel32.dll, con eso tendríamos al toro por los cuernos.

Comenzaremos y explicare breve y es un método algo viejo y simple; les cuento, al iniciar una aplicación el sistema operativo para cargar un proceso utiliza la API CreateProcess, por lo tanto al cargar el debugger la primer dirección en el stack o pila será de retorno a kernel32.dll, bastara en buscar la dirección donde inicia en memoria la librería y el resto solo es buscar la API GetProcAddress..



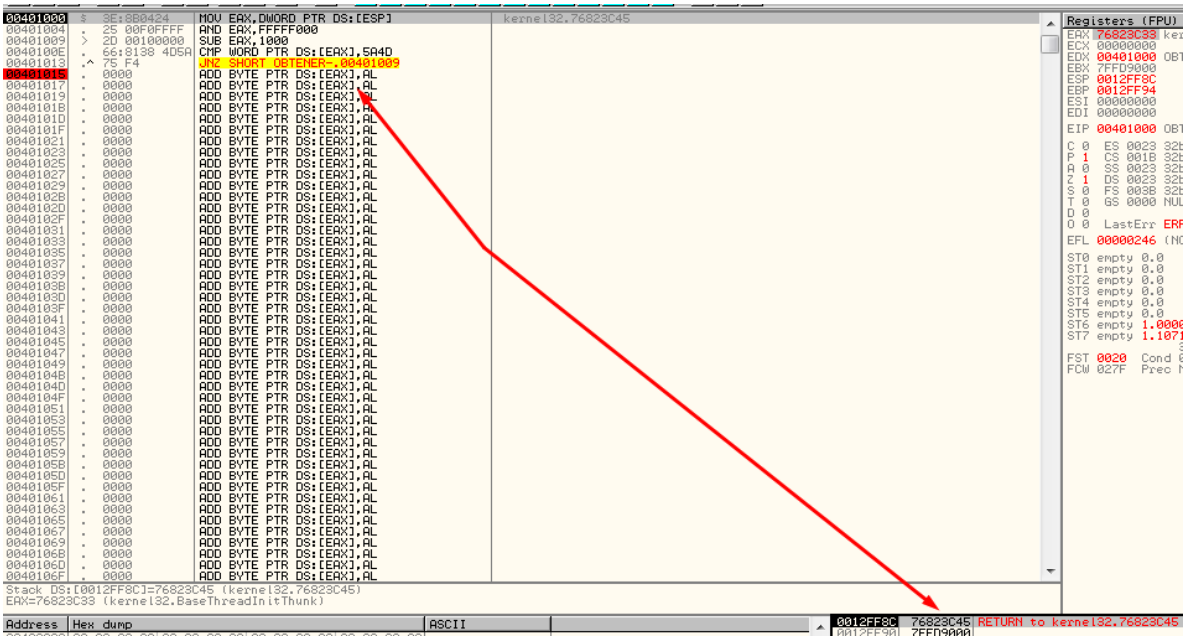
Como muestro al cargar en este caso software dentro del debugger puedo notar que la primer línea del stack es una dirección de retorno a kernel32 que es este caso es 0x76823c45 en otro ordenador las direcciones pueden cambiar.

Ahora bien lo que necesitamos es encontrar el inicio de la librería kernel32, para eso es muy sencillo pues recodemos que dll y .exe tienen cabecera PE por lo tanto inician con "MZ"

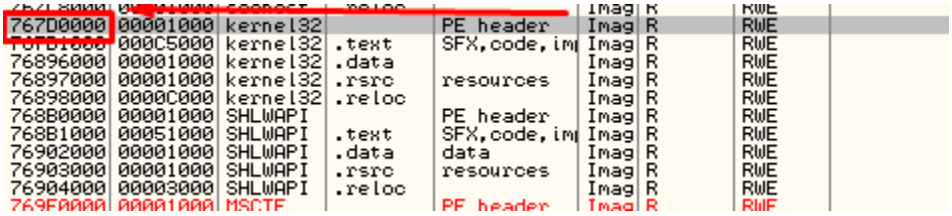
Ahora si abrimos nuestro IDE RADASM y empezamos a escribir este pequeño código:

```
.code


; Facebook:https://www.facebook.com/yo.torrescrack
; twitter: https://twitter.com/TorresCrack248


codigo:


    pushad
    xor ecx,ecx ;limpiamos el registro ecx
    mov eax, dword ptr ds:[esp+20h] ; muevo a "EAX" la direccion de retorno al kernel32

    and eax, 0FFFFF000h ; Esto lo hacemos para descartar los últimos bytes

buscando:
    sub     eax, 1000h  ; vamos restando ya que como todo programa tiene que estar alineado en memoria
    cmp word ptr [eax], 'ZM' ; busca los caracteres "MZ" que es el inicio de la cabecera
    jnz     buscando
```

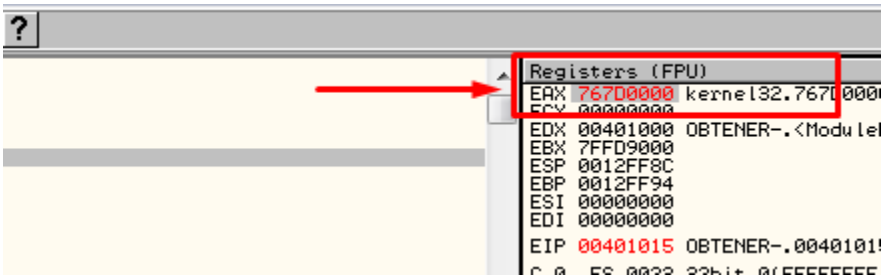
El código esta comentado, pero bueno primero guardamos todo a la pila para después recuperarlo al terminar el meollo, limpiamos el registro ecx y como se desacomodo la pila por el pushad es por eso que movemos esp +20h, después lo que se está haciendo es buscar el inicio de la librería y por ello se descartan los últimos bytes con “and eax, 0fffff000h" y se va restando en 1000h ya que como todo programa debe estar alineado ósea que deben iniciar en una dirección múltiplo de mil, ahora bien podemos probar con olly carguemos nuestro compilado en olly y veamos:



