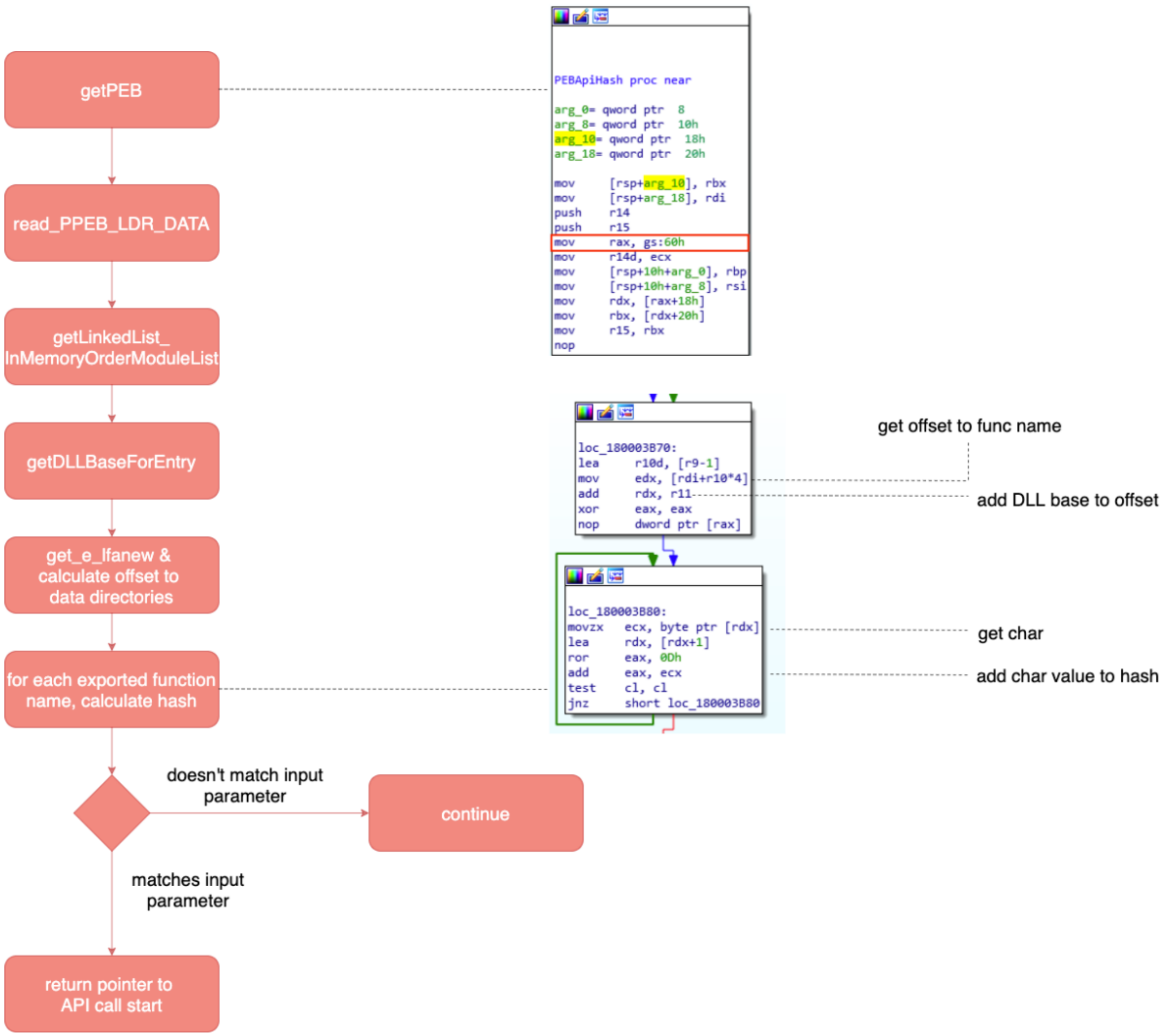
PEB: Where Magic Is Stored

malwareandstuff.com/peb-where-magic-is-stored/

August 22, 2021



Published by **hackingump** on August 22, 2021

As a reverse engineer, every now and then you encounter a situation where you dive deeper into the internal structures of an operating system as usual. Be it out of simple curiosity, or because you need to understand how a binary uses specific parts of the operating system in certain ways . One of the more interesting structures in Windows is the Process Environment Block/PEB. In this article, I'd like to introduce you to this structure and talk about various use cases of how adversaries can abuse this structure for their own purposes.

