TLS Internals

This will be an analysis of the Thread Local Storage of Windows (TLS), explaining how it works and the inside of the functions.
The analysis will be centered in the APIs TlsSetValue/TlsGetValue.

First of all, any number you see in this text will be in hex, in any other case I will specify the base in parentheses, like 85(dec).

TLS as the name says is a place where we can save information related to the thread.

Related to the thread I mean that this information is going to be part of the thread, if the thread is terminated this information is lost.

This clearly gives us an indication of the second part, being related to the thread, each thread will have its own TLS.

This means that TlsSetValue / TlsGetValue and all other TLS APIs are dependent on the thread from which they are called.

Now that we understand the basics, let's understand where this information is stored. The secret is in the TEB / TIB, each thread that is created in a process has its own TEB, the TEB is nothing more than a structure allocated in memory, the TEB keeps information on many things, GDI, TLS, Stack limits, SEH, everything that is related to the thread and not the process will be stored in this place.

A peculiarity of the TEB is that it can be accessed from the FS segment, which gives us many advantages if we are working in ASM.

From now on, I will access the TEB through the FS segment, which means that if I talk about FS: [0], I mean the first byte of the stuct, FS: [1] the second, etc. Got it ?

In the TEB there are two spaces destined for the TLS, one in FS: [e10] with a length of 100 bytes (40 DWORDs), and another in FS: [f94] with a length of 4 bytes (1 DWORD)

We will call the first space TlsSlots, and the second TlsExpansionSlots

TlsSlots will be a list of DWORDs and TlsExpansionSlots will be a pointer to a heap location.

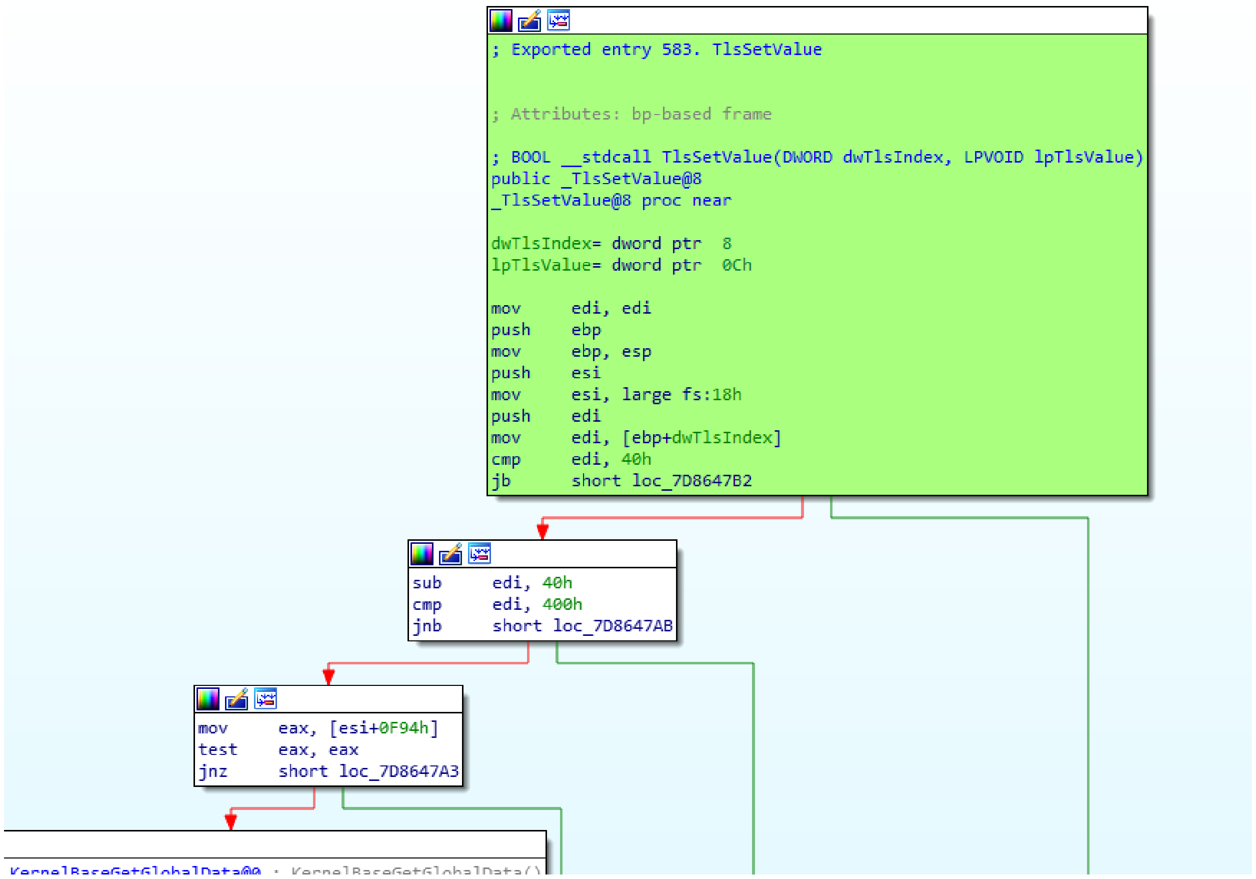
Now that we know where everything related to TLS is stored, let's see how APIs work

