Understanding PEB and LDR Structures using IDA and LummaStealer

By map[name:Alessandro Strino]

Published: 2024-02-04 · Archived: 2026-04-05 15:37:14 UTC

In this post I'm going to explain how Process Environment Block (PEB) is parsed by malware devs and how that structure is abused. Instead of going too deep into a lot of details, I would like to follow an easier approach pairing the theory with a practical real example using IDA and LummaStealer, without overwhelming the reader with a lot of technical details trying to simplify the data structure involved in the process. At the end of the theory part, I'm going to **apply PEB and all related structures in IDA**, inspecting malware parsing capabilities that are going to be applied for resolving hashed APIs.

Let's start.

PEB Structure

The PEB is a crucial data structure that contains various **information** about a **running process**. Unlike other Windows structure (e.g., EPROCESS, ETHREAD, etc..), it exists in the user address space and is available for every process at a **fixed** address in memory (PEB can be found at `fs:[0x30]` in the Thread Environment Block (TEB) for x86 processes as well as at `gs:[0x60]` for x64 processes). Some of documented fields that it's worth knowing are:

- BeingDebugged: Whether the process is being debugged;
- **Ldr**: A pointer to a `PEB_LDR_DATA` structure providing information about loaded modules;
- ProcessParameters: A pointer to a `RTL_USER_PROCESS_PARAMETERS` structure providing information about process startup parameters;
- PostProcessInitRoutine: A pointer to a callback function called after DLL initialization but before the main executable code is invoked

Image Loader aka Ldr

When a process is started on the system, the kernel creates a process object to represent it and performs various kernel-related initialization tasks. However, these tasks do not result in the execution of the application, but in the **preparation of its context and environment**. This work is performed by the ***image loader*** (**Ldr**).

The loader is responsible for several main tasks, including:

- Parsing the import address table (IAT) of the application to look for all DLLs that it requires (and then recursively parsing the IAT of each DLL), followed by parsing the export table of the DLLs to make sure the function is actually present.

- Loading and unloading DLLs at runtime, as well as on demand, and maintaining a list of all loaded modules (the module database).

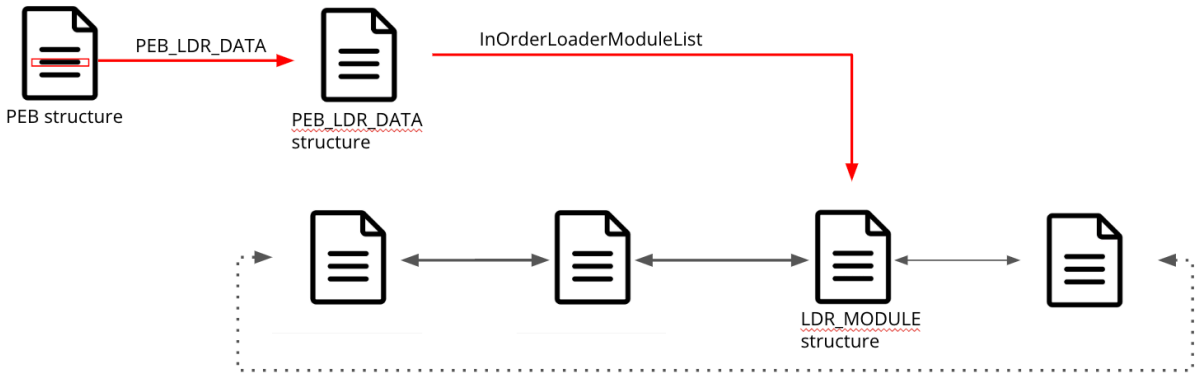

Figure 1: PEB, LDR_DATA and LDR_MODULE interactions

At first glance, these structures might seem a little bit confusing. However, let's simplify them to make them more understandable. We could think about them as a **list where the structure PEB_LDR_DATA is the head of the list** and **each module information is accessed through a double linked list** (*InOrderLoaderModuleList* in this case) that points to **LDR_MODULE**.

How those structures are abused

Most of the times when we see **PEB and LDR_MODULE structure parsing** we are dealing with malwares that are potentially using **API Hashing technique.** Shellcode will typically walk through those structures in order to find the base address of loaded dlls and extract all their exported functions, collecting names and **pointers to the functions that are intended to call**, avoiding to leave direct reference of them within the malware file.

