Capítulo 16

Otro método para encontrar kernel32.dll

Funciones hash

En el capítulo anterior comenzamos a mejorar nuestro virus haciéndolo menos visible gracias a la encriptación, ahora vamos a ver algunas mejoras en cuanto a la búsqueda de la librería kernel32.dll.

La idea que utilizamos en el capítulo 6 es conocida como "Topstack" ya que justamente lo que hicimos fué recuperar la dirección que se encuentra en el tope de la pila y que corresponde a una dirección de retorno de kernel32.dll, a partir de allí buscamos el inicio de la misma en memoria y luego las funciones que exporta y así llegabamos a GetProcAddress. A partir de ahí podíamos obtener todas las funciones que necesitábamos.

Lo que vamos a variar en esta oportunidad es justamente la forma de encontrar la dirección de inicio de kernell32.dll, el resto por ahora lo vamos a dejar igual.

Arranquemos, en el capítulo 2 veíamos que para cada thread o hilo de ejecución, el SO lleva una tabla llamada TIB (Thread Information Block), también conocida como TEB (Thread Environment Block). Con esta tabla podemos obtener mucha información importante sobre el hilo del proceso, por ejemplo la primer entrada del SEH chain (como vimos en el capítulo 6), el ID del proceso, el número del último error encontrado, etc., y todo esto sin necesidad de llamar a ninguna API de Windows.

Dentro de esta estructura se encuentra un puntero a otra llamada PEB (Process Environment Block), que también contiene información muy interesante por lo que vamos a verla un poco más en profundidad ahora y además nos va a dar pié más adelante cuando empecemos a ver algunas protecciones anti-debugging.

Para aclarar un poco las cosas, cuando tenemos un archivo en la computadora es solamente un conjunto de bytes, pero si lo ejecutamos se tranforma en un proceso (esto es simplemente un programa en ejecución), y este proceso va a tener uno o más hilos de ejecución (por lo menos uno). Gracias a los hilos (threads) un programa puede hacer varias tareas a la vez.

Cuando vimos SEH accedíamos al TIB a través del registro fs, haciendo algo como mov eax, fs:[0] que nos devolvía el primer valor de esta estructura. Lo que nos interesa ahora es la dirección del PEB, que se encuentra en el desplazamiento 30h, asi que para obtener la dirección de inicio del mismo haríamos algo así:

```
---------------------------------------------------------------------------------------------
   mov     eax, fs:[030h]
---------------------------------------------------------------------------------------------
```

Bien, ya tenemos la dirección del PEB, ahora veamos como está compuesta:

```
---------------------------------------------------------------------------------------------
typedef struct _PEB {
  UCHAR               InheritedAddressSpace;
  UCHAR               ReadImageFileExecOptions;
  UCHAR               BeingDebugged;
```

```
  UCHAR           BitField;
  ULONG           ImageUsesLargePages: 1;
  ULONG           IsProtectedProcess: 1;
  ULONG           IsLegacyProcess: 1;
  ULONG           IsImageDynamicallyRelocated: 1;
  ULONG           SpareBits: 4;
  PVOID           Mutant;
  PVOID           ImageBaseAddress;
  PPEB_LDR_DATA Ldr; ←desplazamiento: 0Ch
  PRTL_USER_PROCESS_PARAMETERS ProcessParameters;
  …….
  …….
} PEB, *PPEB;
```

Si vemos el Segundo byte, vemos que dice *BeingDebugged*, y es justamente un flag que nos indica si el proceso está siendo depurado, es lo que devuelve la API IsDebuggerPresent, pero sin necesidad de llamar a ninguna function.

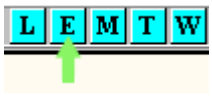
Lo que nos interesa en este momento de esta estructura es el puntero a otra estructura (si, otra más y todavía faltan…) llamada PPEB_LDR_DATA, que se encuentra desplazada 0Ch lugares desde el inicio del PEB.

Veamos como está compuesta esta nueva estructura:

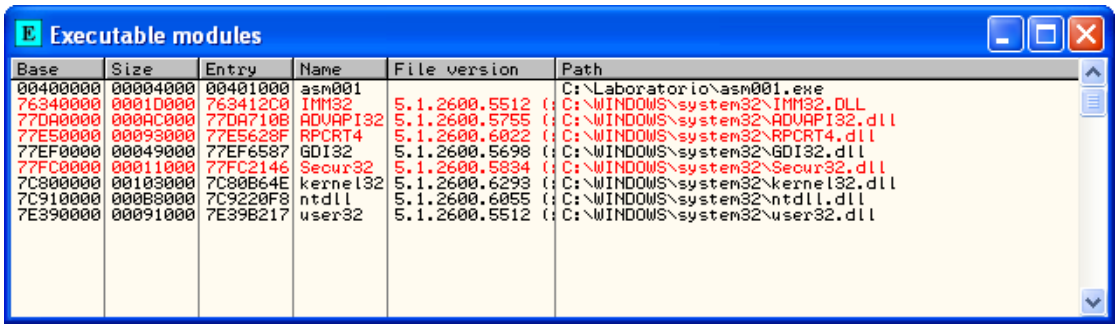
```
typedef struct _PEB_LDR_DATA
{
    ULONG Length;
    UCHAR Initialized;
    PVOID SsHandle;
    LIST_ENTRY InLoadOrderModuleList; ←desplazamiento: 0Ch
    LIST_ENTRY InMemoryOrderModuleList;
    LIST_ENTRY InInitializationOrderModuleList;
    PVOID EntryInProgress;
} PEB_LDR_DATA, *PPEB_LDR_DATA;
```

Además vemos en la ayuda de la MSDN (http://msdn.microsoft.com) que esta estructura contiene información acerca de los módulos cargados por el proceso. En el capítulo 2 veíamos que los módulos de un proceso son los recursos que necesita el programa para ser ejecutado, y son obviamente el propio programa, pero además todas las librerías que necesita y justamente una de estas librerías es kernel32.dll que es lo que estamos buscando.

