Check “Heap Flags” through RtlCreateHeap() Lockbit 3.0

La versión 3.0 de Lockbit trae diferentes estrategias para hacer más complicado su análisis y estas estrategias las va “enriqueciendo” a medida que van avanzando las versiones. Vamos a ver la primera que nos encontraremos durante el análisis, que es cuando los creadores de Lockbit usan RtlCreateHeap() para saber si están siendo depurados o no. Puedes encontrar una investigación completa de esta técnica en el blog de un analista Japonés [1]. Te recomiendo la lectura de cómo ha llevado a cabo el proceso de análisis ya que aquí solo encontrarás un resumen de lo explicado por él.

Para nuestras pruebas utilizamos el hash: A7782D8D55AE5FBFABBAAAEC367BE5BE
Esta muestra después de su ejecución despliega en memoria lo que es **Lockbit 3.0** sin token de acceso y al volcarla a disco el PE el hash es: E5A0136AC4FE028FEA827458B1A70124.

Una vez la cargamos en IDA Pro vemos:

```
.itext:0090946F ; ---------------------------------------------------------------
.itext:0090946F
.itext:0090946F                 public start
.itext:0090946F start:
.itext:0090946F                 nop
.itext:00909470                 nop     dword ptr [eax+eax+00h]
.itext:00909475                 call    nullsub_1
.itext:0090947A                 xchg    ax, ax
.itext:0090947C                 call    sub_8F6390
.itext:00909481                 nop     dword ptr [eax+eax+00000000h]
.itext:00909489                 call    sub_8F9980
.itext:0090948E                 nop
.itext:0090948F                 call    sub_907458
.itext:00909494                 nop     word ptr [eax+eax+00h]
.itext:0090949A                 push    0
.itext:0090949C                 call    dword_9155C0
.itext:009094A2                 nop     dword ptr [eax]
.itext:009094A5                 call    SetLastError
.itext:009094AA                 call    LoadLibraryExA
.itext:009094AF                 call    GetAtomNameW
.itext:009094B4                 call    LoadLibraryW
.itext:009094B9                 call    GetDateFormatW
.itext:009094BE                 call    GetModuleHandleW
.itext:009094C3                 call    LoadLibraryExA
.itext:009094C8                 call    GetDateFormatW
.itext:009094CD                 call    FormatMessageW
.itext:009094D2                 call    GetLastError
```

En la función sub_8F6390, llama a la función RtlCreateHeap() con los parámetros siguientes (es en este punto donde nos vamos a centrar):

```
.text:012763AA push    0
.text:012763AC push    0
.text:012763AE push    0
.text:012763B0 push    0
.text:012763B2 push    0
.text:012763B4 push    41002h
.text:012763B9 call    eax                             ; RtlCreateHeap()
```

- RtlCreateHeap(0,0,0,0,0,41002h);

```
NTSYSAPI PVOID RtlCreateHeap(
  [in]          ULONG                 Flags,
  [in, optional] PVOID                HeapBase,
  [in, optional] SIZE_T               ReserveSize,
  [in, optional] SIZE_T               CommitSize,
  [in, optional] PVOID                Lock,
  [in, optional] PRTL_HEAP_PARAMETERS Parameters
);
```

La función RtlCreateHeap devuelve un PVOID. Vamos a ver que hay en la zona de memoria que nos devuelve la función RtlCreateHeap(0x21e0000 en este caso):

0x21e0000	Private	256 kB	RWX	Heap 32-bit (ID 3)	4 kB	4 kB
0x21e0000	Private: Commit	4 kB	RWX	Heap 32-bit (ID 3)	4 kB	4 kB
0x21e1000	Private: Rese...	252 kB		Heap 32-bit (ID 3)		

```
00000000 20 d4 3c f7 61 1c 00 01 ee ff ee ff 00 00 00 00  .<.a...........
00000010 a8 00 1e 02 a8 00 1e 02 00 00 1e 02 00 00 1e 02 ................
00000020 40 00 00 00 88 05 1e 02 00 00 22 02 3f 00 00 00 @.........".?...
00000030 01 00 00 00 00 00 00 00 f0 0f 1e 02 f0 0f 1e 02 ................
00000040 62 10 04 40 60 00 00 40 00 00 00 00 00 00 10 00 b..@`..@........
00000050 91 d4 3d 47 61 1c 00 00 66 70 d4 2c 00 00 00 00 ..=Ga...fp.,....
00000060 00 fe 00 00 ff ee ff ee 00 00 10 00 00 20 00 00 ............. ..
00000070 00 02 00 00 00 20 00 00 01 01 00 00 ff ef fd 7f ..... ..........
00000080 03 00 38 01 00 00 00 00 00 00 00 00 00 00 00 00 ..8.............
00000090 e8 0f 1e 02 e8 0f 1e 02 17 00 00 00 f8 ff ff ff ................
000000a0 a0 00 1e 02 a0 00 1e 02 10 00 1e 02 10 00 1e 02 ................
000000b0 00 00 00 00 00 00 00 00 50 01 1e 02 90 05 1e 02 ........P.......
000000c0 00 00 00 00 e0 07 1e 02 e0 07 1e 02 38 01 1e 02 ............8...
000000d0 66 70 d4 2c 00 00 00 00 00 00 00 00 00 00 04 00 fp.,............
000000e0 00 10 00 00 00 00 00 00 00 00 00 00 01 00 00 00 ................
000000f0 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000100 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000120 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000130 04 00 00 00 00 00 e0 0f 00 38 6e 3d 00 ff ff ff ff ........8n=.....
00000140 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000150 00 00 00 00 80 00 00 00 01 00 00 00 01 00 00 00 ................
00000160 01 00 00 00 00 00 00 00 c4 00 1e 02 74 01 1e 02 ............t...
00000170 84 01 1e 02 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

Después de crear la zona de memoria en el heap viene lo que sería la técnica antidebug:

```
.text:003463B9 call    eax                             ; RtlCreateHeap()
.text:003463BB mov     esi, eax
.text:003463BD test    esi, esi
.text:003463BF jz      loc_346541
.text:003463C5 mov     eax, [eax+40h]                  ; Antidebug 1
.text:003463C8 shr     eax, 1Ch
.text:003463CB test    al, 4
.text:003463CD jz      short loc_3463D1                ; RtlAllocateHeap()
.text:003463CF rol     esi, 1
```

Como se ve en la imagen empieza recuperando 4 bytes de la posición inicial devuelta por RtlCreateHeap más un offset de 0x40.

Si miramos en la imagen que refleja el estado del heap en memoria donde un debugger está depurando Lockbit (visualización del processhacker), una vez se ha creado, vemos en la posición 0x40 el valor : **40041062 (LE)**.

El puntero devuelto es el puntero a la estructura **_HEAP** (esto es quizá lo más costoso de averiguar, ya que es una estructura interna y hay poca documentación o ninguna) que podemos visualizar en la referencia [2] (increible referencia sobre estructuras) y donde veremos que la posición **0x40** es un unsigned long con nombre **Flags** y la posición **0x44** es otro valor unsigned long con nombre **ForceFlags.**

Sabiendo que son los Flags vemos que está comprobando los 4 bytes de la posición 0x40 (**40041062**) para ver si el valor encaja con alguno de los siguientes:

- HEAP_TAIL_CHECKING_ENABLED (0x20)
- HEAP_FREE_CHECKING_ENABLED (0x40)
- HEAP_VALIDATE_PARAMETERS_ENABLED (0x40000000)

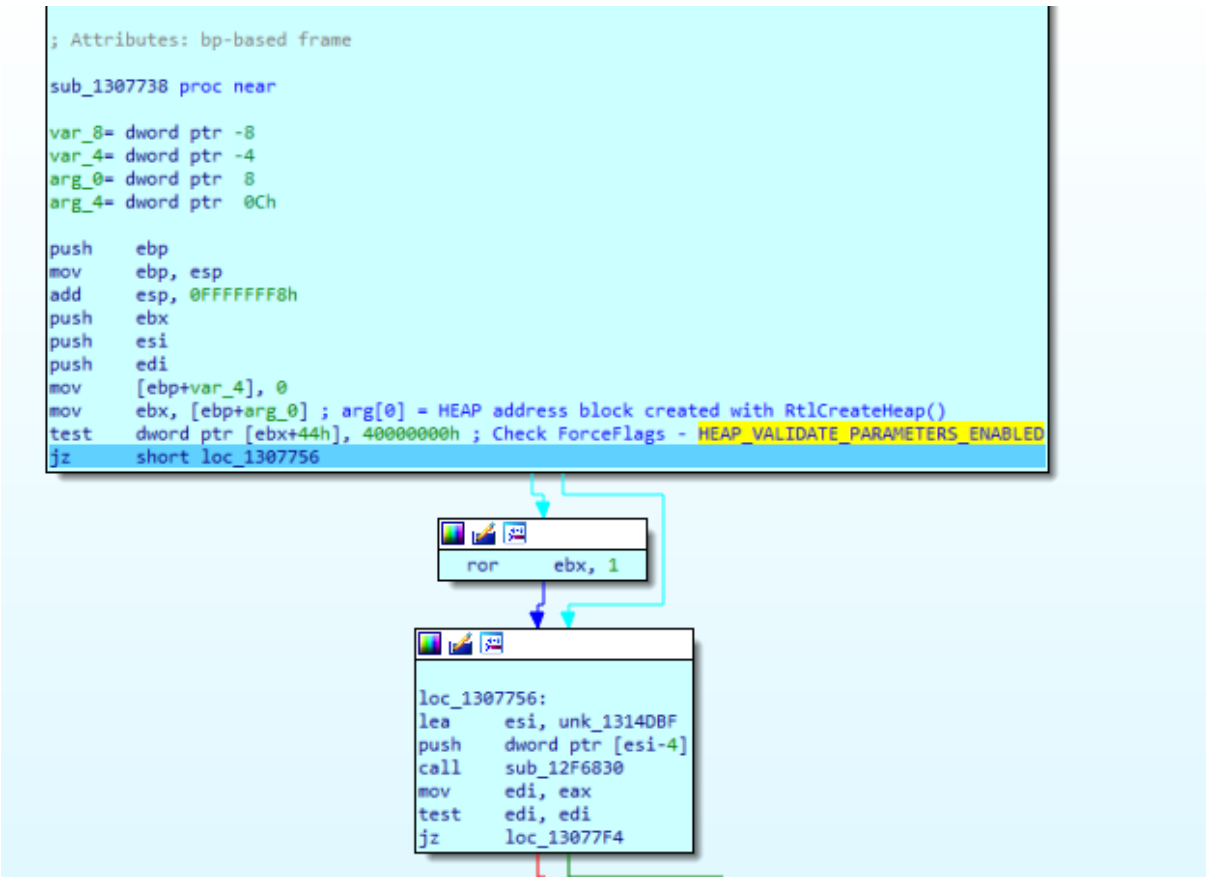
En la web de técnicas antidebug de checkpoint[3] se puede ver descrito como hace el check:

> On 64-bit Windows XP, and Windows Vista and higher, if a debugger is present, the `Flags` field is set to a combination of these flags:
>
> - `HEAP_GROWABLE (2)`
> - `HEAP_TAIL_CHECKING_ENABLED (0x20)`
> - `HEAP_FREE_CHECKING_ENABLED (0x40)`