Introducing PEB

The Process Environment Block is a critical structure in the Windows OS, most of its fields are not intended to be used by other than the operating system. It contains data structures that apply across a whole process and is stored in user-mode memory, which makes it accessible for the corresponding process. The structure contains valuable information about the running process, including:

- whether the process is being debugged or not
- which modules are loaded into memory
- the command line used to invoke the process

All these information gives adversaries a number of possibilities to abuse it. The figure below shows the layout of the `PEB` structure:

```
typedef struct _PEB {
  BYTE                          Reserved1[2];
  BYTE                          BeingDebugged;
  BYTE                          Reserved2[1];
  PVOID                         Reserved3[2];
  PPEB_LDR_DATA                 Ldr;
  PRTL_USER_PROCESS_PARAMETERS  ProcessParameters;
  PVOID                         Reserved4[3];
  PVOID                         AtlThunkSListPtr;
  PVOID                         Reserved5;
  ULONG                         Reserved6;
  PVOID                         Reserved7;
  ULONG                         Reserved8;
  ULONG                         AtlThunkSListPtr32;
  PVOID                         Reserved9[45];
  BYTE                          Reserved10[96];
  PPS_POST_PROCESS_INIT_ROUTINE PostProcessInitRoutine;
  BYTE                          Reserved11[128];
  PVOID                         Reserved12[1];
  ULONG                         SessionId;
} PEB, *PPEB;
```

Now that we’ve talked a little bit about the layout and purpose of the structure, let’s take a look at a few use cases.

Reading the BeingDebugged flag

The most obvious way is to check the `BeingDebugged` to identify, whether a debugger is attached to the process or not. Through reading the variable directly from memory instead of using usual suspects like `NtQueryInformationProcess` or `IsDebuggerPresent` , malware can prevent noisy WINAPI calls. This makes it harder to spot this technique.

However, most debuggers already take care of this. `X64dbg` for example, has an option to hide the Debugger by modifying the PEB structure at start of the debugging session.

Iterating through loaded modules

Another use case, could be iterating the loaded modules and discover DLLs injected into memory with purpose to overwatch the running process. To understand how to achieve this, we need to take a look at the `PPEB_LDR_DATA` structure included in `PEB`, which is provided by the `Ldr` variable:

```
typedef struct _PEB_LDR_DATA {
  BYTE       Reserved1[8];
  PVOID      Reserved2[3];
  LIST_ENTRY InMemoryOrderModuleList;
} PEB_LDR_DATA, *PPEB_LDR_DATA;
```

`PPEB_LDR_DATA` contains the head to a doubly linked list named `InMemoryOrderModuleList`. Each item in this list is a structure from type `LDR_DATA_TABLE_ENTRY`, which contains all the information we need to iterate loaded modules. See the structure of `LDR_DATA_TABLE_ENTRY` below:

```
typedef struct _LDR_DATA_TABLE_ENTRY {
    PVOID Reserved1[2];
    LIST_ENTRY InMemoryOrderLinks;
    PVOID Reserved2[2];
    PVOID DllBase;
    PVOID EntryPoint;
    PVOID Reserved3;
    UNICODE_STRING FullDllName;
    BYTE Reserved4[8];
    PVOID Reserved5[3];
    union {
        ULONG CheckSum;
        PVOID Reserved6;
    };
    ULONG TimeDateStamp;
} LDR_DATA_TABLE_ENTRY, *PLDR_DATA_TABLE_ENTRY;
```