Veamos esto un poco en OllyDBG para que vaya quedando bien claro. Abramos nuestro programa de ejemplo que siempre utilizamos (el que muestra un mensaje por pantalla) y presionemos el icono 'Show modules' (E):



Vamos a ver algo como esto:



Y ahí vemos lo que queremos encontrar, la librería kernel32.dll y su dirección de memoria, en este caso 7C800000. Ok, volvamos a la estructura que teníamos.

De esta estructura nos interesa **InLoadOrderModuleList**, que es una lista doblemente enlazada que contiene punteros a los módulos cargados para el proceso en ejecución. Cada elemento de la lista es un puntero a una estructura LDR_DATA_TABLE_ENTRY.

Veamos como está compuesta:

```
------------------------------------------------------------------------------------------------------
typedef struct _LIST_ENTRY {
    LIST_ENTRY Flink;
    LIST_ENTRY Blink;
} LIST_ENTRY, *PLIST_ENTRY;
------------------------------------------------------------------------------------------------------
```

Como decíamos es una lista doblemente enlazada ya que tiene un puntero a la siguiente entrada (**Flink**), y otro a la anterior (**Blink**).

Cada uno de estos punteros tiene la dirección a otra estructura llamada LDR_DATA_TABLE_ENTRY, que tiene el siguiente formato:

```
------------------------------------------------------------------------------------------------------
typedef struct _LDR_DATA_TABLE_ENTRY {
   LIST_ENTRY InLoadOrderLinks; ←desplazamiento: 00h
   LIST_ENTRY InMemoryOrderLinks;
   LIST_ENTRY InInitializationOrderLinks;
   PVOID DllBase; ←desplazamiento: 018h
   PVOID EntryPoint;
   ULONG SizeOfImage;
   UNICODE_STRING FullDllName;
   UNICODE_STRING BaseDllName; ←desplazam.:02Ch (sumar 04h por que es UNICODE)
   ULONG Flags;
   WORD LoadCount;
   WORD TlsIndex;
  ……..
  ……..
} LDR_DATA_TABLE_ENTRY, *PLDR_DATA_TABLE_ENTRY;
------------------------------------------------------------------------------------------------------
```

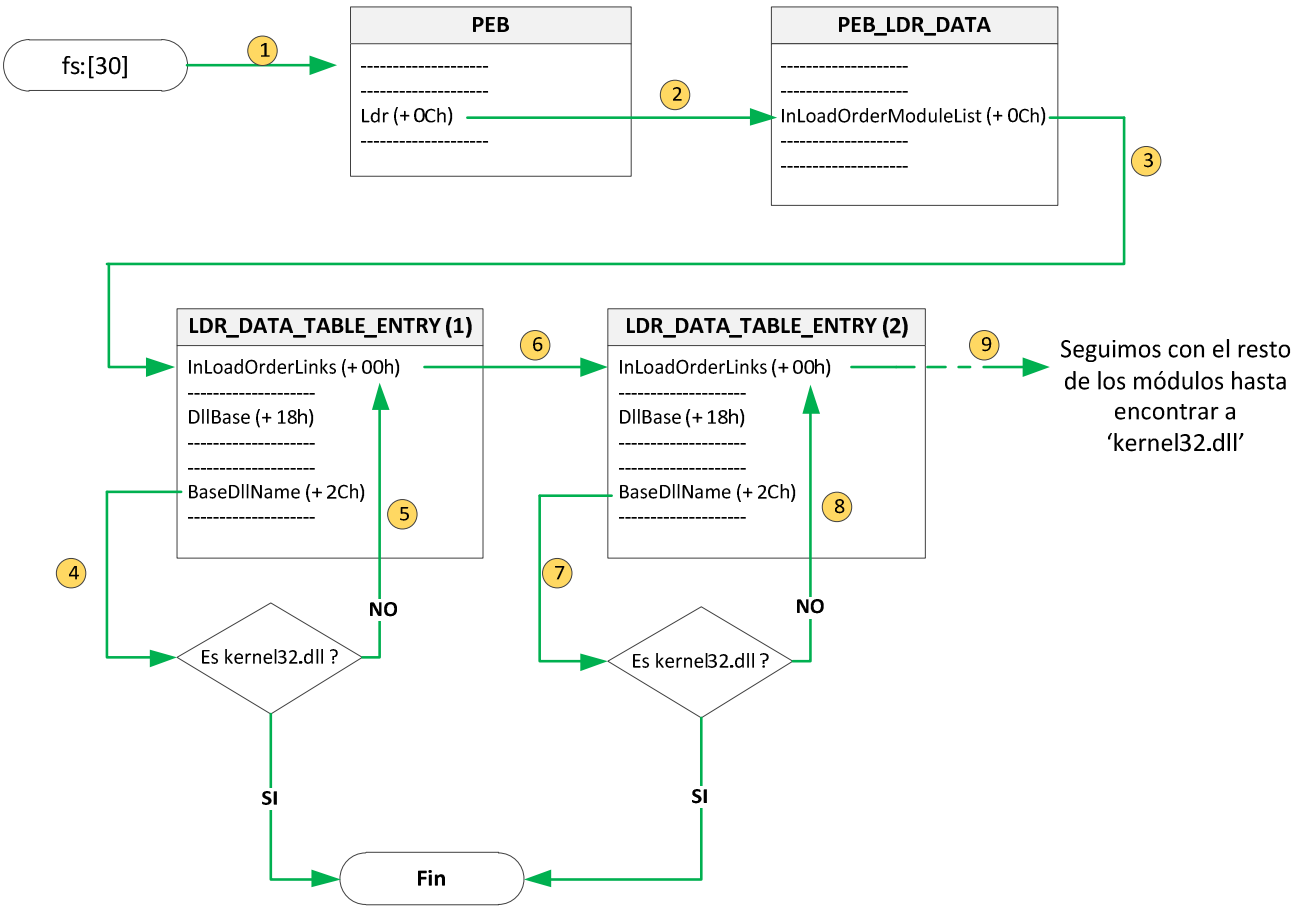
Esta estructura es la que contiene los datos sobre los módulos cargados por el proceso, y nos permite recorrer los módulos cargados de tres formas distintas:

1. Orden de carga -> InLoadOrderLinks (esta es la que vamos a utilizar nosotros)
2. Orden en memoria -> InMemoryOrderLinks
3. Orden de inicialización -> InInitializationOrderLinks

Además esta estructura tiene el nombre de la dll y la ruta completa en el campo **FullDllName**, y su dirección de memoria en **DllBase**.

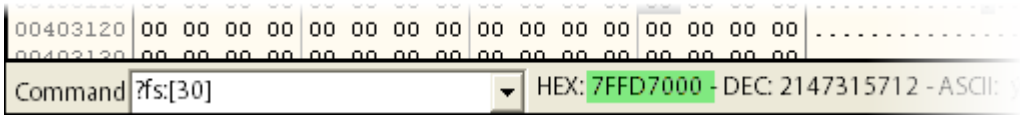
Hay que tener en cuenta que el nombre está almacenado en un string Unicode (ya veremos esto más adelante).

Como es un lío de punteros y estructuras vamos a tratar de aclarar un poco todo esto con un gráfico. Básicamente lo que tendríamos que hacer sería algo así:

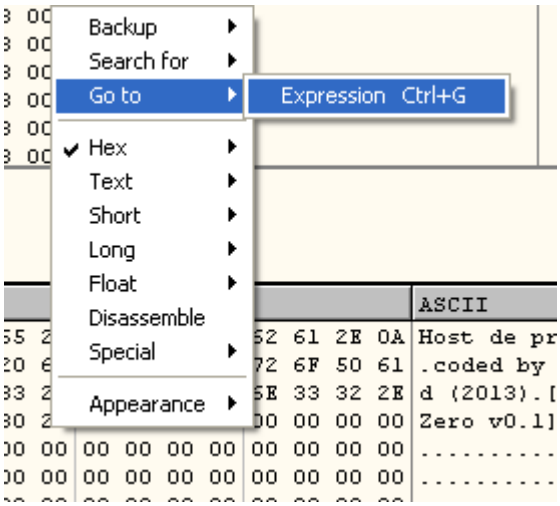


En realidad no es muy complicado, la clave es no perderse entre tantas estructuras y practicar bastante para saber donde estamos en cada momento.

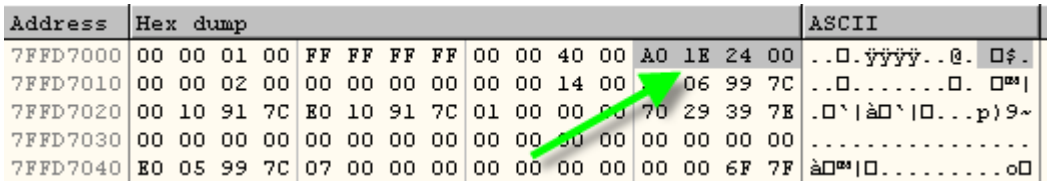
Primero veamos lo que queremos hacer en forma manual y después recién vayamos al código, así se entiende bien. Volvamos a cargar nuestro programa en OllyDBG, y en la barra de comandos escribamos ? fs:[30h]. Vemos que nos devuelve una dirección, en mi caso 7FFD7000, esa justamente es la dirección de inicio del PEB para este proceso:



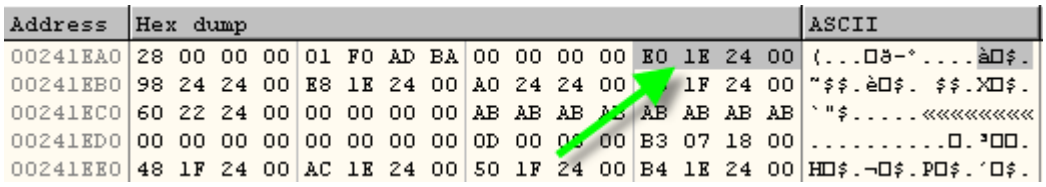
Nos vamos a la sección de dump y presionamos el botón derecho del mouse y seleccionamos 'Go to -> Expression':