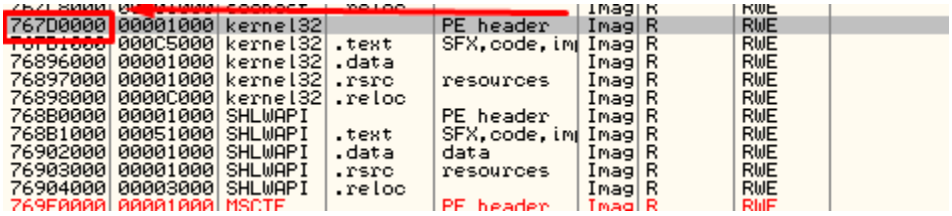
Ahora presionamos el botón “M” (memory map) dentro de nuestro debugger y podremos verificar que kernel32 se cargó en la dirección 767D000:



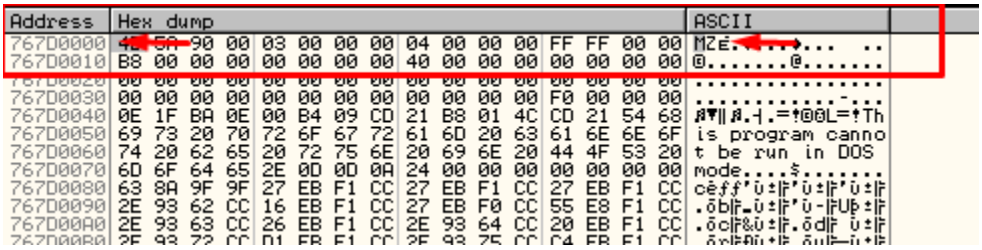
Podemos observar que coloque un BreakPoint al terminar el loop para poder detener el control cuando pare su ejecución al terminar la búsqueda corremos el debugger con f9 y veamos el resultado en "EAX" y veamos si concuerda con la dirección de kernel32:



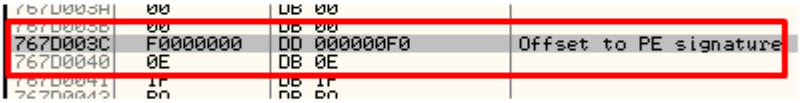
Ahora presionamos el botón "M" (memory map) dentro de nuestro debugger y podremos verificar que kernel32 se cargó en la dirección 767D000:

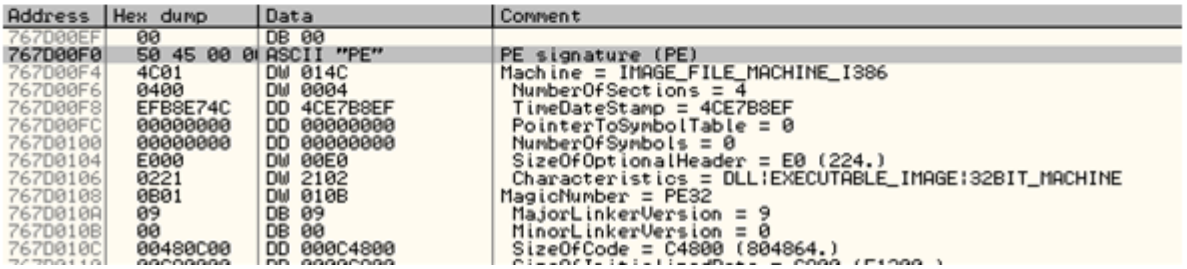


Si vamos a esa dirección de memoria para ver el contenido podemos observar lo que estamos buscando:



Ya que estamos ubicados al inicio de la librería, debemos saber que sabemos que 03Ch lugares mas adelante encontraremos la dirección (RVA) de la cabecera PE. Cambiamos la vista haciendo click derecho y buscamos 03ch lugares mas adelante:





Como podemos observar 03ch lugares mas adelante, vemos nos indica que el offset de la cabecera es + F0 lugares mas delante del inicio de la libreria por lo tanto vamos a esa dirección y vemos el inicio de la cabecera, ahora debemos meter esto en el código conforme vayamos avanzando:

```
;hasta aqui tenemos la direccion de la libreria ahora vamos por busca la API

mov ebx,eax ; guarda el inicio de la libreria en el registro ebx
mov edi,eax ; guarda el inicio de la libreria en el registro edi
add ecx,03ch ; coloca en el registro ecx el valor 03ch para posteriormente sumarlo
add ebx,ecx
mov ebx, [ebx]
add eax,ebx ; sumamos el contenido para poder llegar a la cabecera
```

En los comentarios del código explico todo, que básicamente es lo que hicimos a mano pero plasmado en el código

lo que nos interesa es la tabla de exportación lo cual está ubicada siempre a 78h a partir del "PE" podríamos ver:

0000	Dword	Characteristics	Set to zero (currently none defined)
0004	Dword	TimeDateStamp	Often set to zero
0008	Word	MajorVersion	User-defined version number
000A	Word	MinorVersion	As above
000C	Dword	Name	RVA of DLL name (null-terminated ASCII)
0010	Dword	Base	First valid exported ordinal
0014	Dword	NumberOfFunctions	Number of entries in EAT
0018	Dword	NumberOfNames	Number of entries in ENT

001C	Dword	AddressOfFunctions	RVA of EAT (export address table)
0020	Dword	AddressOfNames	RVA of ENT (export name table)
0024	Dword	AddressOfNameOrdinals	RVA of EOT (export ordinal table)

Bien, ya estamos en la tabla de exportaciones de kernel32.dll, ahora veamos cómo está compuesta:

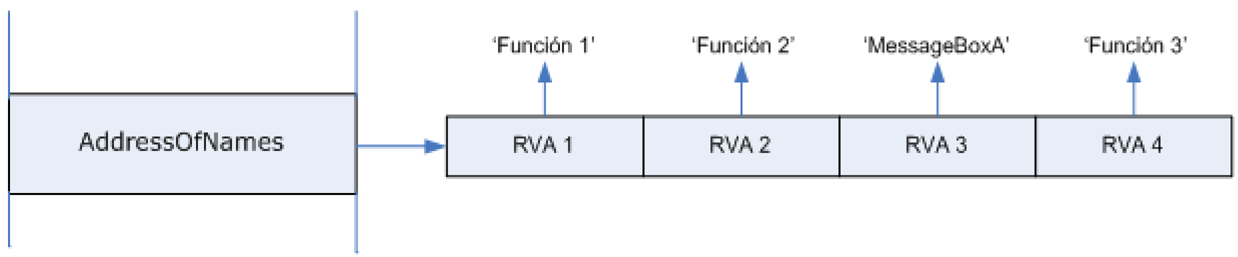
:::

