Traumatic Library Loading

If you want to use it, you have to implement it...

Defcon 32

And why should we trust you ?

Muggle identity

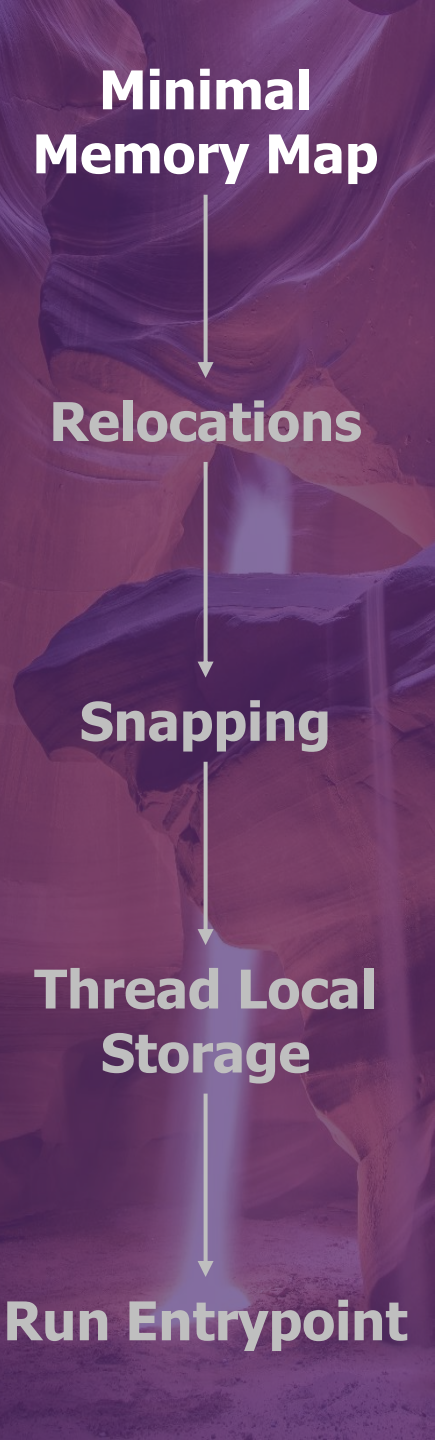
- Yoann DEQUEKER (*@OtterHacker*)
- 27 yo
- Personal website: *otterhacker.github.io*
- OSCP, Cybernetics …

Experience

- Senior pentester @*Wavestone* for almost 5 years
- Dedicated to large-scale *RedTeam* operations – *CAC40* companies
- Development of internal tooling – Mainly malware and Cobalt
- Uncommon process injection pattern – @LeHack 2023, @Defcon31, @Insomnihack 2024

/ **01**

Minimal DLL Loading



From file to memory
> Load your DLL in your process memory

The *Minimal Memory Map* step is used to load the *DLL* in memory. This step involves allocating the memory space, and mapping the *DLL* headers and sections at the expected address.

Step 1 : Read the DLL from the disk and store it in memory

> The first step is **to retrieve the *DLL* content** with a simple file read.
> The *DLL* is stored in memory but is **not mapped to be executed**

Step 2 : Allocate the memory space for the DLL

> **Find the size** of your *DLL*: this information is stored in the *DLL Optional Header*
> **Allocate a memory** space using *VirtualAlloc*