Y luego introducimos la dirección que obtuvimos antes. A partir de ahí nos desplazamos **0Ch** lugares para obtener la dirección del **Ldr**, y anotamos esta nueva dirección:



Nos dirigimos a esa nueva dirección de memoria (241EA0h en mi caso) y vamos a quedar parados en el inicio de PEB_LDR_DATA, luego nos desplazamos **0Ch** lugares y tenemos una nueva dirección, que es la de InLoadOrderLinks, ánimo que falta poco:

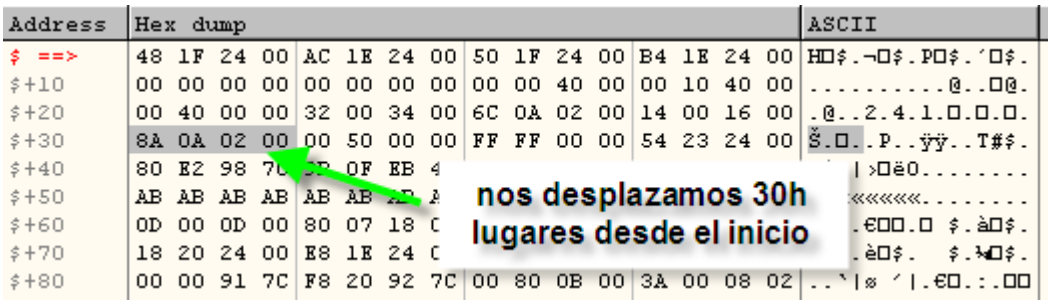


Esa dirección es la de la primer entrada de los módulos cargados por el programa, vamos a esa dirección (Ctrl-G) y quedamos parados en el primer módulo cargado (recordemos que corresponde a una estructura LDR_DATA_TABLE_ENTRY).

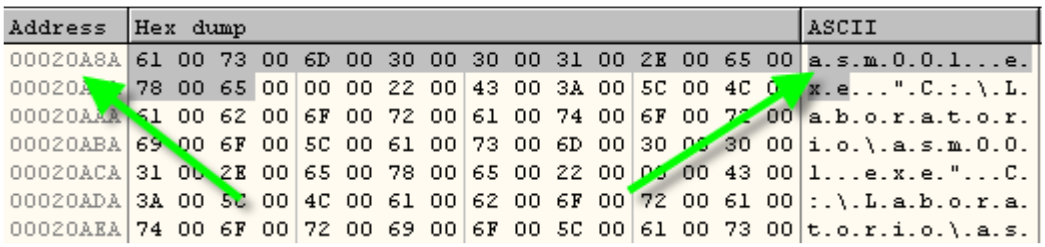
Como vimos anteriormente, si desde el inicio nos desplazamos **02Ch** lugares vamos a tener la dirección de la cadena que contiene el nombre del módulo, pero como es una cadena UNICODE los primeros bytes corresponden a su tamaño, veamos como es esta estructura:

```
typedef struct _UNICODE_STRING
{
    WORD Length;
    WORD MaximumLength;
    WORD * Buffer;
} UNICODE_STRING, *PUNICODE_STRING;
```

En conclusión, para obtener la dirección real debemos desplazarnos 04h lugares más para saltear esos primeros bytes que no nos interesan, o sea que en definitiva nos desplazaremos 30h lugares desde el inicio del PEB_LDR_DATA, veamos:

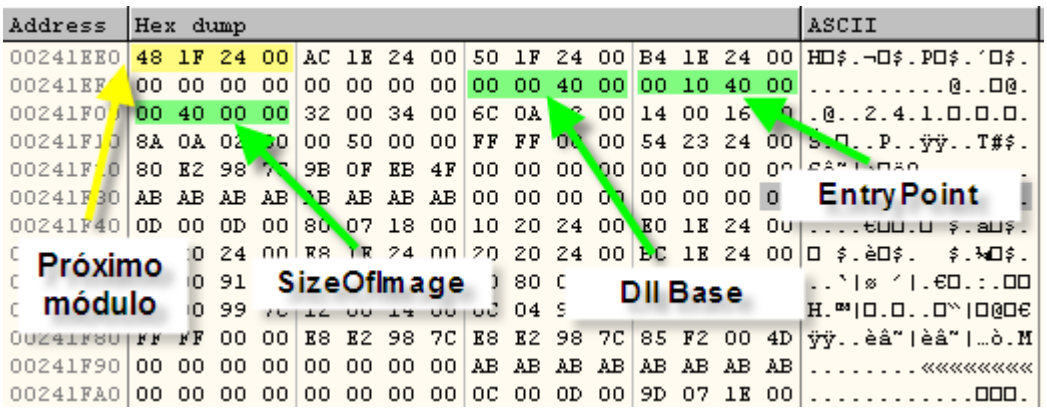


Ok, vayamos a esa dirección (020A8Ah) y veamos:



Bien, lo logramos, ya tenemos el nombre del primer módulo cargado, en este caso es el nombre del propio ejecutable: "asm001.exe" (ojo, se ve así porque está en formato Unicode).