(Trozo tomado de un tutorial de zero)

De esta estructura interesan las direcciones de las últimas tres tablas:

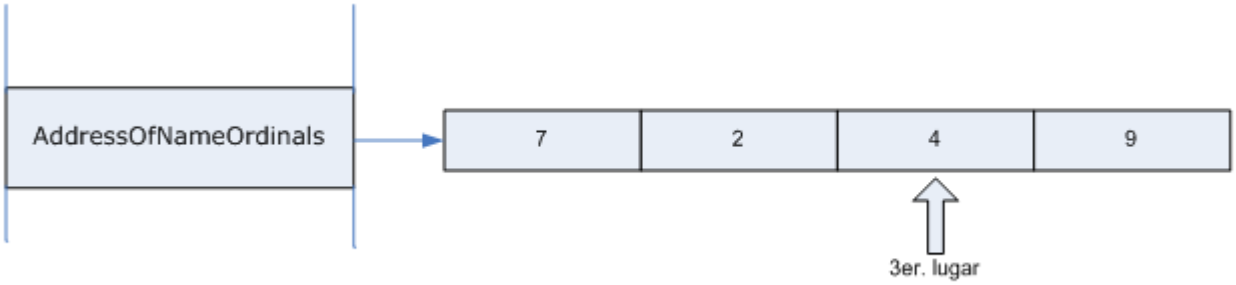
- *AddressOfFunctions: contiene la dirección (RVA) de un listado de punteros (RVA's) a las funciones exportadas por esta librería*
- *AddressOfNames: contiene la dirección (RVA) de un listado de punteros dirigidos a los nombres de las funciones exportadas*
- *AddressOfNameOrdinals: RVA que apunta a un listado que contiene los ordinales de los nombres de las funciones exportadas*

Tenemos que empezar por la tabla AddressOfNames, la cual contiene un listado de punteros a direcciones de memoria (RVA's) donde están guardados lo nombres de las funciones exportadas:

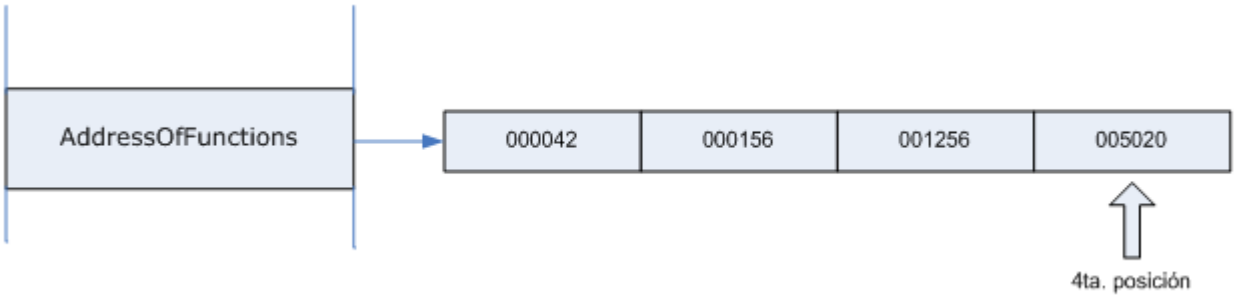


Dos acotaciones, AddressOfNames contiene una RVA que apunta a un listado de RVA's, no directamente a los nombres. Además recordemos que cada una de estas RVA tiene un tamaño double word.

*Bien, vemos que la **tercer RVA** es la que apunta a la función buscada, ese valor nos va a servir de puntero para la próxima tabla, la AddressOfNameOrdinals:*



*En el **tercer lugar** de esta tabla tenemos un valor (en este caso hipotético es 4) que nos servirá de puntero a la última tabla, la AddressOfFunctions:*

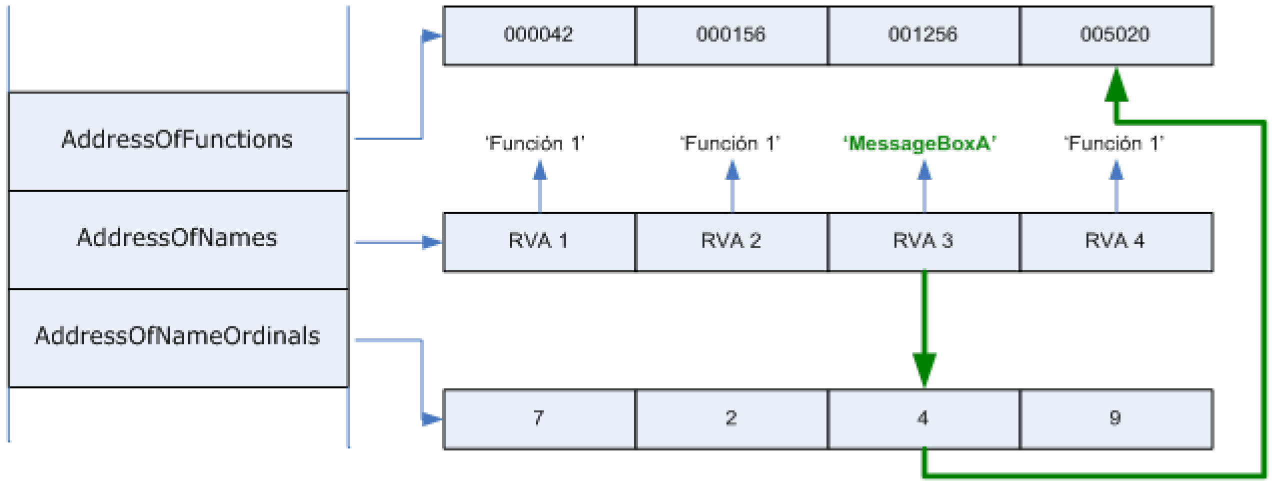


En el caso de esta tabla, los valores son del tipo Word, esto también lo vamos a necesitar luego cuando realicemos nuestro código.

Vemos que en la cuarta posición tenemos el valor 005020, esto nos indica la RVA donde vamos a encontrar el inicio de la función buscada en memoria, para que podamos hacer algo como esto:

```
call    405020
```

Recordemos que es una RVA, por lo que tendremos que sumarle la dirección base antes de utilizarla.
Para que quede un poco mas claro, veamos todo esto en un solo paso:



Es un esquema un poco complejo, pero una vez que lo analicemos y luego cuando veamos el código va a quedar mas claro.