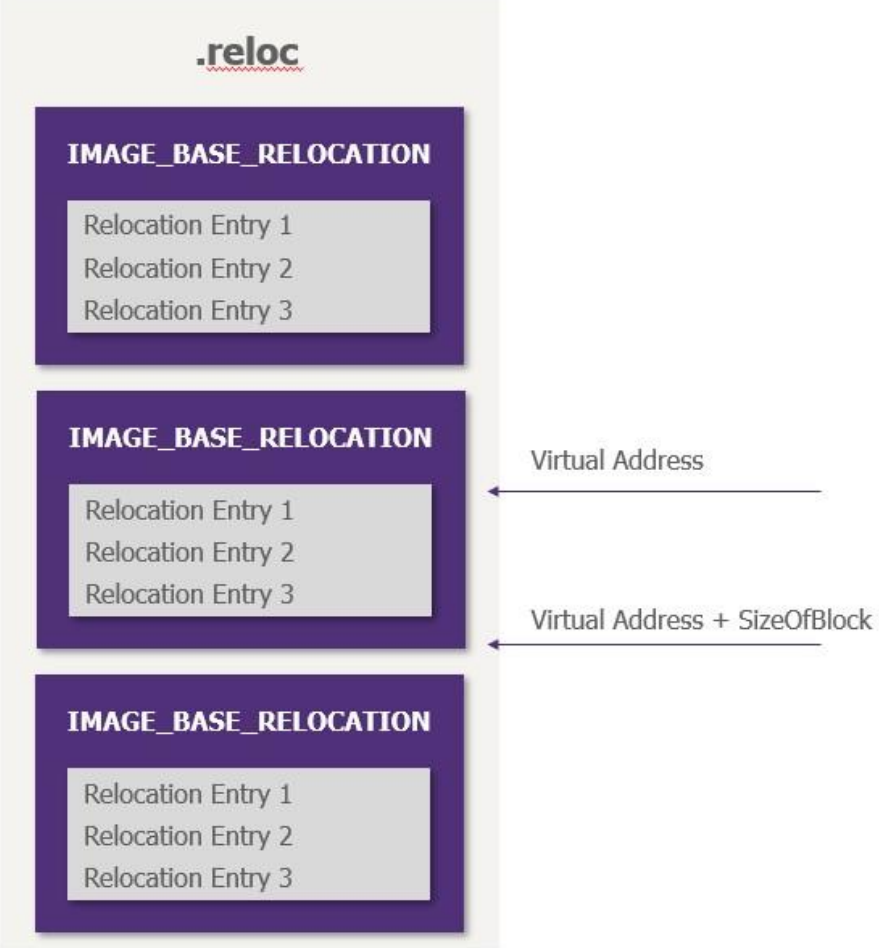
Step 3 : Map the DLL in memory

> **Copy the *DLL* header** on the allocated memory space
> **Copy the *DLL* sections** according to the *DLL* mapping
> All the information can be found in the *DLL* header and in the sections headers

Open the AMSI.DLL with PEBear, try to locate the different information needed for this step: the DLL size, the header size, and the address and size of each section.

Take the loading offset into accounts
> Use relocations to resolve the absolute references

The *DLL* is **compiled to be loaded to a specific address**. If the *DLL* is not loaded at this address, the address shift will **break all absolute references**. The *.reloc* section can be used to identify these references and fix the offset



Step 1 : Parse the relocation information

> The *.reloc* section contains different blocks, each of them containing different relocation entries pointing on specific references

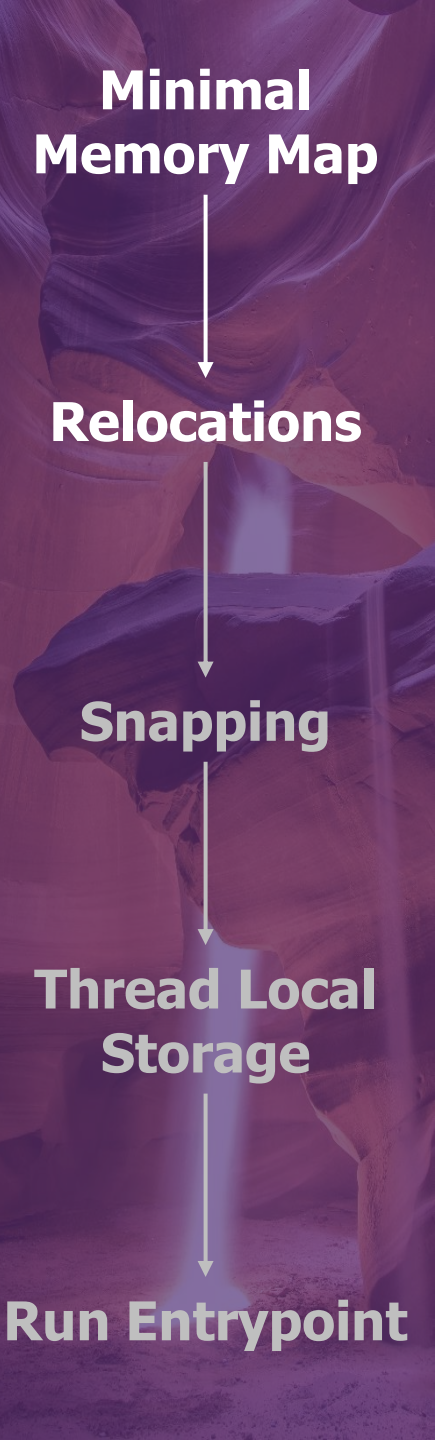
Step 2 : Iterate over IMAGE_BASE_RELOCATION