TlsSetValue.

This API has 2 parameters, one is the ID and the other the value

What id? Well that id is nothing more than an identifier to know where we keep our value, what value? the value can be any DWORD.

Let's start to reverse



```
; Exported entry 583. TlsSetValue


; Attributes: bp-based frame

; BOOL __stdcall TlsSetValue(DWORD dwTlsIndex, LPVOID lpTlsValue)
public _TlsSetValue@8
_TlsSetValue@8 proc near

dwTlsIndex= dword ptr  8
lpTlsValue= dword ptr  0Ch

mov     edi, edi
push    ebp
mov     ebp, esp
push    esi
mov     esi, large fs:18h
push    edi
mov     edi, [ebp+dwTlsIndex]
cmp     edi, 40h
jb      short loc_7D8647B2
```

```
sub     edi, 40h
cmp     edi, 400h
jnb     short loc_7D8647AB
```

```
mov     eax, [esi+0F94h]
test    eax, eax
jnz     short loc_7D8647A3
```

```
KernelBaseGetGlobalData@0 : KernelBaseGetGlobalData()
```

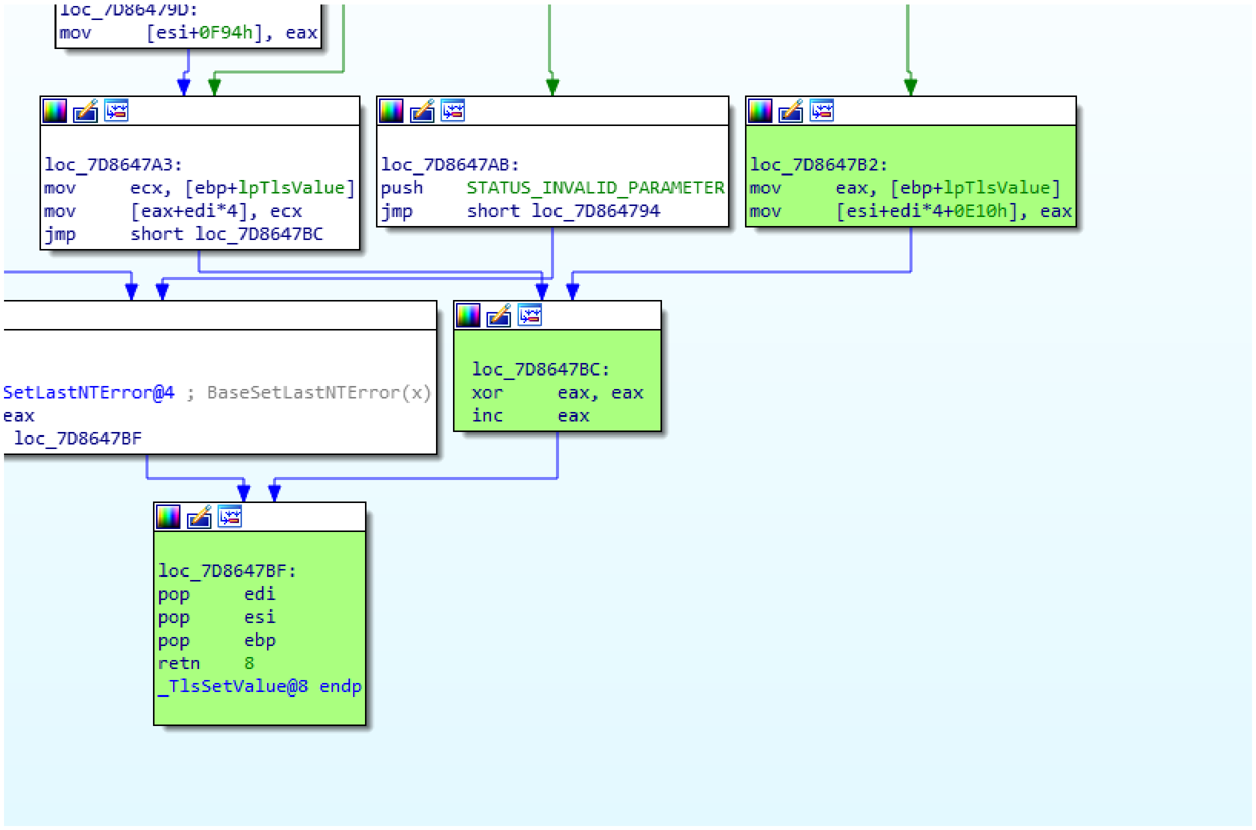
Here I will extend a second because it is important, if I do mov eax, fs: [0], what I am going to get is the value of fs: [0] and not its address, if I wanted to obtain the base address of the TEB I would have to use lea, but lea does not work with the segment fs (try it), the secret is that in fs: [18] the address of the base of the TEB is stored, then doing mov edi, fs: [18] I get the address of the TEB

Clarified this let's continue ...

The first thing we see is a check of our ID, checking if the value is less than or greater than 40 uhm ...

Remember, EDI have the Index and ESI the base of the TEB

Let's follow the path less than 40



```
loc_7D86479D:
mov     [esi+0F94h], eax
```

```
loc_7D8647A3:
mov     ecx, [ebp+lpTlsValue]
mov     [eax+edi*4], ecx
jmp     short loc_7D8647BC
```

```
loc_7D8647AB:
push    STATUS_INVALID_PARAMETER
jmp     short loc_7D864794
```

```
loc_7D8647B2:
mov     eax, [ebp+lpTlsValue]
mov     [esi+edi*4+0E10h], eax
```

