# [UnDocumented] PEB, accediendo a los módulos

JUN20 2012 ESCRITO POR NOX

# Introducción

El tan usado y toqueteado PEB, cuando me encontraba realizando un proyecto, de pronto necesité usar APIs sin importarlas, es dónde me acuerdo del PEB y de buscar la documentación oficial.

Para los que hacen o analizan malware, o crean exploits seguramente lo han visto/usado, la estructura PEB no tiene una documentación exacta en la MSDN y la poquísima que tiene está capada, no muestra todos los miembros de las estructuras.

Por eso decidí hacer está investigación sobre como acceder a los módulos.

**Un ejemplo de uso:**

Para usar APIs sin importarlas (conocidísimo en malware o exploit) una de las formas es obtener el módulo de kernel32, recorrer la cabecera opcional y obtener la RVA del directorio de exportaciones, esta contiene tres tablas NumberOfFunctions, NumberOfNames y AddressOfFunctions en resumidas cuentas estás tablas se usan para obtener una de las funciones que exporta esta librearía "GetProcAddress" y así obtener las direcciones de diferentes funciones así como de otras librearías.

Para que esto se pueda cumplir necesitamos obtener el **módulo** de Kernel32 (Esta librería exporta la función "GetProcAddress") y si no hay una documentación oficial exacta y lo poco que encuentras está en ingles o si encuentras códigos sueltos pero no sabes porque se codeo así…

Ese es el fin de este escrito, pero el uso del PEB tiene muchas utilidades queda a cada investigar con ya documentación acerca de sus estructuras 😃.

# Al Laburo

El PEB (**Process Environment Block**) se encuentra en la estructura TIB (**Thread Information Block**) también conocido como TEB (**Thread Evnironment Block**), precisamente en el offset +0×30, el segmento FS apunta a la TIB y podemos acceder de la siguiente manera.

```
xor eax, eax
mov eax, dword ptr fs:[eax + 18h]; eax = TIB
```

En mi ansias de poder usar APIs sin importarlas, comencé a buscar info y sobre todo revisar la documentación oficial, mientras hacía esto vi totales diferencias en los codes posteados con lo que decía la MSDN, es dónde comencé a usar OllyDBG y mirar las estructuras volcándolas, habían valores que no mencionaba la MSDN, es ahí dónde comienzo la travesía de intentar, codear y hacer pruebas. Para esto mi amigo el Peligroso Apo (APOKLIPTIKO conocido en CrackSLatinoS) me ayudó en la búsqueda de "documentación" para facilitar la compresión (si desean encontrar algo perdido por la red, Apo es el indicado, salta firewalls, evita baneos de IP, usa proxys para lograr su objetivo, hasta encuentra las páginas prohibidas).

La estructura **_PEB** se encuentra en el desplazamiento +0×030.

```
0:000> dt ntdll!_TEB
   +0x000 NtTib            : _NT_TIB
   +0x01c EnvironmentPointer : Ptr32 Void
   +0x020 ClientId         : _CLIENT_ID
   +0x028 ActiveRpcHandle  : Ptr32 Void
   +0x02c ThreadLocalStoragePointer : Ptr32 Void
   +0x030 ProcessEnvironmentBlock : Ptr32 _PEB
;  [...]
```

La estructura **_PEB** es la siguiente.

```
0:000> dt ntdll!_PEB
   +0x000 InheritedAddressSpace : UChar
   +0x001 ReadImageFileExecOptions : UChar
   +0x002 BeingDebugged    : UChar
   +0x003 BitField         : UChar
   +0x003 ImageUsesLargePages : Pos 0, 1 Bit
   +0x003 IsProtectedProcess : Pos 1, 1 Bit
   +0x003 IsLegacyProcess  : Pos 2, 1 Bit
   +0x003 IsImageDynamicallyRelocated : Pos 3, 1 Bit
   +0x003 SkipPatchingUser32Forwarders : Pos 4, 1 Bit
   +0x003 SpareBits        : Pos 5, 3 Bits
   +0x004 Mutant           : Ptr32 Void
   +0x008 ImageBaseAddress : Ptr32 Void
   +0x00c Ldr              : Ptr32 _PEB_LDR_DATA
;  [...]
```