> - `HEAP_VALIDATE_PARAMETERS_ENABLED (0x40000000)`

Queda claro que está comprobando las flags del heap y lo curioso es cómo está accediendo a la flag a partir de la estructura devuelta por RtlHeapCreate().

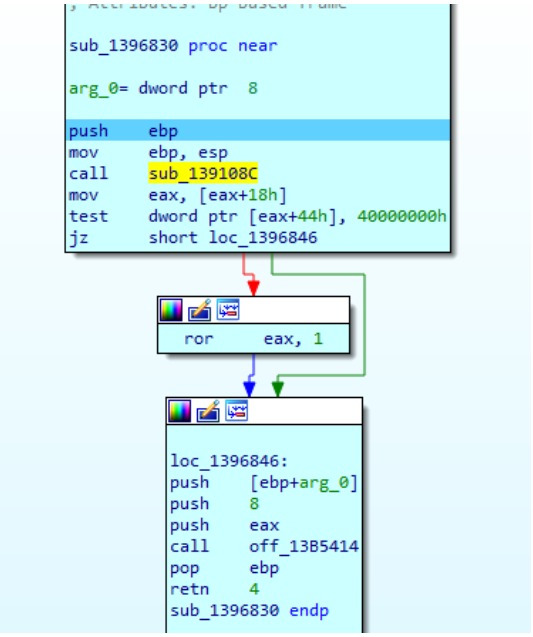
Después en el caso de que detecte que está en un debugger hace un “rol esi, 1” haciendo que cuando se utilice la dirección de un error de acceso a la dirección. Para el bypass de esta técnica antidebug parcheamos el rol esi,1 con un nop.

En otro punto del malware, pasadas dos técnicas antidebug más, nos encontramos que el malware realiza un check de las heap flags pero de un modo un poco diferente:



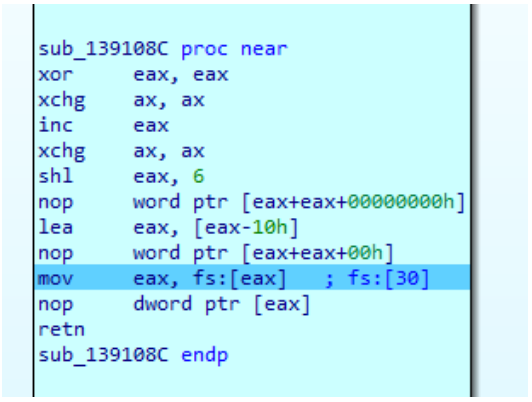
En esta función hace un check de las ForceFlags concretamente, HEAP_VALIDATE_PARAMETERS_ENABLED (0x40000000), para ver si está en un debugger y si está en un debugger manipula la dirección almacenada en ebx para que después al ser utilizada se produzca una excepción.

Si pasamos este check volvemos a encontrarnos con un nuevo check de alguna de las heap flags con una estrategia diferente para acceder a la flag:



Ahora se ve como llama a una función donde el resultado que devuelve le suma 0x18 y después accede a las Force Flags a través del offset 0x44. Lo que devuelve la dirección sub_139108C es la dirección del PEB. Mediante el offset 0x18 obtenemos la dirección base del Heap y de nuevo con esta dirección base se comprueba la flag HEAP_VALIDATE_PARAMETERS_ENABLED.

Para obtener la base del PEB usan esta función que lee fs:[30h]. No ponen el valor 30h directamente sino que lo calculan y después leen el valor:



Referencias:

[1] https://ameblo.jp/reverse-eg-mal-memo/entry-12773724929.html
[2] http://terminus.rewolf.pl/terminus/structures/ntdll/_HEAP_combined.html
[3] https://anti-debug.checkpoint.com/techniques/debug-flags.html#manual-checks-heap-flags