::

Primero:

```
; edi se queda con el valor de inicio de la libreria

xor ebx,ebx
mov ecx, 078h
add eax,ecx ; sumamos desde la direccion el PE + 78h para encontrar la ExportTable
mov ebx, [eax] ; sacamos el RVA y lo colocamos en ebx
add ebx, edi ; sumamos la imagebase  de la libreria con el RVA del ExportTable
mov eax, [ebx+20h] ; segun la tabla que mostre sacamos el RVA de AddressOfNames
add eax, edi ; aqui sumamos y obtenemos el offset de AddressOfNames
push ebx; pone en la pila el inicio de la exportable que despues usaremos
push edi ; se pone en la pila el contenido de EDI que posteriormente usaremos (inicio de libreria)


; eax contiene el offset de AddressOfNames
```

Como vemos el código está muy comentado, básicamente lo que se hace es sumar la dirección del PE + 78 que es donde se encuentra la ExportTable, obtenemos el RVA y lo sumamos con 20h que según la descripción con la tabla de arriba es la dirección de AddressOfNames, y

guardamos datos en la pila para posteriormente usarlos, lo hacemos de esta manera pues quiero que se adapte para solo copiar y pegar.. dentro del debugger (ya lo verán mas adelante)

Ahora vamos a ir recorriendo esa tabla y vamos a buscar la API 'GetProcAddress' por su nombre, para eso realizamos la siguiente rutina:

```
; eax contiene el offset de AddressOfNames

xor ebx,ebx ; preparamos nuestro contador
sub ebx,4

xor ebp,ebp ; ponemos a cero el segundo contador


BuscadorGPA:


inc ebp ; un tipo de contador (guia) que utilizaremos al entrar en las otras tablas..
add ebx,4
mov edx, [eax+ebx] ; ira toma el contenido de eax + el contador, recuerden que en eax esta el offset de AddresOfNames
pop edi ; se toma de la pila lo que edi contenia (Offset kernel32)
add edx, edi ;
push edi
mov ecx,0Eh
mov esi,edx ; mueve a ESI el puntero del primero nombre de la lista de API's
lea edi, offset GPA ; mueve a EDI el offset donde se encuentra la API que buscaremos
cld
repe cmpsb ; realiza las comprobaciones byte a byte hasta encontrar el nombre GetProcAddress
jnz BuscadorGPA
```

Preparamos dos contadores poniéndolos a cero, lo primero que hará para ir sumando cada 4 bytes y buscar el nombre de cada api, el segundo (EBP) es para tener un índice para posteriormente usarla en la siguientes tablas; lo que se hace en esta parte del código que igual esta muy comentado para que no se queden con dudas, pero les explico, básicamente lo que se hace es comparar cada 4 bytes dentro de AddressOfNames el nombre de la API "GetProcAddress" lo cual lo hace por medio de la instrucción *cmpsb*. Esta instrucción sirve para comparar cadenas de un byte, donde la primer cadena la tenemos que tener referenciada con el registro esi y la segunda con el registro edi. (se muestra en el código como se colocan en edi y esi lo correspondiente)

Por ejemplo si queremos comparar una cadena 'C' con otra 'D', tenemos que tener lo siguiente:

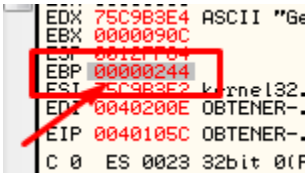
La vamos a utilizar para comparar byte a byte lo que tenemos en memoria con una variable que definimos con el nombre de la API buscada, de la forma:

GPA db “GetProcAddress” ,0

Como solo compara un byte la tenemos que ponerla dentro de otra instrucción: rep (o repe), la cual repite la instrucción que le sigue la cantidad de veces que le indiquemos en el registro ecx. Por eso debemos inicializar a ecx con el largo de la cadena a comparar, en este caso GetProcAddress tiene 14 bytes, o sea que inicializamos a ecx con el valor 0Eh (14 en hexadecimal).

De toda la rutina anterior lo único que nos va a importar es justamente el contador que esta ubicado en el registro EBP, el cual vamos a utilizar como índice para la próxima tabla.

Si seguimos esta rutina dentro de ollydebugger al finalizar la rutina podríamos observar que el registro EBP (nuestro índice) guardo lo que queremos:



Ya casi acabamos. Lo que haremos ahora es obtener la dirección de la próxima tabla (AddressOfNameOrdinals), que está a 024h a partir del inicio de la tabla de exportaciones:

```
pop edi
pop ebx

dec ebp ; acomodo el contador (guia)

; ebx contiene el inicio del exportable
; edi contiene el inicio del kernel32

; hacemos lo mismo que anteriomente solo que ahora buscando la direccion de AddressOfNameOrdinals

xor ecx,ecx
mov eax, [ebx+24h]
add eax,edi


mov ecx,ebp
imul ecx,2 ; multiplico por 2

add eax,ecx


; eax esta valiendo el valor sacado de AddressOfNameOrdinals
```

Recuperamos con POP edi,ebx los datos guardados anteriormente en la PILA, sacamos el RVA de AddresaOfNamesOrdinals y lo sumamos con EDI que contiene el inicio del kernel32 y asi obtenemos el offset de AddressOfNameOrdinals.