> Each relocation entry contains two information: the offset and the type. The offset is used to locate the address that must be relocated, the type indicates how the relocation must be applied

Step 3 : Apply the relocation

> Locate the address to relocate using the offset, apply the relocation according to the relocation type

Take the loading offset into accounts
> Use relocations to resolve the absolute references



Being able to parse the different relocation information can be challenging. All you need to know is how to **retrieve the first relocation** bloc address, how to **jump to the following block** and how to **navigate through the relocation** entries

How to parse the relocation information

> The first relocation block address **is pointed by the *DataDirectory*** stored in the *DLL* header

```
typedef struct _IMAGE_BASE_RELOCATION {
    DWORD   VirtualAddress;
    DWORD   SizeOfBlock;
} IMAGE_BASE_RELOCATION;
```

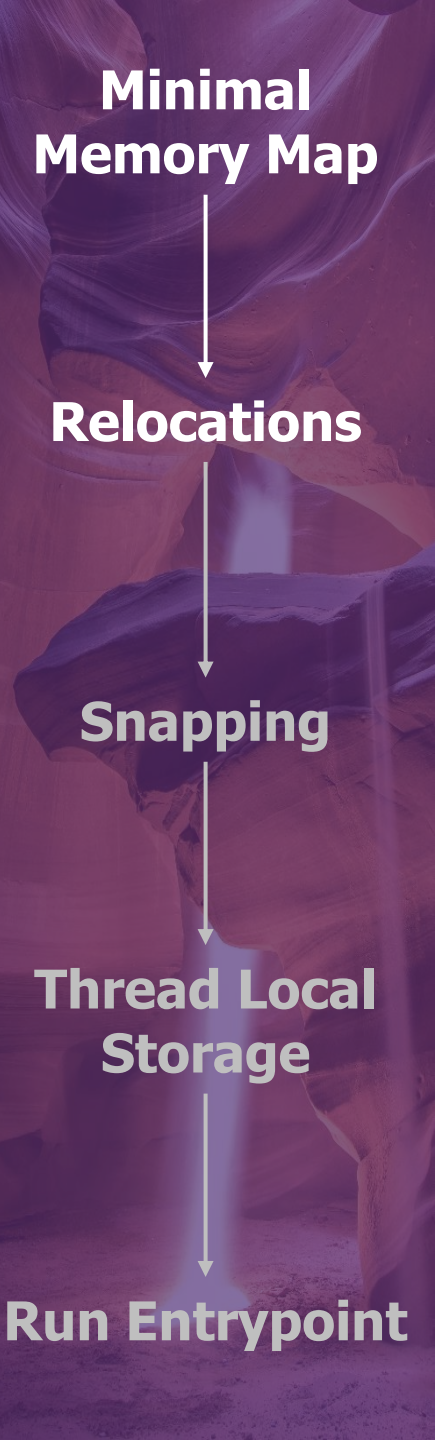
How to parse the relocation entry

> Each relocation entry can be parsed using a *C* structure
> The address to relocate is **pointed by** the **relocation entry address + *Offset***

```
typedef struct _IMAGE_RELOCATION_ENTRY {
    WORD Offset : 12;
    WORD Type : 4;
} IMAGE_RELOCATION_ENTRY;
```