Se obtiene el puntero de la estructura **_PEB** a continuación.

```
xor eax, eax
mov eax, dword ptr fs:[eax + 30h]; eax = _PEB
```

El miembro **Ldr (LoaderData)** apunta a la estructura **_PEB_LDR_DATA** y en esta podemos encontrar los módulos cargados.

```
0:000> dt ntdll!_PEB_LDR_DATA
   +0x000 Length           : Uint4B
   +0x004 Initialized      : UChar
   +0x008 SsHandle         : Ptr32 Void
   +0x00c InLoadOrderModuleList : _LIST_ENTRY
   +0x014 InMemoryOrderModuleList : _LIST_ENTRY
   +0x01c InInitializationOrderModuleList : _LIST_ENTRY
   +0x024 EntryInProgress  : Ptr32 Void
   +0x028 ShutdownInProgress : UChar
   +0x02c ShutdownThreadId : Ptr32 Void
```

Con el siguiente código podemos obtener la estructura **_PEB_LDR_DATA**.

```
xor eax, eax
mov eax, [fs:eax + 30h]
mov eax, [eax + 0Ch]; eax = _PEB_LDR_DATA
```

Existen 3 listas para poder obtener los módulos, y están en orden que fueron, cargados, en memoria y de inicialización, *InLoadOrderModuleList, InMemoryOrderModuleList* y *InInitializationOrderModuleList* respectivamente.

El siguiente código deja el puntero del miembro **InLoadOrderModuleList** en eax.

```
xor eax, eax
mov eax, [fs:eax + 30h]
mov eax, [eax + 0Ch]
lea eax, [eax + 0Ch]
```

El siguiente código deja el puntero del miembro **InMemoryOrderModuleList** en eax.

```
xor eax, eax
mov eax, [fs:eax + 30h]
mov eax, [eax + 0Ch]
lea eax, [eax + 14h]
```

El siguiente código deja el puntero del miembro **InInitializationOrderModuleList** en eax.

```
xor eax, eax
mov eax, [fs:eax + 30h]
mov eax, [eax + 0Ch]
lea eax, [eax + 1Ch]
```

**LIST_ENTRY** es una estructura doble enlazada, contiene el siguiente (flink) y anterior (blink) módulo .

```
0:000> dt ntdll!_LIST_ENTRY
   +0x000 Flink            : Ptr32 _LIST_ENTRY
   +0x004 Blink            : Ptr32 _LIST_ENTRY
```

*InLoadOrderModuleList, InMemoryOrderModuleList* y *InInitializationOrderModuleList* apuntan a una estructura (especificamente los flinks de ellos) **_LDR_DATA_TABLE_ENTRY** y estas tiene una variación según corresponda al miembro actual usado.

```
0:000> dt nt!_LDR_DATA_TABLE_ENTRY
ntdll!_LDR_DATA_TABLE_ENTRY
   +0x000 InLoadOrderLinks : _LIST_ENTRY
   +0x008 InMemoryOrderLinks : _LIST_ENTRY
   +0x010 InInitializationOrderLinks : _LIST_ENTRY
   +0x018 DllBase          : Ptr32 Void
   +0x01c EntryPoint       : Ptr32 Void
   +0x020 SizeOfImage      : Uint4B
   +0x024 FullDllName      : _UNICODE_STRING
   +0x02c BaseDllName      : _UNICODE_STRING
   +0x034 Flags            : Uint4B
   +0x038 LoadCount        : Uint2B
   +0x03a TlsIndex         : Uint2B
   +0x03c HashLinks        : _LIST_ENTRY
   +0x03c SectionPointer   : Ptr32 Void
   +0x040 CheckSum         : Uint4B
   +0x044 TimeDateStamp    : Uint4B
   +0x044 LoadedImports    : Ptr32 Void
   +0x048 EntryPointActivationContext : Ptr32 _ACTIVATION_CONTEXT
   +0x04c PatchInformation : Ptr32 Void
   +0x050 ForwarderLinks   : _LIST_ENTRY
   +0x058 ServiceTagLinks  : _LIST_ENTRY
   +0x060 StaticLinks      : _LIST_ENTRY
   +0x068 ContextInformation : Ptr32 Void
   +0x06c OriginalBase     : Uint4B
   +0x070 LoadTime         : _LARGE_INTEGER
```