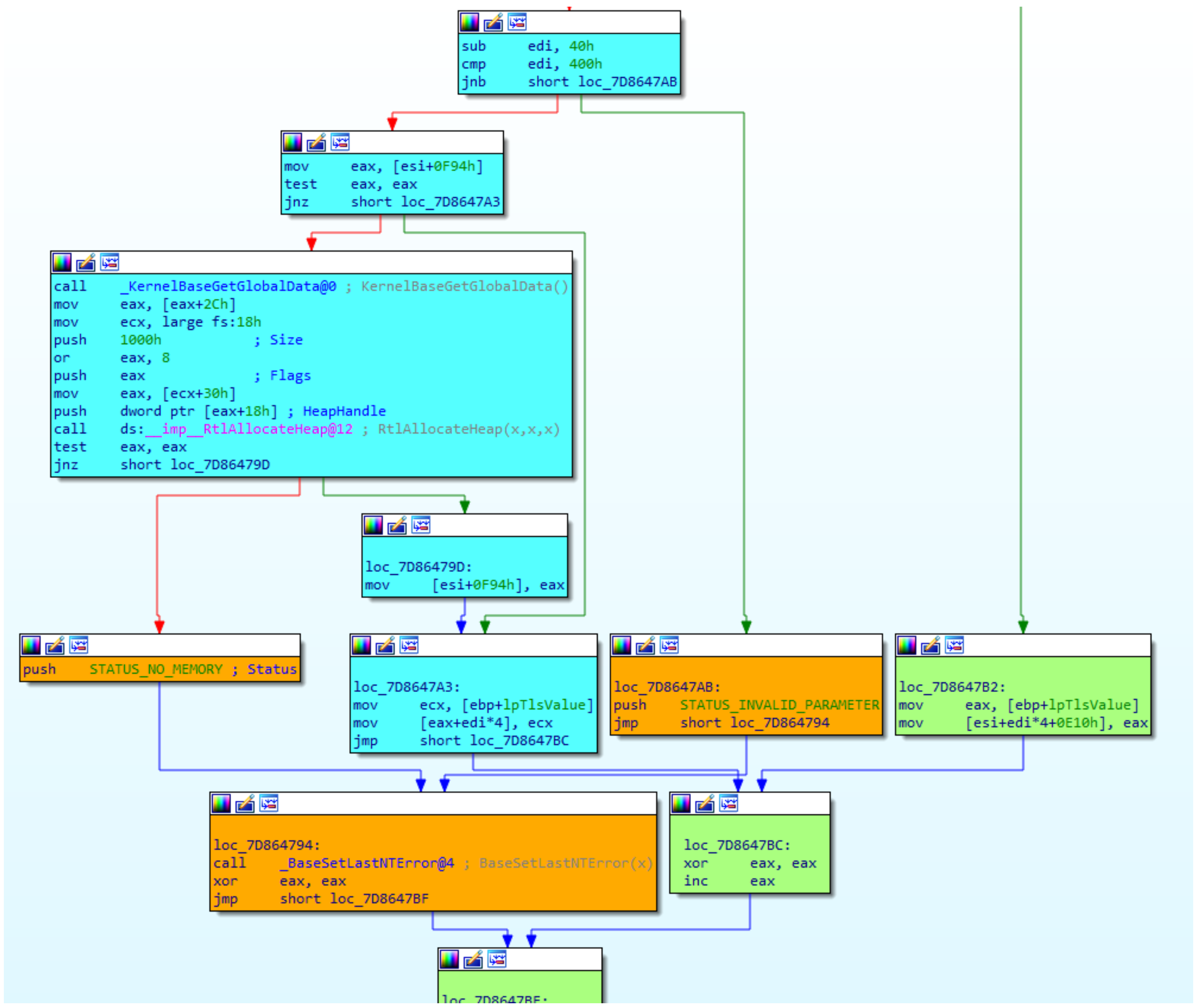
```
SetLastNTError@4 ; BaseSetLastNTError(x)
eax
 loc_7D8647BF
```

```
loc_7D8647BC:
xor     eax, eax
inc     eax
```

```
loc_7D8647BF:
pop     edi
pop     esi
pop     ebp
retn    8
_TlsSetValue@8 endp
```

We see something quite simple and clear, we load EAX with the value and then save it using mov, well as you see the mov makes the following calculation ESI + e10 + EDI * 4, ESI + e10 points to the first slot of the TlsSlots, EDI is our index that varies from 0 to 39 with this we can save in any of the TlsSlots.

Well so far we have 40 slots that would happen if we use an index greater than 39, well let's explore.



```
sub     edi, 40h
cmp     edi, 400h
jnb     short loc_7D8647AB
```

```
mov     eax, [esi+0F94h]
test    eax, eax
jnz     short loc_7D8647A3
```

```
call    _KernelBaseGetGlobalData@0 ; KernelBaseGetGlobalData()
mov     eax, [eax+2Ch]
mov     ecx, large fs:18h
push    1000h           ; Size
or      eax, 8
push    eax             ; Flags
mov     eax, [ecx+30h]
push    dword ptr [eax+18h] ; HeapHandle
call    ds:__imp__RtlAllocateHeap@12 ; RtlAllocateHeap(x,x,x)
test    eax, eax
jnz     short loc_7D86479D
```

```
loc_7D86479D:
mov     [esi+0F94h], eax
```

```
push    STATUS_NO_MEMORY ; Status
```

```
loc_7D8647A3:
mov     ecx, [ebp+lpTlsValue]
mov     [eax+edi*4], ecx
jmp     short loc_7D8647BC
```

```
loc_7D8647AB:
push    STATUS_INVALID_PARAMETER
jmp     short loc_7D864794
```

```
loc_7D8647B2:
mov     eax, [ebp+lpTlsValue]
mov     [esi+edi*4+0E10h], eax
```

```
loc_7D864794:
call    _BaseSetLastNTError@4 ; BaseSetLastNTError(x)
xor     eax, eax
jmp     short loc_7D8647BF
```

```
loc_7D8647BC:
xor     eax, eax
inc     eax
```

```
loc_7D8647BF:
```