How to apply the relocation

> Applying a relocation is **just adding the offset |*ExpectedLoadAddress – CurrentLoadAddress*|** to the address pointed by the relocation



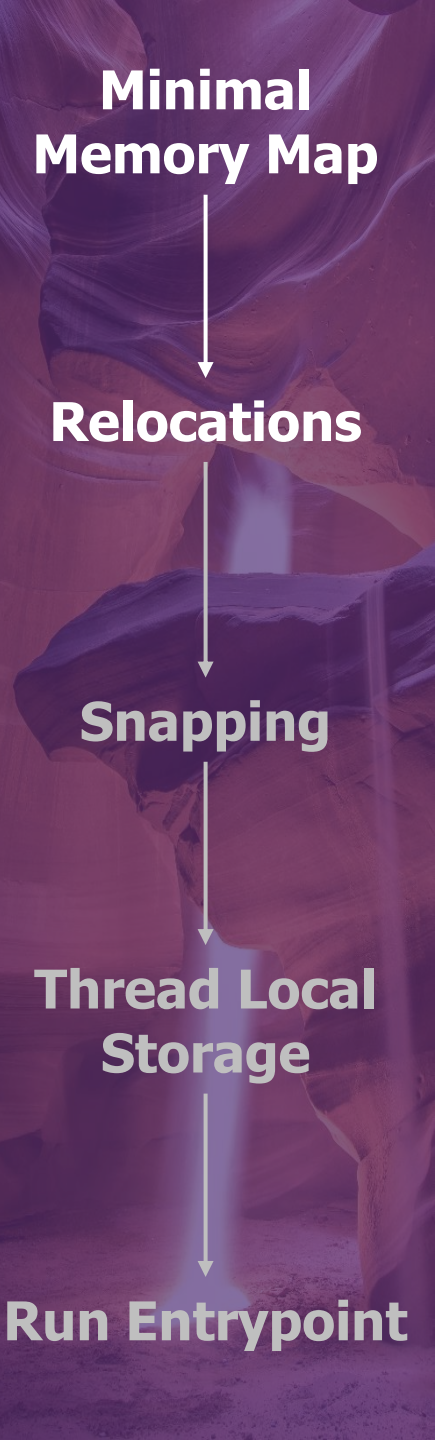
Take the loading offset into accounts
> Use relocations to resolve the absolute references

How to apply the relocation

> Applying a relocation is just **adding the offset |*ExpectedLoadAddress – CurrentLoadAddress*|** to the address pointed by the relocation

Name	Value	Description
IMAGE_REL_BASED_ABSOLUTE	0x00	The relocation is skipped. It is often used to pad a block
IMAGE_REL_BASED_HIGH	0x01	The 16 high bits of the offset is added to the current relocation value
IMAGE_REL_BASED_LOW	0x03	The 16 low bits of the offset is added to the current relocation value
IMAGE_REL_BASED_HIGHLOW	0x04	The 32 bits of the offset is added to the current relocation value
IMAGE_REL_BASED_DIR64	0x10	The 64 bits of the offset is added to the current relocation value

Take the loading offset into accounts
> Use relocations to resolve the absolute references