So by iterating the doubly linked list, we are able to discover the base address and full name of all modules loaded into memory of the running process. The snippet below is a small Proof of Concept. It iterates the linked list and prints the library name to stdout. I created it for the purpose of this blog article. You are free to use it, however I will also upload it to my github repo the upcoming days:

```
#include <Windows.h>
#include <iostream>
#include <shlwapi.h>


#define NO_STDIO_REDIRECT

typedef struct _UNICODE_STRING
{
    USHORT Length;
    USHORT MaximumLength;
    PWSTR Buffer;
} UNICODE_STRING, * PUNICODE_STRING;


typedef struct _LDR_DATA_TABLE_ENTRY_MOD {
    LIST_ENTRY InMemoryOrderLinks;
    PVOID Reserved2[2];
    PVOID DllBase;
    PVOID EntryPoint;
    PVOID Reserved3;
    UNICODE_STRING FullDllName;
    BYTE Reserved4[8];
    PVOID Reserved5[3];
    union {
        ULONG CheckSum;
        PVOID Reserved6;
    };
    ULONG TimeDateStamp;
} LDR_DATA_TABLE_ENTRY_MOD, * PLDR_DATA_TABLE_ENTRY_MOD_MOD;




int main(int argc, char** argv[]){


    PLDR_DATA_TABLE_ENTRY_MOD_MOD lib = NULL;
    _asm {
        xor eax, eax
        mov eax, fs:[0x30]
        mov eax, [eax + 0xC]
        mov eax, [eax + 0x14]
        mov lib, eax
    };
    printf("[+] Initialised pointer to first LDR_DATA_TABLE_ENTRY_MOD\n");


    // Loop as long as we don't reach the head of the linked list again
    while ( lib->FullDllName.Buffer != NULL ) {

        printf("[+] %S\n", lib->FullDllName.Buffer);
        lib = (PLDR_DATA_TABLE_ENTRY_MOD_MOD)lib->InMemoryOrderLinks.Flink;
    }
```

```
    printf("[+] Done!\n");


        return 0;
```

If you are wondering how I am able to access the `PEB` in the code below, you should take a look at the inline assembly in the `main` method, especially the instruction `mov eax, fs:[0x30]`. FS is a segment register, similar to GS. FS can be used to access thread-specific memory. Offset `0x30` allows you to access the linear address of the Process Environment Block.

Finally, we want to take a look at a real world example of how `PEB` can be abused.

How the MATA Framework abuses PEB

This use case was introduced to me while reverse engineering a Windows variant of the MATA Framework. According to Kaspersky[1], the MATA Framework is used by the Lazarus group and targets multiple platforms.

Malware authors have a high interest in obfuscation, because it increases the time needed to reverse engineer it. One way to hide API calls is to use API Hashing. I have written about Danabot's API Hashing[2] before and how to overcome it. MATA also uses this technique.

However instead of using the WIN API calls to retrieve the address of DLLs loaded into memory, MATA abuses the Process Environment Block to fetch base addresses. Let's take a look at how MATA for Windows achieves this:

MATA API Hashing

The input of the `APIHashing` method takes an integer as the only parameter, this is the hash for the corresponding API call.