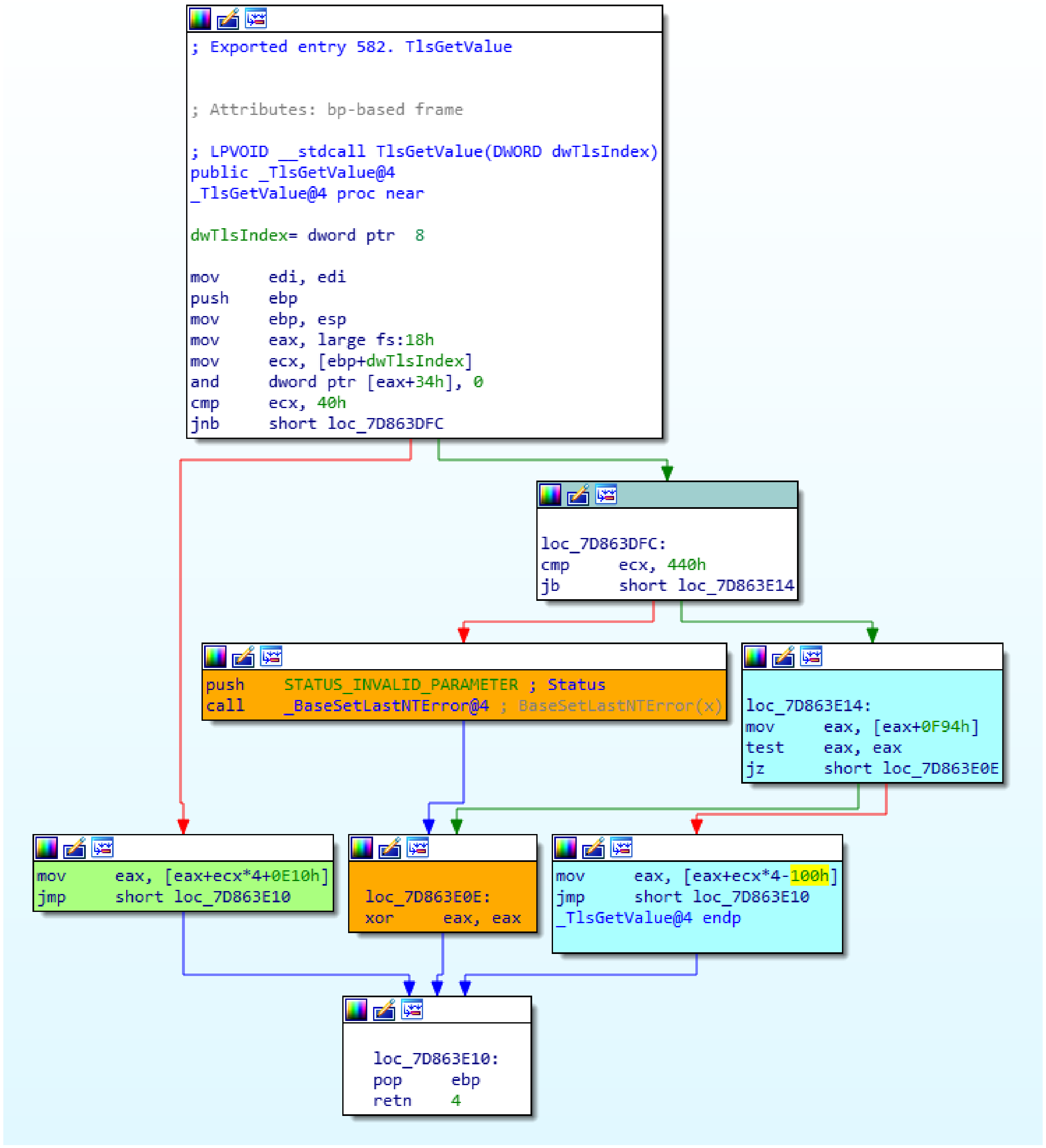
The first thing that is done is to subtract our index 40, which means that 40 will be our 0 for this part, here the upper limit of the index is checked, where it is compared with 400, if the index is greater than or equal to 400 then we return with STATUS_INVALID_PARAMETER.