Miembros muy interesantes se pueden observar en esta estructura, *BaseAddress*, contiene la dirección del módulo cargado, *FullDllName*, muestra que uno de sus miembros (*Buffer*) contiene el PATH completo del módulo que ha sido cargado (ejem. 'C:\Windows\SYSTEM32 \ntdll.exe'). Tampoco podemos evitar darnos cuenta del tipo de dato **_UNICODE_STRING**, a continuación se muestra su estructura.

```
0:000> dt nt!_UNICODE_STRING
ntdll!_UNICODE_STRING
   +0x000 Length           : Uint2B
   +0x002 MaximumLength    : Uint2B
   +0x004 Buffer           : Ptr32 Uint2B
```

La estructura **_UNICODE_STRING** tiene como miembro más resaltante a *Buffer* que es UNICODE y contiene el *BaseName* del miembro *BaseDllName* (ejem. 'ntdll.exe') del módulo.

# APIs sin importarlas

Para conseguir ese objetivo, es necesario obtener el Módulo de Kernel32.dll, desplazarnos hasta el directorio de exportaciones, este directorio contiene a tres tablas, *NumberOfFunctions*, *NumberOfNames* y *AddressOfFunctions*, usando estas tablas en resumidas cuentas se encontrará la dirección de GetProcAddress y así obtener las direcciones de las APIs que se necesite. El fin de este escrito es mostrar la manera de acceder a los módulos de las tres formas (de inicialización, carga y memoria), pero no viene mal una pequeña referencia en dónde se puede aplicar lo aprendido en esta entrada.

# En orden de Carga

La estructura **_LDR_DATA_TABLE_ENTRY** usada para este orden sería el siguiente:

```
0:000> dt nt!_LDR_DATA_TABLE_ENTRY
ntdll!_LDR_DATA_TABLE_ENTRY
   +0x000 InLoadOrderLinks : _LIST_ENTRY
   +0x008 InMemoryOrderLinks : _LIST_ENTRY
   +0x010 InInitializationOrderLinks : _LIST_ENTRY
   +0x018 DllBase          : Ptr32 Void ; < ----------
   +0x01c EntryPoint       : Ptr32 Void
   +0x020 SizeOfImage      : Uint4B
   +0x024 FullDllName      : _UNICODE_STRING
   +0x02c BaseDllName      : _UNICODE_STRING
          ntdll!_UNICODE_STRING
             +0x02c Length           : Uint2B
             +0x02e MaximumLength    : Uint2B
             +0x030 Buffer           : Ptr32 Uint2B ; <---------
   +0x034 Flags            : Uint4B
   +0x038 LoadCount        : Uint2B
   +0x03a TlsIndex         : Uint2B
   +0x03c HashLinks        : _LIST_ENTRY
   +0x03c SectionPointer   : Ptr32 Void
   +0x040 CheckSum         : Uint4B
   +0x044 TimeDateStamp    : Uint4B
   +0x044 LoadedImports    : Ptr32 Void
   +0x048 EntryPointActivationContext : Ptr32 _ACTIVATION_CONTEXT
   +0x04c PatchInformation : Ptr32 Void
   +0x050 ForwarderLinks   : _LIST_ENTRY
   +0x058 ServiceTagLinks  : _LIST_ENTRY
   +0x060 StaticLinks      : _LIST_ENTRY
   +0x068 ContextInformation : Ptr32 Void
   +0x06c OriginalBase     : Uint4B
   +0x070 LoadTime         : _LARGE_INTEGER
```

Código para acceder a los módulos.

```
        push 30h
        pop esi
        lodsd fs:[esi]; eax = PEB
        ;Créditos a Karcrack por las 3 primeras lineas.

        mov eax, [eax + 0Ch]; eax = PEB.Ldr (_PPEB_LDR_DATA)
        lea eax, [eax + 0Ch]; eax = PEB->_PPEB_LDR_DATA->InLoadOrderModuleList

NextModule:
        assume eax: ptr LIST_ENTRY
        mov eax, [eax].Flink; PEB->_PPEB_LDR_DATA->InLoadOrderModuleList.Flink
        assume eax: nothing
        ; eax = accede al siguiente módulo.
        ; eax = _LDR_DATA_TABLE_ENTRY.InLoadOrderLinks[i]


        mov ebx, [eax + 30h]; ebx = _LDR_DATA_TABLE_ENTRY.BaseDllName[i].Buffer
        ;ebx es puntero a BaseDllName de tipo UNICODE

        cmp byte ptr[ebx+6*2], '3'  ;// BaseDllName[6] == '3' ¿?
```

```
        jne NextModule

        mov ebx, [eax + 18h]; _LDR_DATA_TABLE_ENTRY.DllBase[i] = Module.Kernel32
```