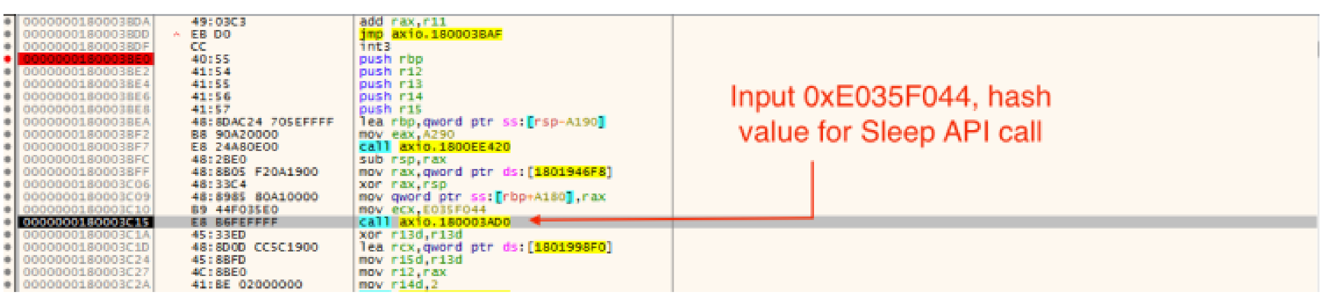
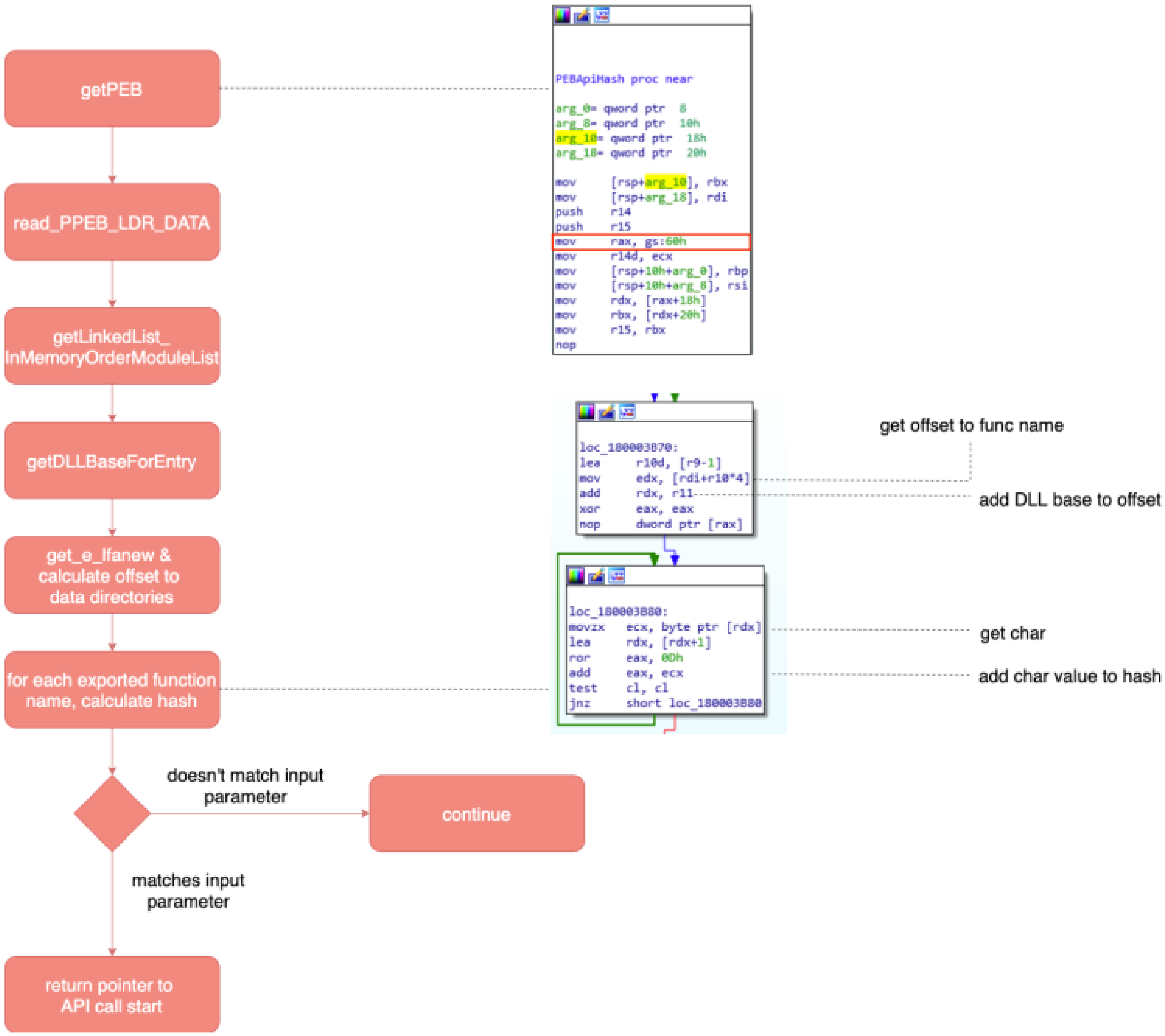


Figure 1: Call to APIHash method

Right after the prologue, it retrieves a pointer to `PEB` by reading it from the Thread Environment Block via the segment register `GS`. Similar to our proof of concept above, MATA now fetches the address to the head of the linked list provided by `InMemoryOrderModuleList`. Each item of the linked list provides the DLL base address of the corresponding loaded module.

From there, the malware reads the `e_lfanew` field, which contains the offset to the file header. By adding the base address, `e_lfsanew` and `0x88` it jumps directly to the data directories of the corresponding PE. From the data directories, MATA accesses the exported function names in a similar way as I've described in my blog article about DanaBot's API Hashing[3]. The hashing algorithm is fairly simple. Each integer representation of a character is added and the result of the addition is `ROR'd` by `0xD` consecutively each iteration. If the final hash matches the input parameter, the address to the function is retrieved. The following figure explains the function at a high level:



High level overview of API Hashing of MATA malware

Learning from each other

That's it with the blog article, I hope you enjoyed it! There are probably way more use cases and real world cases of how the `PEB` is and and can be abused. If you can think of another one, feel free to leave a comment below and share it, so that we can learn from each other!