Here we get critical information, how many total slots we have available. the sum gives us 40 + 400 = 440

Let's continue, move to eax, the value of [esi + 0f94], in that we had the base of the TEB, therefore what we are doing is reading the value in FS: [f94], that value is the TlsExpansionSlots, and check it that if it is not 0, if the value is 0 (here I will go a little fast, it is only a malloc) then it makes a HeapAlloc of 1000 and saves the pointer in fs: [f94] (1000 bytes = 400 dwords)

Now that we have the expansion we can access it, very similar to the method of the first 40 index with a mov [eax + edi * 4], ecx, where the address of the HeapAlloc is found in eax, and in edi the index, again this heap is considered as a list of dwords and we access it as such with the * 4, of course in ecx is the value we want to save

TlsGetValue



```
; Exported entry 582. TlsGetValue


; Attributes: bp-based frame

; LPVOID __stdcall TlsGetValue(DWORD dwTlsIndex)
public _TlsGetValue@4
_TlsGetValue@4 proc near

dwTlsIndex= dword ptr  8

mov     edi, edi
push    ebp
mov     ebp, esp
mov     eax, large fs:18h
mov     ecx, [ebp+dwTlsIndex]
and     dword ptr [eax+34h], 0
cmp     ecx, 40h
jnb     short loc_7D863DFC
```

```
loc_7D863DFC:
cmp     ecx, 440h
jb      short loc_7D863E14
```

```
push    STATUS_INVALID_PARAMETER ; Status
call    _BaseSetLastNTError@4 ; BaseSetLastNTError(x)
```

```
loc_7D863E14:
mov     eax, [eax+0F94h]
test    eax, eax
jz      short loc_7D863E0E
```

```
mov     eax, [eax+ecx*4+0E10h]
jmp     short loc_7D863E10
```

```
loc_7D863E0E:
xor     eax, eax
```

```
mov     eax, [eax+ecx*4-100h]
jmp     short loc_7D863E10
_TlsGetValue@4 endp
```

```
loc_7D863E10:
pop     ebp
retn    4
```

The analysis of TlsGetValue is quite trivial, the function takes the parameter of the index and returns a dword

Again the trick of fs: [18] appears but we know what it is, we check if it is less than 40, if it is, take the green path, if the value is between 40 and 399 then take the cyan path, if the value is outside that range returns 0 and with NT error. The cyan path reads the value of FS: [f94] which was the value of the expansion heap, and accesses it as explained in the TlsSetValue,

Something interesting is that in this case the mov eax, [eax + ecx * 4-100h] contains a -100 this is because in the TlsSetValue I had subtracted 40 from the index when it was an index that pointed to the expansion area, in this case it did not subtract but (40index = 40dwords = 100bytes) does it directly in the reading.

A particularity that we see in the first box is the and [eax + 34h], 0, remember when I said that all the information related to the thread is stored in the TEB? Well in fs: [34] the LastError is saved, doing and 0 we clean it, but why?

In this case, returning 0 of the function is not enough to determine if there was a failure or not, a slot could have the value 0 stored and that would cause it to return 0 which would be perfectly normal, then to solve this problem in addition to returning 0, The NT error system is used to report the failure. I clarify this because in the TLSSetValue, the return value was Boolean, where if 0 was returned it meant that there was a fault, the NT error system only extends about the failure, but is not critical.

DSTN @MZ_IAT

Thanks to @Farenain.