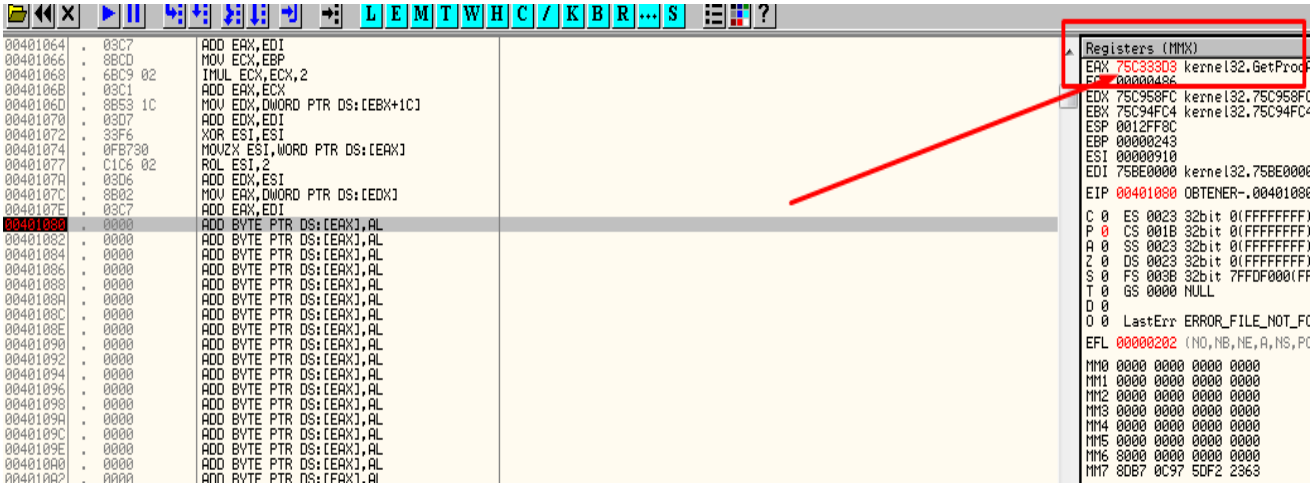
Ahora prepararemos el índice para acceder al valor de la tabla, para eso tenemos que multiplicar el contador EBP por dos, esto es porque cada dirección de esta tabla ocupa 2 bytes, o sea que por ejemplo el cuarto valor de esta tabla lo vamos a encontrar en la posición 4 * 2.

Bien, ahora vamos con el final:

```
; hacemos lo mismo que anteriomente solo que ahora buscando la direccion de AddressOfFunctions

mov edx, [ebx+01ch]
add edx,edi
xor esi,esi
movzx esi, word ptr [eax]
rol esi,2 ; multiplico por 4
add edx,esi  ; normalizo el RVA

mov eax,[edx]
add eax,edi

; eax tiene la direccion de la API GetProcAddress "}:)"
```

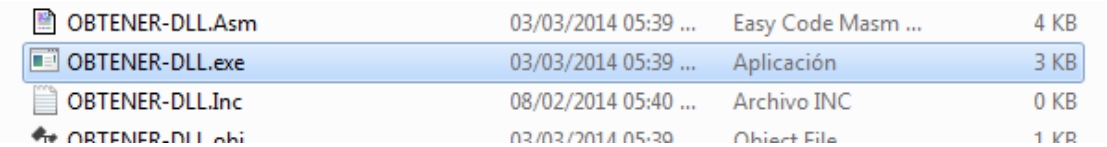
Es casi lo mismo que las anteriores rutinas, le sumamos a 01ch que es donde está el RVA de AddressOfFunctions y le sumamos el inicio del kernel32 que está en EDI y luego muevo el valor del puntero que obtuvimos anteriormente (eax) a esi y lo multiplico por cuatro. Esto es similar a lo que vimos anteriormente, pero como ahora la tabla es de valores doubleword, debo multiplicar por cuatro.

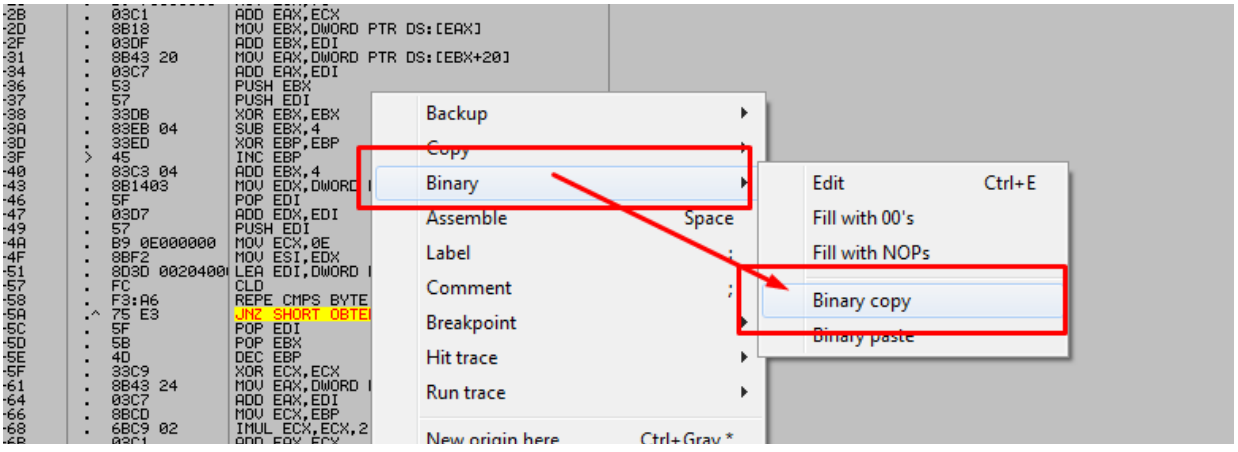
En edx tenía la dirección obtenida, como es un puntero a otra dirección, lo que hago es obtener el contenido de esa dirección y luego 'normalizarla'. Al finalizar eax contendrá la dirección de la API que tanto buscábamos 😀