Al ingresar al label **NextModule** accede a la siguiente lista (**LIST_ENTRY**) y por ende a la estructura **_LDR_DATA_TABLE_ENTRY** correspondiente.

Finalmente ebx es puntero del *BaseDllName[i].Buffer* del tipo UNICODE y se compara la séptima posición de kernel**'3'**2.dll, si se cumple, el módulo de kernel32.dll es guardado en ebx si no, sigue al siguiente módulo para realizar las mismas operaciones.

**Nota:**

Si volcamos la estructura **PEB->_PPEB_LDR_DATA->InLoadOrderModuleList**, apuntando a la estructura **_LDR_DATA_TABLE_ENTRY**.

Parados en el siguiente punto del código.

```
        push 30h
        pop esi
        lodsd fs:[esi]; eax = PEB
        ;Créditos a Karcrack por las 3 primeras lineas.

        mov eax, [eax + 0Ch]; eax = PEB.Ldr (_PPEB_LDR_DATA)
        lea eax, [eax + 0Ch]; eax = PEB->_PPEB_LDR_DATA->InLoadOrderModuleList

NextModule:
        assume eax: ptr LIST_ENTRY
        mov eax, [eax].Flink
```

Volcamos la dirección de eax.

```
CPU Dump
Address   Hex dump                                          ASCII
005D1AE8  78 1B 5D 00|8C 78 D4 77|80 1B 5D 00|94 78 D4 77| x] Œx Ô w€] ”x Ô w
005D1AF8  00 00 00 00|00 00 00 00|00 00 40 00|1D 10 40 00|           @ @
```

005D1AE8 está apuntando a **LDR_DATA_TABLE_ENTRY.InLoadOrderLinks[i]** y 005D1AF0 **a LDR_DATA_TABLE_ENTRY.InMemoryOrderLinks[i]**, sabiendo esto, pasamos a la siguiente orden.

# En orden de Memoria

La estructura **_LDR_DATA_TABLE_ENTRY** usada para este orden sería el siguiente:

```
0:000> dt nt!_LDR_DATA_TABLE_ENTRY
ntdll!_LDR_DATA_TABLE_ENTRY
   +0x000 InMemoryOrderLinks : _LIST_ENTRY
   +0x008 InInitializationOrderLinks : _LIST_ENTRY
   +0x010 DllBase          : Ptr32 Void ; < ----------
   +0x014 EntryPoint       : Ptr32 Void
   +0x018 SizeOfImage      : Uint4B
   +0x01C FullDllName      : _UNICODE_STRING
   +0x024 BaseDllName      : _UNICODE_STRING
          ntdll!_UNICODE_STRING
             +0x024 Length           : Uint2B
             +0x026 MaximumLength    : Uint2B
             +0x028 Buffer           : Ptr32 Uint2B ; <---------
   +0x02C Flags            : Uint4B
;  [...]
```

Código para acceder a los módulos.

```
        push 30h
```

```
        pop esi
        lodsd fs:[esi]; eax = PEB

        mov eax, [eax + 0Ch]; eax = PEB.Ldr (_PPEB_LDR_DATA)
        lea eax, [eax + 14h]; eax = PEB->_PPEB_LDR_DATA>InMemoryOrderModuleList

NextModule:
        assume eax: ptr LIST_ENTRY
        mov eax, [eax].Flink; PEB->_PPEB_LDR_DATA->InMemoryOrderModuleList.Flink
        assume eax: nothing
        ;  eax = _LDR_DATA_TABLE_ENTRY.InMemoryOrderLinks[i]

        mov ebx, [eax + 28h]; ebx = _LDR_DATA_TABLE_ENTRY.BaseDllName[i].Buffer
        cmp byte ptr[ebx+6*2], '3'  ;// BaseDllName[6] == '3' ¿?
        jne NextModule

        mov ebx, [eax + 10h]; _LDR_DATA_TABLE_ENTRY.DllBase[i] = Module.Kernel32
```

La diferencia aquí es que la estructura **_LDR_DATA_TABLE_ENTRY** comienza desde el miembro **InMemoryOrdenLinks** esta es la diferencia para cada una de los ordenes de la lista de módulos.