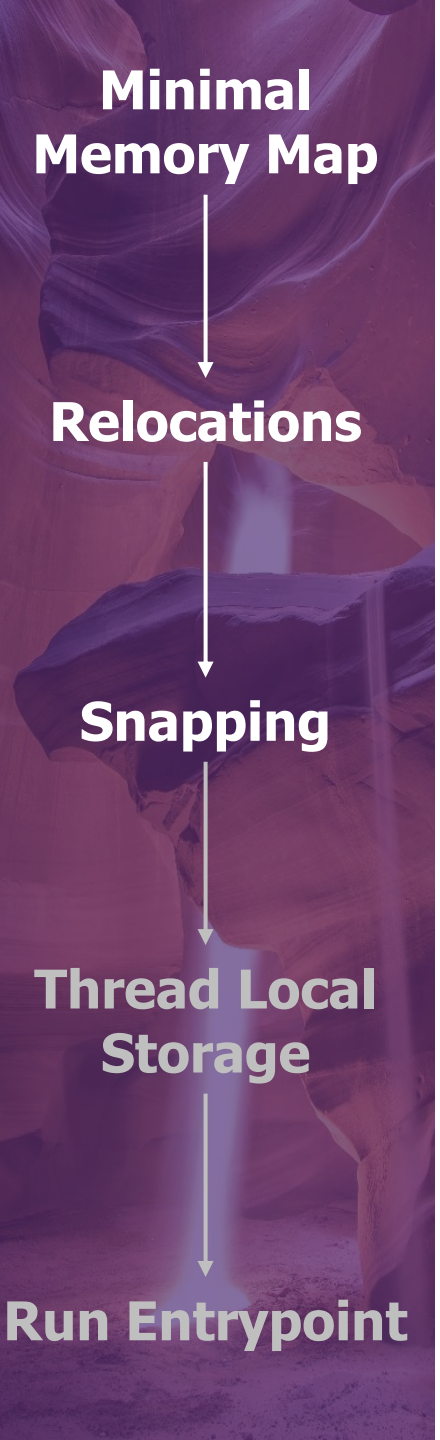



```
================================================================
# current_entry point to the following structure
typedef struct _IMAGE_RELOCATION_ENTRY {
    WORD Offset : 0x88003;
    WORD Type : 1;
} IMAGE_RELOCATION_ENTRY;
================================================================

// We find the address to relocate using the relocation bloc address and the
// offset given in the relocation entry
relocation_address = (DWORD64)relocation_bloc->VirtualAddress + current_entry_address->Offset

// We compute the offset to add to the relocated address using the expected image load
// and the current image load
offset = pe->expected_image_address - pe->current_image_address

// We apply the relocation according to the relocation entry type
*relocation_address += LOWORD(offset)
```



Resolve the external functions
> Load additional references and resolve the external functions

The *DLLs* can **use functions defined in other *DLL***, this is a recursive process. When you are loading a *DLL*, you **must resolve the external functions address**.

The Import Directory Table

> The import directory table, whose address is **pointed in the *DataDirectory*** contains the name of all the *DLL* needed by the application to work. Each *DLL* referenced in this table **must be loaded during the *DLL* loading process**

IAT and ILT

> The *ILT* is a table containing the **name of all functions that must be resolved in the *DLL*** pointed in the *Import Directory Table*
> The *IAT* is **where the resolved function address will be stored** to be used by the *DLL*

Resolve the functions

> For each entry in the *Import Directory Table*, we load the associated *DLL*
> We retrieve the *ILT* and *IAT* address for the current entry
> For each function contained in the *ILT*, we resolve it and store the resolved address in the *IAT*

Resolve the external functions
> Load additional references and resolve the external functions

Minimal Memory Map

Relocations

Snapping

Thread Local Storage

Run Entrypoint

Some *DLL* are not always needed and will **be loaded just in time** only when a specific function in the *DLL* will be used. This is **called the delayed *DLL imports***. These *DLL* are given in the *Delayed Import Table.*

Use PEBear to analyze the Import and Delayed Import table

> Load the AMSI.DLL function in *PEBear* and **look at the Import table and the Delayed Import table**