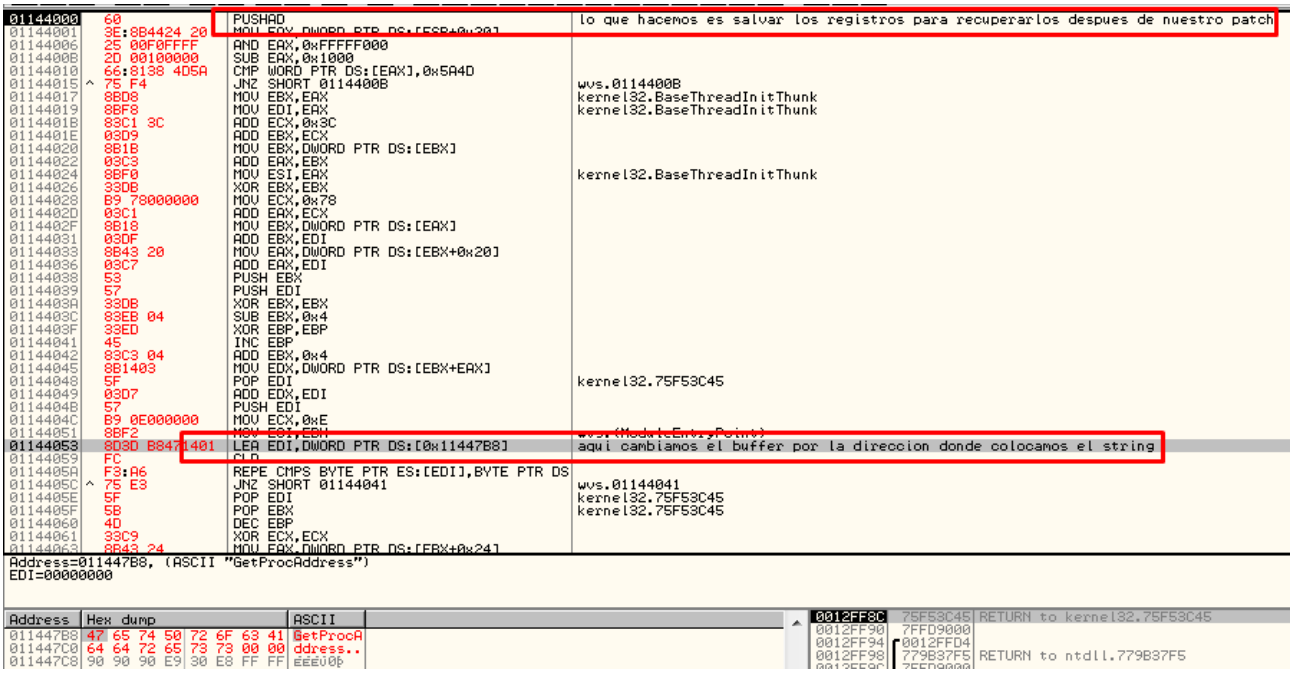
Algunos se preguntaran porque me complique tanto la vida si pudo ser mas corto el codigo, pues es que la idea era que tuviéramos un codigo de tal forma que solo haciendo cambios minimos podríamos utiliarlo en cualquier otro inline a themida.

los únicos cambios que deberemos hacer al copiar y pegar el código es cambiar la dirección donde se encuentra el string “GetProcAddress” como muestro adelante:

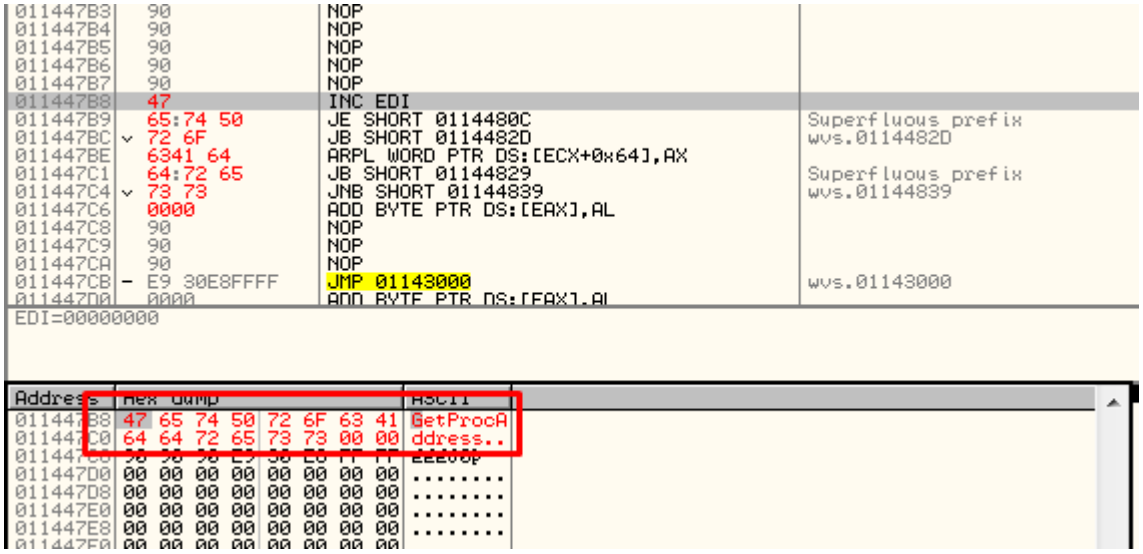




Copiamos el código, ahora vamos a abrir el software en otro ollydebugger y pegamos el código en la zona creada anteriormente:



Bajamos y casi al final del código injertado podremos ver el jmp que nos agregó topo para redirigir a la zona donde estaba el punto de entrada y como no usaremos esa parte metemos el string “GetProcAddress” en esa dirección y modificamos muestra la imagen de arriba.



Bien ahora guardamos los cambios y listo empecemos con lo demás que es la parte interesante, como sabrán este código lo que hace es que al terminar nos devuelva el offset de "GetProcAddress" y lo que hace esa API es devolvernos la dirección de cualquier API que le pidamos, con eso podemos sacar mas api's, que en este caso la única que quiero es "CreateThread" que básicamente nos sirve para crear un hilo y se preguntaran ¿que haremos con ese hilo?, sencillo un hilo que vaya buscando los bytes ya descifrados en memoria (pues recordemos está cifrado) conforme sigue la ejecución del software y ya una vez encontrados procedemos a parchear, algo similar lo explique en el código de un loader simple, que mostré en el "elladodelmal", según recuerdo;

http://www.elladodelmal.com/2014/02/loader-simple-acceder-al-binario.html

veamos la información de los parámetros que necesitamos meter para usar "GetProcAddress":

C++

```
FARPROC WINAPI GetProcAddress(
  _In_  HMODULE hModule,
  _In_  LPCSTR lpProcName
);
```

Parameters

hModule [in]

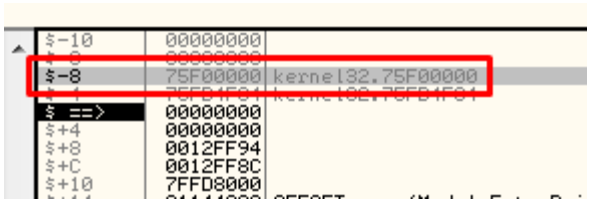
A handle to the DLL module that contains the function or variable. The **LoadLibrary**, **LoadLibraryEx**, **LoadPackagedLibrary**, or **GetModuleHandle** function returns this handle.

The **GetProcAddress** function does not retrieve addresses from modules that were loaded using the **LOAD_LIBRARY_AS_DATAFILE** flag. For more information, see **LoadLibraryEx**.

lpProcName [in]

The function or variable name, or the function's ordinal value. If this parameter is an ordinal value, it must be in the low-order word; the high-order word must be zero.