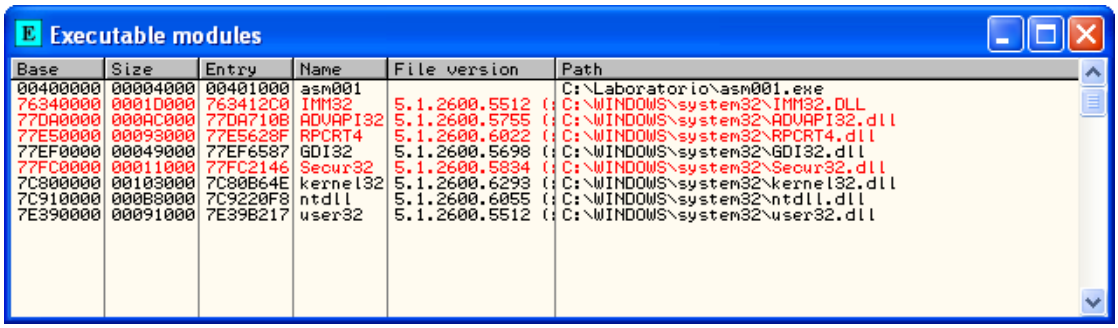
Veamos de paso que otra información podemos sacar de este módulo, volvamos al inicio del LDR_DATA_TABLE_ENTRY (241EE0h):



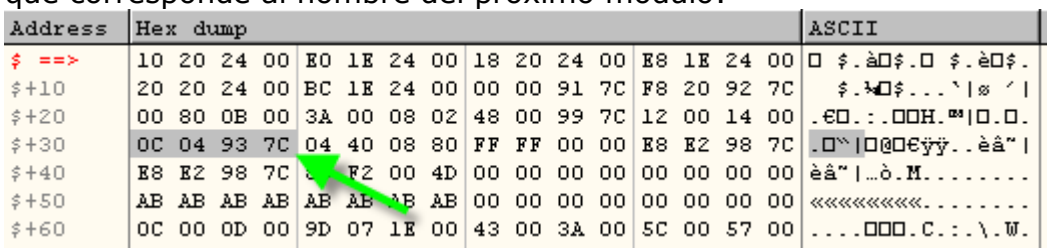
Acá tenemos información muy interesante, pasémosla en limpio:

Dll Base: 400000
EntryPoint: 401000
SizeOfImage: 4000

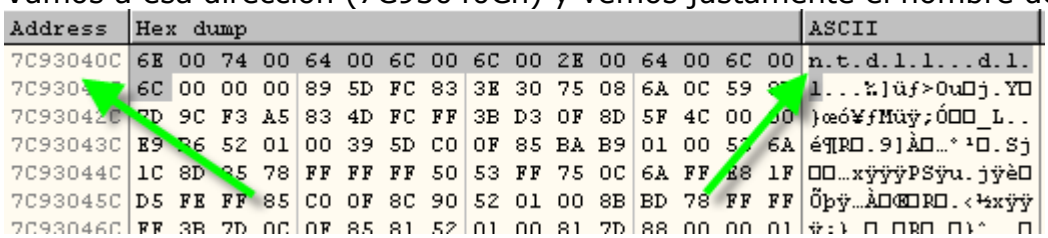
Para verificar esto podemos ir a 'Show modules' (E) y comparar estos valores con las columnas Base, Size y Entry:



Además tenemos la dirección del próximo módulo cargado (en este caso 241F48h), recordemos que es un FLink (Forward link), vamos ahí y veamos el desplazamiento 30h, que corresponde al nombre del próximo módulo:



Vamos a esa dirección (7C93040Ch) y vemos justamente el nombre del módulo:



Que corresponde a: "ntdll.dll".

Y así podríamos seguir hasta encontrar todos los módulos cargados por el programa.
En nuestro caso nos interesa recorrer esta estructura hasta encontrar "kernel32.dll".

Ahora veamos como hacemos esto desde el código, sería algo así:

```
---------------------------------------------------------------------------------------------
   mov     edx, fs:[30h]              ; edx =inicio del PEB
   mov     edx, [edx + 0Ch]           ; edx = Ldr (inicio del PEB_LDR_DATA)
   add     edx, 0Ch                   ; edx = puntero al inicio del primer módulo
   cld                                ; limpia el byte de dirección para la comparación

proximoModulo:
   mov     edx, [edx]                  ; dirección de inicio del módulo
   mov     esi, [edx + 02Ch + 04h]     ; esi = puntero al nombre del módulo
   lea     edi, offset nombreK32       ; edi = puntero a la cadena 'kernel32.dll'
   mov     ecx, 0Ch                    ; largo de la cadena 'kernel32.dll'

   repe    cmpsb                       ; compara [esi] con [edi] byte a byte
   jnz     proximoModulo               ; no es igual, busco el próximo módulo

   mov     eax, [edx + 18h]            ; en eax me queda la dirección de kernel32.dll
---------------------------------------------------------------------------------------------
```

Y en la sección .data defino la cadena a utilizar:

```
---------------------------------------------------------------------------------------------
nombreK32   db 'k',0,'e',0,'r',0,'n',0,'e',0,'l',0,'3',0,'2',0,'.',0,'d',0,'l',0,'l',0
---------------------------------------------------------------------------------------------
```

Recordemos que la definimos así porque no es ASCII, es UNICODE y cada carácter ocupa dos lugares.