This is a simple trick that tries to evade some basic protections mechanism that could arise when we see clear references to malware-related functions such as: *VirtualAlloc, VirtualProtect, CreateProcessInterW, ResumeThread, etc…*

API Hashing

By employing API hashing, malware creators can ensure that specific Windows APIs remain hidden from casual observation. Through this approach, malware developers try to add an extra layer of complexity by concealing suspicious Windows API calls within the Import Address Table (IAT) of PE.

API hashing technique is pretty straightforward and it could be divided in three main steps:

1. *Malware developers prepare a set of hashes corresponding to WINAPI functions.*
2. *When an API needs to be called, it looks for loaded modules through the PEB.Ldr structure.*
3. *Then, when a module is find, it goes through all the functions performing the hash function until the result matches with the given input.*

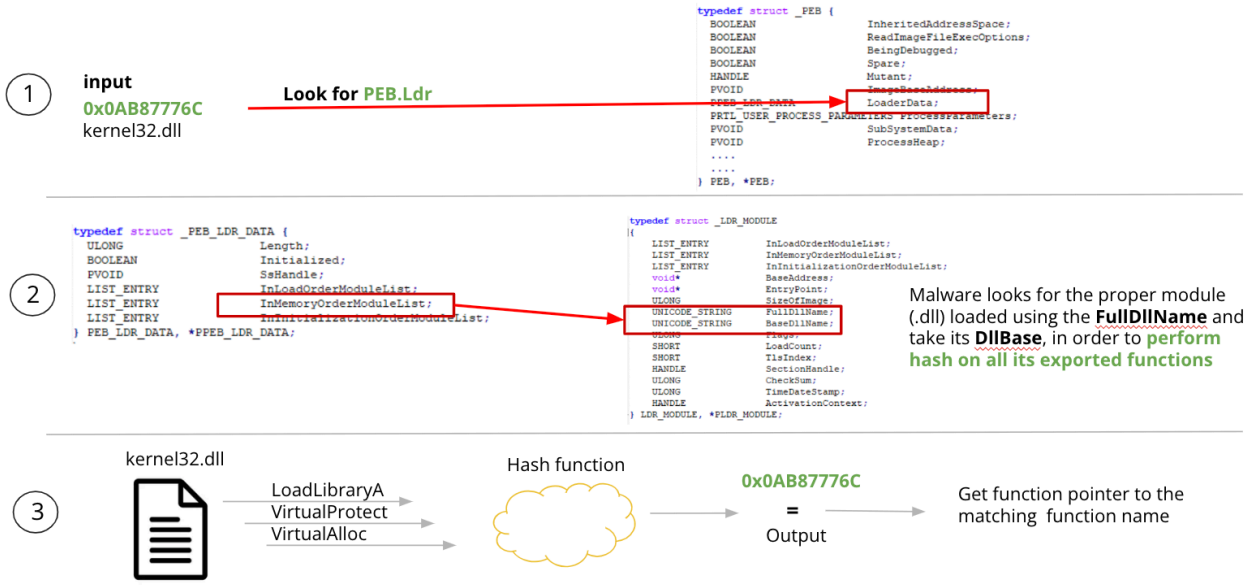

Figure 2: API Hashing Overview

Now that we have a more understanding of the basic concepts related to API hashing, PEB and Ldr structures, let's try to put them in practice using LummaStealer as an example.

Parsing PEB and LDR with LummaStealer

Opening up the sample in IDA and scrolling a little after the main function it is possible to bump into very interesting functions that perform some actions on a couple of parameters that are quite interesting and correlated to explanation so far.

```
.text:00407ACE
.text:00407AD3 59                      pop     ecx
.text:00407AD4 59                      pop     ecx
.text:00407AD5 8B F8                   mov     edi, eax
.text:00407AD7 BA 10 3E 42 00          mov     edx, offset aKernel32Dll_0      ; "kernel32.dll"
.text:00407ADC 53                      push    ebx
.text:00407ADD 57                      push    edi
.text:00407ADE 56                      push    esi
.text:00407ADF B9 05 F5 28 73          mov     ecx, 7328F505h
.text:00407AE4 E8 EA 07 00 00          call    sub_4082D3
.text:00407AE4
.text:00407AE9 FF D0                   call    eax
.text:00407AE9
.text:00407AEB 8B CF                   mov     ecx, edi
.text:00407AED E8 0B B0 FF FF          call    sub_402AFD
.text:00407AED
```