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    union {
        DWORD   Characteristics;
        DWORD   OriginalFirstThunk;
    } DUMMYUNIONNAME;
    DWORD   TimeDateStamp;
    DWORD   ForwarderChain;
    DWORD   Name;
    DWORD   FirstThunk;
} IMAGE_IMPORT_DESCRIPTOR;
```

Import Table structure

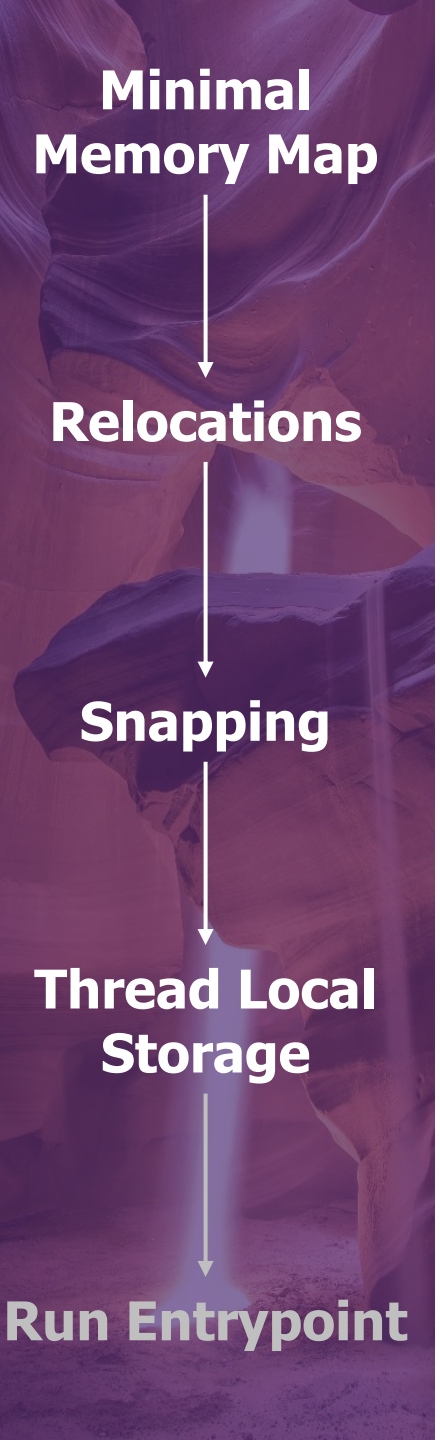
```
typedef struct _IMAGE_DELAYLOAD_DESCRIPTOR {
    union {
        DWORD AllAttributes;
        struct {
            DWORD RvaBased : 1;
            DWORD ReservedAttributes : 31;
        } DUMMYSTRUCTNAME;
    } Attributes;

    DWORD DllNameRVA;
    DWORD ModuleHandleRVA;
    DWORD ImportAddressTableRVA;
    DWORD ImportNameTableRVA;
    DWORD BoundImportAddressTableRVA;
    DWORD UnloadInformationTableRVA;
    DWORD TimeDateStamp;
```

Delayed Import Table structure