Agreguemos este código en un programa y veámoslo en OllyDBG:

```
00401000 r$  64:8B15 3000(MOV EDX,DWORD PTR FS:[30]
00401007 .   8B52 0C      MOV EDX,DWORD PTR DS:[EDX+C]
0040100A .   83C2 0C      ADD EDX,0C
0040100D .   FC           CLD
0040100E >   8B12         MOV EDX,DWORD PTR DS:[EDX]
00401010 .   8B72 30      MOV ESI,DWORD PTR DS:[EDX+30]
00401013 .   8D3D 3A304000 LEA EDI,DWORD PTR DS:[40303A]
00401019 .   B9 0C000000  MOV ECX,0C
0040101E .   F3:A6        REPE CMPS BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
00401020 .^  75 EC        JNZ SHORT pruebal.0040100E
00401022 .   8B42 18      MOV EAX,DWORD PTR DS:[EDX+18]
```

Esta es toda la rutina (por ahora), si la ejecutamos paso a paso con F8 y nos detenemos antes de ejecutar 'repe cmpsb' vemos en los registros las dos cadenas que va a comparar:

```
Registers (FPU)
EAX 00000000
ECX 0000000C
EDX 00241EE0
EBX 7FFDA000
ESP 0012FFC4
EBP 0012FFF0
ESI 00020A8A UNICODE "asm001.exe"
EDI 0040303A UNICODE "kernel32.dll"
```

En este caso no es la que nos interesa, asi que salta al próximo módulo y vemos:

```
ESI 7C93040C UNICODE "ntdll.dll"
EDI 0040303A UNICODE "kernel32.dll"
```

Tampoco es, veamos el siguiente:

```
ESI 00241FD8 UNICODE "kernel32.dll"
EDI 0040303A UNICODE "kernel32.dll"
```

Si, este es, veamos que en este caso no salta y sale de la rutina. Si ahora vamos al desplazamiento 18h desde el inicio del LDR_DATA_TABLE_ENTRY, vamos a tener la dirección de memoria de kernel32.dll, en este caso la guardo en eax:

```
00401013 .   8D3D 3A304000 LEA EDI,DWORD PTR DS:[40303A]
00401019 .   B9 0C000000  MOV ECX,0C
0040101E .   F3:A6        REPE CMPS BYTE PTR ES:[EDI],BYTE PTR
00401020 .^  75 EC        JNZ SHORT asm001.0040100E
00401022 .   8B42 18      MOV EAX,DWORD PTR DS:[EDX+18]          kernel32.7C800000
```

Vemos incluso que el OllyDBG detecta que es la dirección de inicio de kernel32.dll en memoria y lo aclara en la pantalla.

Bien, logramos nuestro objetivo, ahora podemos mejorar un poco esta rutina, para eso, en vez de comparar cadenas UNICODE lo que vamos a hacer es trabajar con hash de esas cadenas. Esto lo hace más seguro ya que no estamos guardando cadenas que pueden resultar sospechosas para los antivirus, además reduce el tamaño del código.

Veamos de que se trata esto. Una función hash es una función cuyo resultado se obtiene a través de un algoritmo que tiene como entrada un conjunto de elementos (que suele ser una cadena), y los convierte en un valor de salida de longitud fija. Normalmente la cadena de entrada tiene una longitud más elevada que la de salida, por ese se las llama también funciones de resumen.

En definitiva, podemos decir que es una función ***f*** que aplicada a una entrada **X** nos da una salida **Y** que siempre es la misma:

f(X) = Y

Ahora, lo que haríamos en nuestro programa sería lo siguiente:

1) calcular el valor **Y** para la cadena **'kernel32.dll'** de antemano

2) guardar ese valor en nuestro programa (en vez de la cadena)

3) ir tomando uno a uno los nombres de los módulos y aplicarles la función ***f***

4) Comparar el resultado de la misma con el valor **Y** guardado, y si coincide es porque hemos encontrado la que nos interesa

Es importante siempre que se trabaja con funciones hash evitar las colisiones, estas se refieren a que si aplicamos la función ***f*** a dos valores distintos obtendríamos el mismo resultado, con el peligro de que nos equivoquemos al creer que hemos encontrado el valor buscado cuando en realidad es otro. Esto se resumiría de la siguiente forma:

f (X) = **Y**

f (Z) = **Y**

En este caso hay colisión porque con dos entradas distintas (X y Z) obtenemos el mismo resultado. Como en nuestro caso los valores de entradas van a ser muy acotados podemos utilizar una función que tiene un bajo nivel de colisiones que es la siguiente:

```
---------------------------------------------------------------------------------------------
ror  edi, 0x0d    ; rotate right 13 (0x0d)
add edi, eax
---------------------------------------------------------------------------------------------
```

Esta es una de las funciones más comunes para generar un hash, que se hizo muy popular debido a que venía incluída en Metasploit (herramienta para desarrollar y ejecutar exploits), pero se pueden crear distintas variaciones para nuestro código.

ROR significa 'ROtate Right' (rotación a la derecha), y trabaja a nivel de bits, por ejemplo si tenemos el número 8 en binario:

00000000000000000000000000**1**000

y hacemos un **ror registro, 13** quedaría:

00000000**1**00000000000000000000000

Vemos que al desbordar por la derecha, los bits vuelven a ingresar por la izquierda.