Solo son dos parámetros, el primero es la dirección de la dll que contiene a la API que en este caso es kernel32.dll lo cual esa dirección ya la tengo por algún lado del stack guardada, el segundo parámetro es el nombre de la API que esa la ubicaremos cerca de donde ubicamos el nombre de "GetProcAddress" veamos cómo quedaría:




```
0114407C   03D6            ADD EDX,ESI
0114407E   8B02            MOV EAX,DWORD PTR DS:[EDX]
01144080   03C7            ADD EAX,EDI                     kernel32.75F00000
01144082   8B4C24 F8       MOV ECX,DWORD PTR SS:[ESP-0x8]  kernel32.75F00000
01144086   68 90471401     PUSH 0x1144790                  ASCII "CreateThread"
0114408B   51              PUSH ECX                        kernel32.75F00000
0114408C   FFD0            CALL EAX                        kernel32.GetProcAddress
0114408E   90              NOP
0114408F   90              NOP
01144090   90              NOP
01144091   90              NOP
```

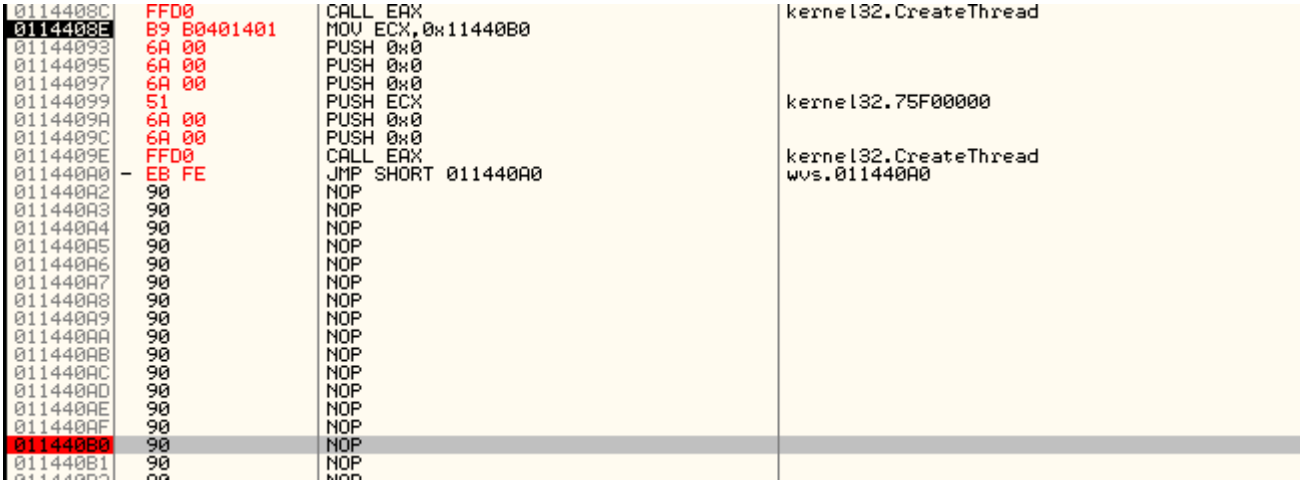
Aquí movemos al registro ECX la dirección del kernel32 que como vemos en la pila quedo ubicado en ESP-8, después metemos los parámetro que ya había comentado y hacemos una llamada a EAX que contiene la dirección de GetProcAddress, si pasamos esas instrucciones podemos observar que el resultado está en el registro EAX devolviéndonos la dirección de la API que necesitamos:

```
0114404B   57              PUSH EDI                        kernel32.75F00000
0114404C   B9 0E000000     MOV ECX,0xE
01144051   8BF2            MOV ESI,EDX                     kernel32.75F00000
01144053   8D3D B8471401   LEA EDI,DWORD PTR DS:[0x11447B8] aqui cambiamos el buffer por la direccion donde colo
01144059   FC              CLD
0114405A   F3:A6           REPE CMPS BYTE PTR ES:[EDI],BYTE PTR D
0114405C ^ 75 E3           JNZ SHORT 01144041              wvs.01144041
0114405E   5F              POP EDI                         kernel32.75F00000
0114405F   5B              POP EBX                         kernel32.75FB4FC4
01144060   4D              DEC EBP
01144061   33C9            XOR ECX,ECX                     kernel32.75F00000
01144063   8B43 24         MOV EAX,DWORD PTR DS:[EBX+0x24]
01144066   03C7            ADD EAX,EDI                     kernel32.75F00000
01144068   8BCD            MOV ECX,EBP
0114406A   6BC9 02         IMUL ECX,ECX,0x2                kernel32.75F00000
0114406D   03C1            ADD EAX,ECX                     kernel32.75F00000
0114406F   8B53 1C         MOV EDX,DWORD PTR DS:[EBX+0x1C]
01144072   03D7            ADD EDX,EDI                     kernel32.75F00000
01144074   33F6            XOR ESI,ESI
01144076   0FB730          MOVZX ESI,WORD PTR DS:[EAX]
01144079   C1C6 02         ROL ESI,0x2
0114407C   03D6            ADD EDX,ESI
0114407E   8B02            MOV EAX,DWORD PTR DS:[EDX]      wvs.00905A4D
01144080   03C7            ADD EAX,EDI                     kernel32.75F00000
01144082   8B4C24 F8       MOV ECX,DWORD PTR SS:[ESP-0x8]  kernel32.75F00000
01144086   68 90471401     PUSH 0x1144790                  ASCII "CreateThread"
0114408B   51              PUSH ECX                        kernel32.75F00000
0114408C   FFD0            CALL EAX                        kernel32.CreateThread
0114408E   90              NOP
0114408F   90              NOP
01144090   90              NOP
01144091   90              NOP
```

Vaya ahora bien en código les explico los parámetros que debemos meter como lo hicimos anteriormente pero ahora para cargar nuestro hilo:

```
mov eax, offset proced
invoke CreateThread,NULL,NULL,eax,NULL,NULL,NULL
```

Todo es “NULL” o “0” y lo único que debe importarnos es ubicar en algún registro la dirección donde queremos poner nuestro hilo y quedaría algo así en el código injertado:

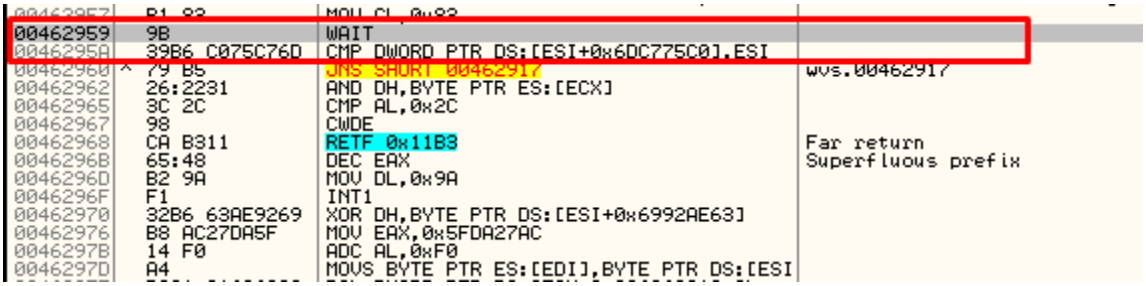


Como verán meti en el registro ECX un offset que será donde ubicaremos nuestro loop para buscar código y será nuestro hilo… (ignoren el ultimo "JMP" lo ubique para probar si el hilo era tomado) por lo tanto con esto avanzado podríamos recuperar los valores de la pila a como estaban con la instrucción POPAD (recuerden que al inicio colocamos un pushad para salvar los registro en la pila) y regresar a donde originalmente iniciaba el programa y tendríamos un hilo corriendo en paralelo buscando los bytes a parchear que adelante mostrare, mientras pongamos el código restante para devolver la ejecución al inicio del programa como normalmente comenzaría:

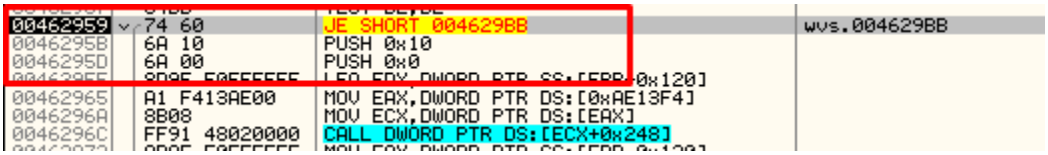
Recuperamos los registros y ponemos un salto a donde iniciaba el programa antes de agregar la sección con topo, antes de guardar los cambios que ya hemos hecho, pasemos al offset donde se encontraría el hilo y metamos el código, pero antes recordemos algo en qué dirección se ubicaba el salto que queríamos cambiar (no tocare detalles de crackeo del software) , pero quiero mostrarles como luce cuando aún no es ejecutado el programa y el código está cifrado:

(ESTA NO ES LA LIMITACION DEL SOFTWARE, ES UN EJEMPLO)

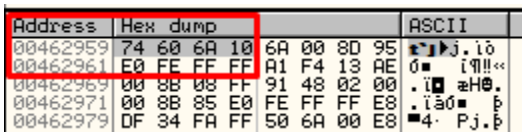
Cifrado:



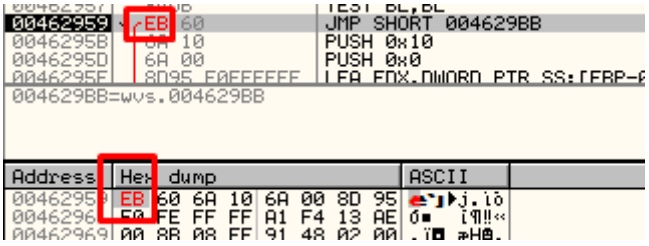
Descifrado:



Ahora bien, los bytes descifrados son los siguientes y por lo tanto los que deberíamos buscar en nuestro loop son los siguientes:

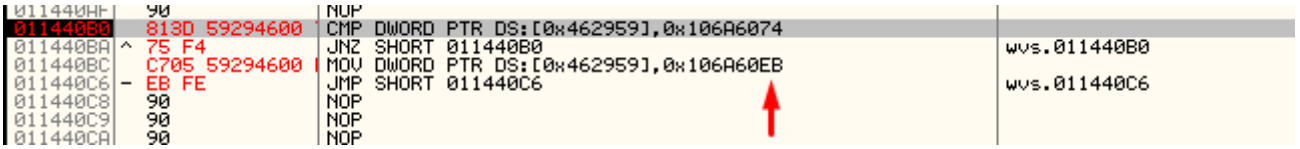


Y cambiarlos por lo que muestro a continuación:



Como ven se marcan en rojo los cambios que hice cambie el salto JE (condicional) por un JMP para que salte siempre sin excepción, solo cambia un byte (EB) con el parche que queremos colocar, por lo tanto una vez encontrado en memoria cuando se ha descifrado el programa parchearemos ese byte y colocaremos “EB” y listo.

Ahora escribamos el código siguiente:



Como vimos hace una búsqueda de los bytes y sigue repitiéndose el loop hasta que los encuentra y pasa guardar lo bytes nuevos en esa dirección que como verán solo cambia el "EB" y mas abajo coloque un loop infinito (solo para probar lo que ya hemos hecho, si dejamos esto asi el loop infinito consumiría muchos recursos) podríamos hacer mas cosas como cerrar el hilo, poner un sleep o demas cosas, el tema es que ya terminamos de hacer toda la máquina de un inline patching y esto era básicamente lo que quería mostrarles! Y ojala que les sirva de algo esto ya que me tomo muchas horas armarlo ☺.

aquí les dejo el proyecto: http://uppit.com/4r8bnmh4fxrc/OBTENER-DLL.rar

Antes de terminar, les cuento que hace un rato hable con guan de dio y le platique la idea de hacer una tool con un método "general" para hacer inline patching a themida, osea automatizar todo esto que hice a mano en una herramienta que básicamente haga lo siguiente: nos pida offset y bytes nuevos a colocar en la dirección y listo haría todo lo que tarde horas en 2 segundos, creo que podría ser una excelente herramienta para inlinear a themida, cabe mencionar que ya he probado el código con diferentes sistemas operativos y repare detalles que causaban problemas, sin mas les mando un saludo y si alguien se quiere unir al proyecto espero sus comentarios! , saludos a todos!

Alejandro (torrescrack)

Tora_248@hotmail.com