```
typedef struct _IMAGE_THUNK_DATA {
    union {
        ULONGLONG ForwarderString;
        ULONGLONG Function;
        ULONGLONG Ordinal;
        ULONGLONG AddressOfData;
    } u1;
} IMAGE_THUNK_DATA;
```

IAT and ILT structure



TLS and SEH
> Initialize the critical Windows features

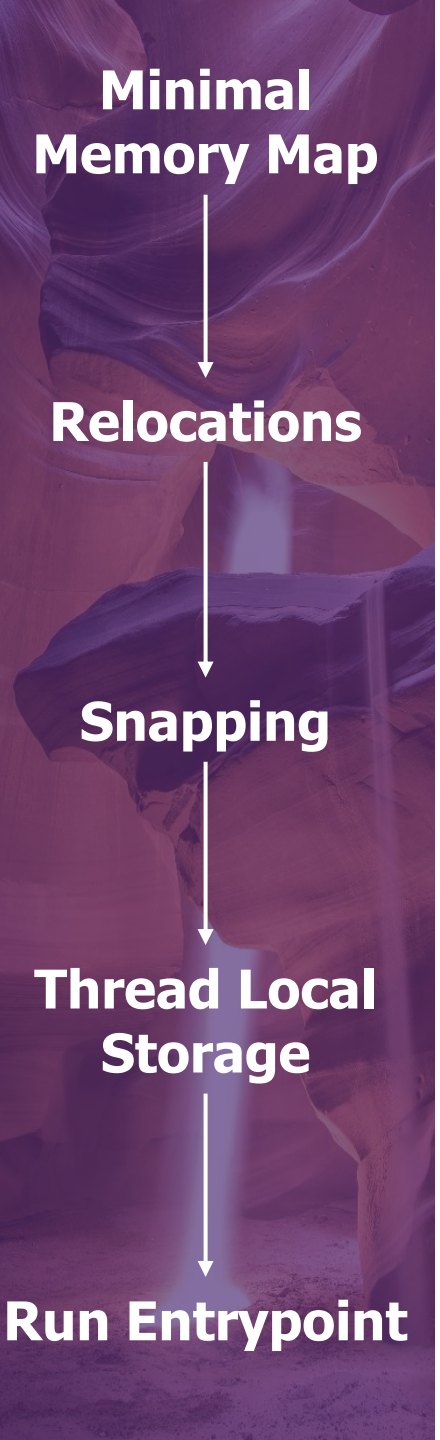
The *TLS* callback **allow the execution of custom code in unitary thread**, for setting up or cleaning up resources. They are specified in the *DLL* header, and can be used for various purposes, including debugging, resource management, and sometimes even malware activities.

Locate the TLS table

- > The *TLS* table is **pointed by the *DataDirectory***
- > Each entry contains the *TLS* callback address that **can be cast into a function and executed**

Register SEH exceptions

- > The *SEH* exceptions **allow handling and recovering of various exceptions**
- > The table address is pointed by the *DataDirectory* and they **can be registered using the *RtlAddFunctionTable*** *WIN32 API.*



This is the end, where all begins
> Executing the DLL entry points

Once mapped, **the *DLL Main* must be run**. All initialization actions will be performed by the *DLL*. If the *DLL* main exit without error, you will have **successfully loaded the *DLL***

Set the protection on the different sections

> Once the *DLL* has been loaded in memory, the **different section protection must be set**
> The protection expected by each section is **compiled in the section headers** under the *Characteristic* field