Esta función de hash que vamos a usar básicamente rota los bits del registro edi, que es donde vamos acumulando el resultado y luego le suma cada caracter de la cadena buscada. Independientemente del largo de la función de entrada, siempre obtendremos como salida un valor de 32 bits.

Veamos ahora como sería la función para obtener el valor hash de kernel32.dll para reemplazar luego el string por este valor en la función que vimos anteriormente.

En la sección de datos definimos la cadena a buscar:

```
---------------------------------------------------------------------------------------------------
buscado     db "kernel32.dll", 0
---------------------------------------------------------------------------------------------------
```

Para llamar a la función empujamos en la pila la dirección de la cadena a buscar:

```
---------------------------------------------------------------------------------------------------
    push    offset buscado
    call    calculaHash
---------------------------------------------------------------------------------------------------
```

Y la función de cálculo del hash quedaría:

```
---------------------------------------------------------------------------------------------------
calculaHash proc
    cld
    xor     eax, eax
    xor     edi, edi
    mov     dword ptr esi, [esp + 04h]

hashing:
    lodsb
    test    al, al
    jz      salir
    ror     edi, 0Dh
    add     edi, eax
    jmp     hashing

salir:
    mov     eax, edi
    ret     4

calculaHash endp
---------------------------------------------------------------------------------------------------
```

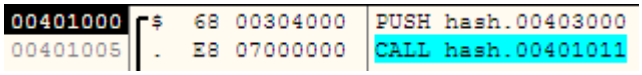
En la primera parte lo que hacemos es determinar la dirección de rotación de la instrucción ror (cld hace que la rotación sea hacia adelante, o sea hacia la derecha), luego limpiamos los registros que vamos a utilizar y recuperamos de la pila la dirección del string buscado.

Luego cargamos el valor de cada carácter en AL, verificamos si es cero (o sea si llegamos al final del string), hacemos la rotación y acumulamos en edi.

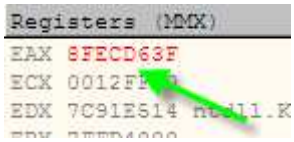
Para terminar movemos el valor a EAX, solo por una cuestión de orden ya que la mayoría de las funciones en assembler devuelven el resultado en ese registro, y volvemos de la función con un ret 4 para balancear la pila.

Este programa no produce ninguna salida, por lo que tendremos que compilarlo y verlo en OllyDBG.

Ejecutemos con F8 las dos primeras líneas del programa:



Y veamos el valor que queda cargado en EAX:



Vemos que EAX tiene el valor 8FECD63F, que es justamente el hash del string 'kernel32.dll'. También podemos ejecutar el código línea a línea con F7 para ir viendo como calcula el valor nuestra función.

En la rutina que vimos al principio hacíamos lo siguiente:

```
---------------------------------------------------------------------------------------------
   lea      edi, offset nombreK32     ; edi = puntero a la cadena 'kernel32.dll'
   mov      ecx, 0Ch                  ; largo de la cadena 'kernel32.dll'

   repe     cmpsb                     ; compara [esi] con [edi] byte a byte
   jnz      proximoModulo             ; no es igual, busco el próximo módulo
---------------------------------------------------------------------------------------------
```

Ahora cambiamos esto por:

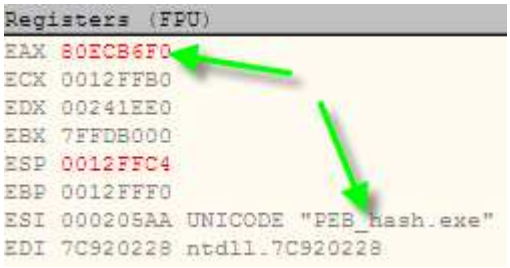
```
---------------------------------------------------------------------------------------------
   push     esi
   call     calculaHash

   cmp      eax, 8FECD63Fh            ; hash de kernel32.dll: 8FECD63Fh
   jnz      proximoModulo             ; no es igual, busco el próximo módulo
---------------------------------------------------------------------------------------------
```

El resto queda igual y agregamos la función de cálculo del hash a nuestro código. Lo único que le vamos a cambiar a nuestra función de hash es que al princpio guardaremos los valores de esi y edi para restaurarlos antes de salir de la misma, lo que hará que cambiemos la línea donde recuperamos la dirección del string de la pila:

```
---------------------------------------------------------------------------------------------
   mov      dword ptr esi, [esp + 0Ch]
---------------------------------------------------------------------------------------------
```

Veamos todo esto en OllyDBG. Cargamos el programa y vamos ejecutando con F8 hasta que calcula el hash del primer módulo de nuestro programa, en este caso el propio programa ejecutable:



O sea que 80ECB6F0 es el valor hash del string 'PEB_hash.exe'. Seguimos ejecutando y viendo como calcula el hash de cada módulo del programa hasta que llega a kernel32.dll:

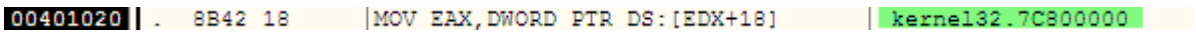
Luego el programa compara este valor con el que habíamos obtenido anteriormente y concluye así que se encontró el módulo buscado:

```
---------------------------------------------------------------------------------------------------
   cmp     eax, 8FECD63Fh
---------------------------------------------------------------------------------------------------
```

Luego solo queda recuperar la dirección de la dll, recordemos que estamos parados sobre una estructura LDR_DATA_TABLE_ENTRY, y que en el deslazamiento 18h encontraremos la dirección de la misma:

```
---------------------------------------------------------------------------------------------------
   mov     eax, [edx + 18h]
---------------------------------------------------------------------------------------------------
```

Listo, para ver que justamente hemos encontrado la dirección de kernel32.dll podemos ver en OllyDBG que este la reconoce y lo aclara en el código:



Esta función de hash no solamente nos servirá para encontrar la dirección de kernel32.dll, sino que podremos utilizarla cada vez que comparemos algún valor para ver si es el buscado y que no quede tan evidente (además de ahorrar unos cuantos bytes).