Figure 3: Wrapper function for hash resolving routine in LummaStealer

Before function ***call sub_4082D3*** (highlighted) we could see some mov operation of two values:

Those parameters are quite interesting because:

- The former represents an interesting dll that contains some useful functions such as *LoadLibrary, VirtualAlloc, etc.*.
- The latter appears to be a hash (maybe correlated to the previous string).

If we would like to make an educated guess, it is possible that this function is going to find a function (within *kernel32.dll*) whose hash corresponds to the input hash. However, let's try to understand if and how those parameters are manipulated in the function call, validating also our idea.

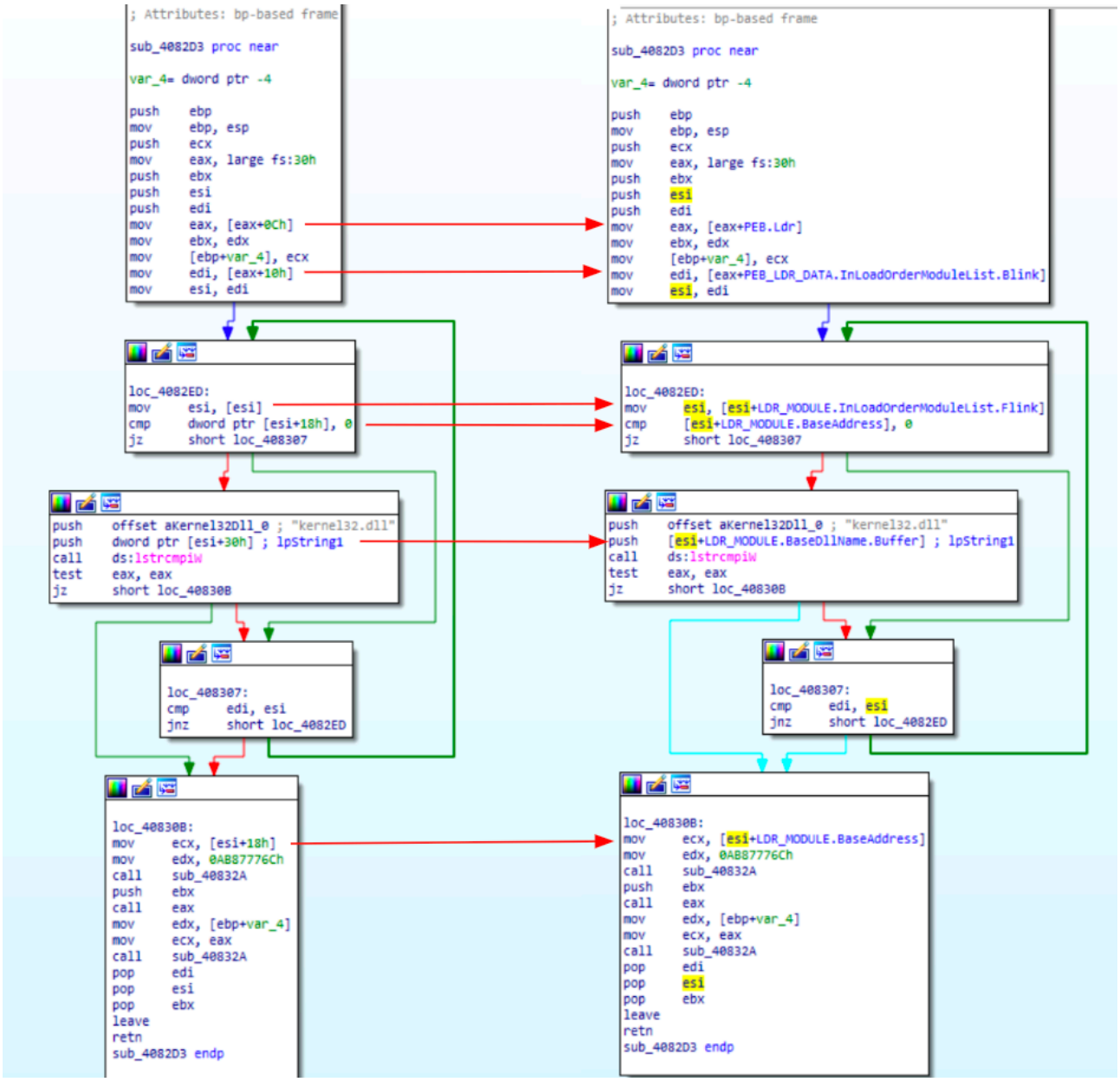

Figure 4: Parsing PEB and LDR_MODULE for API hash routine.

Through Figure 6, you can see the exact same code, before (left side) and after (right side) renaming structures. Examining the code a little bit we should be able to recall the concepts already explained in the previous sections.

Let's examine the first block of code. Starting from the top of the code we could spot the instruction `mov eax, (large)fs:30h` that is going to collect the **PEB pointer**, storing its value in **eax.** Then, right after this instruction we could see **eax used with an offset(0xC)**. In order to understand what is going on, its possible to collect the PEB structure and look for the 0xC offset. Doing that, it's clear that **eax is going to collect the Ldr pointer**. The last instruction of the first block is `mov edi, [eax+10h]` . This is a crucial instruction that needs a dedicated explanation:

If you are going to look at **PEB_LDR_DATA** you will see that 0x10 offset (for x64 bit architecture) points to **InLoadOrderModuleList** (that contains, according to its description, `pointers to previous and next LDR_MODULE in initialization order`). Through this instruction, malware is going to take a **LDR_MODULE** structure (as explained in Figure 3), settling all the requirements to parse it.

Without going too deep in the code containing the loop (this could be left as an exercise), it is possible to see that the next three blocks are going to find the *kernel32.dll* iterating over the **LDR_MODULE** structure parameters.

At the very end of the code, we could see the last block calling a function using the dll pointers retrieved through the loop, using another hash value. This behavior give us another chance for a couple of insight:

- This code is a candidate to settle all parameters that are going to be used for API hash resolving routine (as illustrated in the API Hashing section), since that its output will be used as a function call.
- The string *kernel32.dll* gave us some hints about possible candidate functions (e.g., *LoadLibraryA, VirtualAlloc, etc..*).

With this last consideration, it's time to conclude this post avoiding adding more layers of complexity, losing our focus on PEB and related structures.

Function recap

Before concluding, let's try to sum up, what we have seen so far, in order to make the analysis even more clear:

1. The function `4082D3` takes **two parameters** that are a hash **value** and a string containing a **dll library**.
2. Iterating over the loaded modules, it looks for the module name containing the hardcoded **kernel32.dll**.
3. Once the module is found, it invokes another function (`40832A`), passing a pointer to the base address of the module and a hash value.
4. The function returns a pointer to a function that takes as an argument the **dll name passed to** 4082D3. This behavior suggests that some sort of *LoadLibrary* has been resolved on point 3.
5. As a final step, the function `40832A` is called once again, using the hash value passed as a parameter in the function `4082D3` and a base address retrieved from the point 4.

Following all the steps it's easy to spot that the `40832A` function is the actual API hash resolving routine and the function `4082D3` has been used to settle all the required variables.

Conclusion

Through this blog post I tried to explain a little bit better how the PEB and related structures are parsed and abused by malwares. However, I also tried to show how malware analysis could be carried out examining the code and renaming structures accordingly. This brief introduction will be also used as a starting point for the next article where I would like to take the same sample and emulate the API hashing routine in order to resolve all hashes, making this sample ready to be analyzed.

Note about simplification

It’s worth mentioning that to make those steps easier, there has been a simplification. In fact, PEB_LDR_DATA contains three different structures that could be used to navigate modules, but for this blogpost, their use could be ignored. Another structure that is worth mentioning it’s LDR_DATA_TABLE_ENTRY that could be considered a corresponding to the LDR_MODULE structure.

References:

Undocumented Windows Internal Structures:

- The Undocumented Functions
- PEB_LDR_DATA

LummaSteler sample:

- MalwareBazaar
