# Check “Heap Flags” through RtlCreateHeap() Lockbit 3.0

Lockbit 3.0 brings different strategies to make its analysis more complicated and these strategies are "enriched" as the versions advance. Let's take a look at the first one we will encounter during analysis, which is when Lockbit's creators use RtlCreateHeap() to find out if they are being debugged or not. You can find a complete investigation of this technique in the blog of a Japanese analyst [1]. I recommend you to read how he carried out the analysis process since here you will only find a summary of what he explained.

For our tests we use the hash: A7782D8D55AE5FBFABBAAAEC367BE5BE
This sample after its execution displays in memory what is Lockbit 3.0 without access token and when dumped to disk the PE hash is: E5A0136AC4FE028FEA827458B1A70124.

Once we load it in IDA Pro we see:

```
.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
```

In function sub_8F6390, it calls the RtlCreateHeap() function with the following parameters (this is where we are going to focus):

```
.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
);
```

The RtlCreateHeap function returns a PVOID. Let's see what is in the memory area returned by the RtlCreateHeap function (0x21e0000 in this case):

| | | | | | | |
|---|---|---|---|---|---|---|
| ◢ 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 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 ................
```

After creating the memory area in the heap comes the antidebug technique:

```
.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
```

As shown in the image it starts by retrieving 4 bytes from the initial position returned by RtlCreateHeap plus an offset of 0x40.

If we look at the image that reflects the state of the heap in memory where a debugger is debugging Lockbit (visualization of the processhacker), once it has been created, we see at position 0x40 the value : **40041062 (LE).**

The returned pointer is the pointer to the **_HEAP** structure (this is perhaps the most expensive to find out, since it is an internal structure and there is little or no documentation) that we can visualize in reference [2] (incredible reference about structures) and where we will see that position **0x40** is an unsigned long with name **Flags** and position **0x44** is another unsigned long value with name **ForceFlags**.

Knowing what the Flags are we see that it is checking the 4 bytes at position 0x40 (40041062) to see if the value matches any of the following:

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

The checkpoint antidebug techniques website[3] describes how the check is done:

> 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)

It is clear that it is checking the flags of the heap and the curious thing is how it is accessing the flag from the structure returned by RtlHeapCreate().

References:


[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