Bien, pero todavía faltarían un par de cosas para dejarla completamente funcional: una sería controlar la cantidad de iteraciones que realiza en el ciclo de búsqueda de kernel32.dll, sino podría quedar en un ciclo infinito, ya que la única opción de salir del mismo es justamente encontrar la dirección buscada (y por algún motivo esto puede no suceder). Esto lo solucionamos fácilmente creando un contador, por ejemplo con el

registro ECX que lo vamos incrementando en cada loop del ciclo y si llega a un valor determinado cortamos el ciclo y salimos.

El otro inconveniente es que al comparar strings para ver si encontramos 'kernel32.dll', nos podemos encontrar con que se llame 'Kernel32.dll', 'KERNEL32.DLL', etc. Para evitar esto lo que vamos a hacer es pasar los caracteres a mayúsculas antes de aplicar la función hash, asi siempre estaremos buscando el hash de 'KERNEL32.DLL' , cuyo valor es 6E2BCA17h.

Una rutina típica y sencilla para pasar una cadena a mayúsculas es la siguiente:

```
---------------------------------------------------------------------------------------------------
pasa_mayusculas proc
   mov     esi, [esp + 04h]       ; recuperamos el parámetro del stack

inicio:
   cmp     byte ptr [esi], 0h     ; es cero?  entonces terminamos y salimos
   jz      listo

   cmp     byte ptr [esi], 'a'    ; saltar si es menor que 'a'
   jb      proximo
   cmp     byte ptr [esi], 'z'    ; saltar si es mayor que 'z'
   ja      proximo

   sub     byte ptr [esi], 20h    ; convierte en mayuscula

proximo:
   inc     esi                    ; próximo caracter
   jmp     inicio

listo:
   ret     4
---------------------------------------------------------------------------------------------------
```

Y para llamarla haríamos:

```
---------------------------------------------------------------------------------------------------
   push    offset cadena
   call    pasa_mayusculas
---------------------------------------------------------------------------------------------------
```

Para entender bien como trabaja esta rutina veamos la diferencia que hay entre los valores ascii de los caracteres en mayúsculas y minúsculas:

A: 41h
a: 61h

B: 42h
b: 62h

Z: 5ah
z: 7ah

O sea que entre un carácter en minúscula y otro en mayúscula la diferencia es **20h**, entonces es simple, restamos ese valor a cada carácter y vamos a obtener su

equivalente en mayúsculas. El resto de la rutina es simple y está comentado en el código.

Como aclaración, porque se lo pueden encontrar en algún código, en vez de hacer un sub 20h, también pueden encontrar lo siguiente:

```
---------------------------------------------------------------------------------------------
   and     byte ptr [esi], 11011111b
---------------------------------------------------------------------------------------------
```

Que en definitiva es lo mismo si lo pensamos a nivel de bits, veamos:

A (41h): 01**0**00001
a (61h): 01**1**00001

Z (5ah): 01**0**11010
z (7ah): 01**1**11010

Vemos que la diferencia entre el carácter en mayúsculas y minúsculas está en el sexto bit (esto es porque 2^5 = 32d = 20h), o sea que si ponemos en cero ese bit y dejamos el resto igual es lo mismo que restarle 20h al número.

Perdón si explico demasiado cada cosa y se hace un poco pesado, pero la idea es que quede todo bien claro y se entienda lo mejor posible.

Ahora bien, esta rutina la expliqué en forma independiente del código para que se entienda bien lo que hace, pero no podemos pasarle el nombre del módulo a esta rutina para convertirlo a mayúsculas porque nos daría una excepción, ya que no podemos cambiar este valor en LDR_DATA_TABLE_ENTRY directamente. Para solucionar esto, simplemente vamos pasando cada carácter a mayúscula antes de calcular el hash.

Entonces el cálculo del hash quedaría así:

```
---------------------------------------------------------------------------------------------
calculaHash proc
   cld
   xor     eax, eax
   xor     edi, edi

hashing:
   lodsw
   cmp     al, ah
   je      salir

mayuscula:
   cmp     eax, 'a'      ; saltar si es menor que 'a'
   jb      seguir

cmp     eax, 'z'      ; saltar si es mayor que 'z'
   ja      seguir

   sub     eax, 20h      ; convierte en mayúscula
seguir:
   ror     edi, 0Dh
```

```
    add     edi, eax
    jmp     hashing

salir:
    mov     eax, edi
    ret

calculaHash endp
```

Básicamente lo único que cambia es lo que está en azul. Espero que se entienda todo bien, quise explicarlo de esta manera justamente para que quedara más claro.

Bueno, eso esto todo por ahora, espero que les sea útil y nos vemos en la próxima.

:: zeroPad ::
Junio 2014