Si volcamos la data estructura **_LDR_DATA_TABLE_ENTRY.InMemoryOrdenLinks[i]** de la misma forma pasada.

```
CPU Dump
Address   Hex dump                                          ASCII
005D1AF0  80 1B 5D 00|94 78 D4 77|00 00 00 00|00 00 00 00| €] ”xÔw
005D1B00  00 00 40 00|1D 10 40 00|00 30 00 00|46 00 48 00|   @ @  0  F H
005D1B10  30 19 5D 00|0E 00 10 00|68 19 5D 00|00 40 00 00| 0]   h]  @
```

005D1AF0 como mencioné antes apunta a **LDR_DATA_TABLE_ENTRY.InMemoryOrderLinks[i]**
y la dirección 005D1B18 apunta al miembro **_LDR_DATA_TABLE_ENTRY.BaseDllName[i].Buffer**, que en este caso y en el anterior empiezan con el módulo de la aplicación que usé como target.

# En orden de Inicialización

La estructura **_LDR_DATA_TABLE_ENTRY** usada para este orden sería el siguiente:

```
0:000> dt nt!_LDR_DATA_TABLE_ENTRY
ntdll!_LDR_DATA_TABLE_ENTRY
   +0x000 InInitializationOrderLinks : _LIST_ENTRY
   +0x008 DllBase          : Ptr32 Void ; < ----------
   +0x00C EntryPoint       : Ptr32 Void
   +0x010 SizeOfImage      : Uint4B
   +0x014 FullDllName      : _UNICODE_STRING
   +0x01C BaseDllName      : _UNICODE_STRING
          ntdll!_UNICODE_STRING
             +0x01C Length           : Uint2B
             +0x01E MaximumLength    : Uint2B
             +0x020 Buffer           : Ptr32 Uint2B ; <----------
   +0x024 Flags            : Uint4B
;  [...]
```

Código para acceder a los módulos.

```
        push 30h
        pop esi
        lodsd fs:[esi]; eax = PEB

        mov eax, [eax + 0Ch]; eax = PEB.Ldr (_PPEB_LDR_DATA)
        lea eax, [eax + 1Ch]; eax = PEB._PPEB_LDR_DATA.InInitializationOrderModuleList

NextModule:
        assume eax: ptr LIST_ENTRY
        mov eax, [eax].Flink; PEB._PPEB_LDR_DATA.InInitializationOrderModuleList.Flink
        assume eax: nothing
```

```
; eax = _LDR_DATA_TABLE_ENTRY.InInitializationOrderModuleLinks[i]

mov ebx, [eax+20h]; _LDR_DATA_TABLE_ENTRY.BaseDllName[i].Buffer
cmp byte ptr[ebx+6*2], '3'  ;// BaseDllName[6] == '3' ¿?
jnz NextModule

mov ebx, [eax+08h]; ebx = _LDR_DATA_TABLE_ENTRY.DllBase[i] = Module.Kernel32
```

Una ventaja de usar este último orden es que el primero módulo es ntdll.dll y no como las anteriores, por ende se demoraría menos en encontrar al módulo de kernel32.dll y es el que usaría yo.

Y creo que eso sería todo, si hay algún error espero sus comentarios, no duden en notificarlo mediante sus comentarios.
Me pareció solo comentar como acceder a los módulos y indicar las diferencias en las ordenes, ya que son las que no se encuentra una documentación decente en la MSDN y es porque no HAY, y la única que encuentras es del orden en memoria, pero capada, sin todos los miembros reales de la estructura.

Un agradecimiento a mis grandes amigos de la lista de CrackSLatinoS, a el Peligroso Apo, InDuLgEo, Fly, The Swash, Ricnar, Guan estás últimas personas resaltan, porque se han involucrado en mi avance de este arte Gracias a ellos y a ti por leer este escrito  .

**E.O.F**