> Use *VirtualProtect* to set the section protection

Locate the entry point and execute the main function

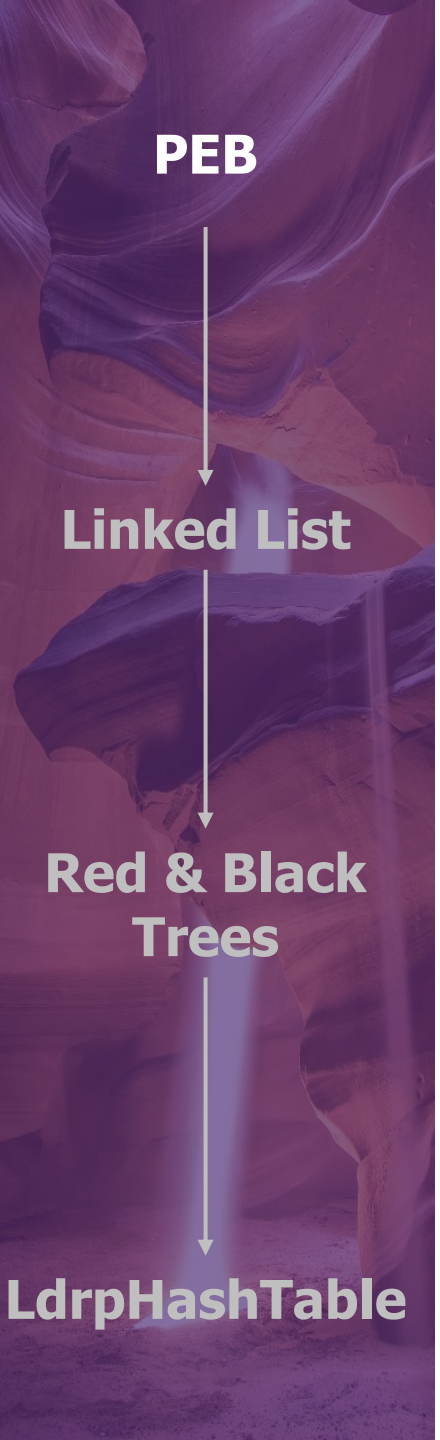
> The *entrypoint* address **is pointed by the *DLL Optional Header***
> It is possible to **cast the address into a function pointer** and call the *DLL entrypoint*

Now, it's your time to play with DLL

/ **02**

WIN32 API Support : play with the PEB

Process environment block
> The bible for your process



The *PEB* (Process Environment Block) is a data structure in the operating system that contains **information about a process,** such as its loaded modules, environment variables, and process startup parameters

Locate the PEB

> The *PEB* is located at **the offset *0x60* of the *TEB*** (*Thread environment block*)

What is the link between PEB and DLL ?

> The *PEB* contains the *LDR_DATA* and *LDR_ENTRY* structure that **contains the information of the different modules** loaded by the process

> There are **three main lists** containing the *DLL* loaded information:

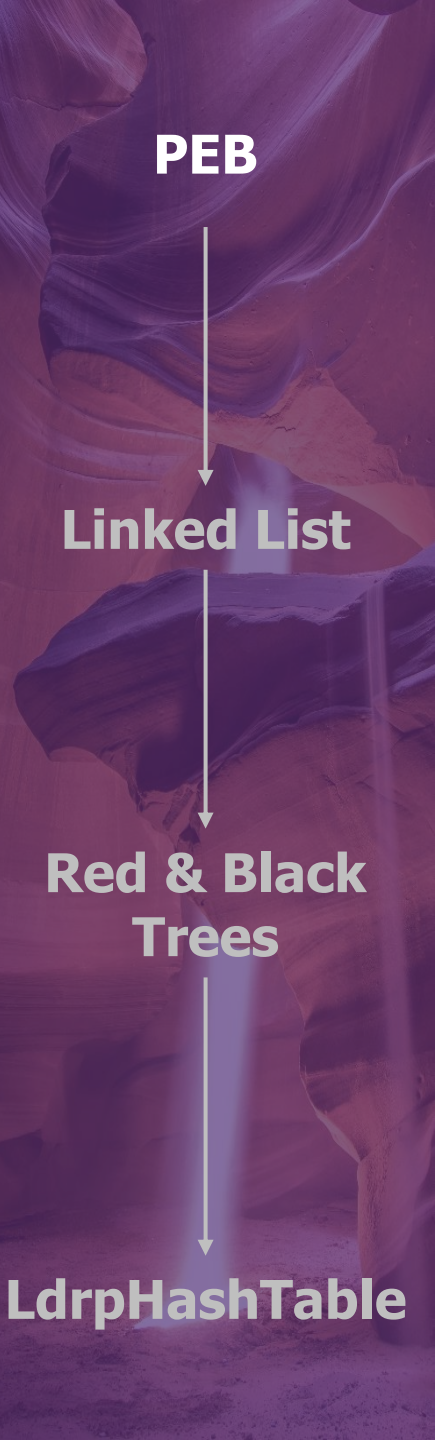
> *InLoadOrderModuleList*

> *InMemoryOrderModuleList*

> *InInitializationOrderModuleList*

```
void get_ldr_entry(){
    PPEB peb = (PPEB)__readgsqword(0x60);
    PLIST_ENTRY hdr = NULL;
    PLIST_ENTRY ent = NULL;
    PLDR_DATA_TABLE_ENTRY ldr = NULL;
    hdr = &(peb->Ldr->InLoadOrderModuleList);
    ent = hdr->Flink;
    for (; hdr != ent; ent = ent->Flink){
        ldr = (void*)ent;
        printf("%ls\n", ldr->BaseDllName.Buffer);
    }
}
```

Process environment block
> The bible for your process

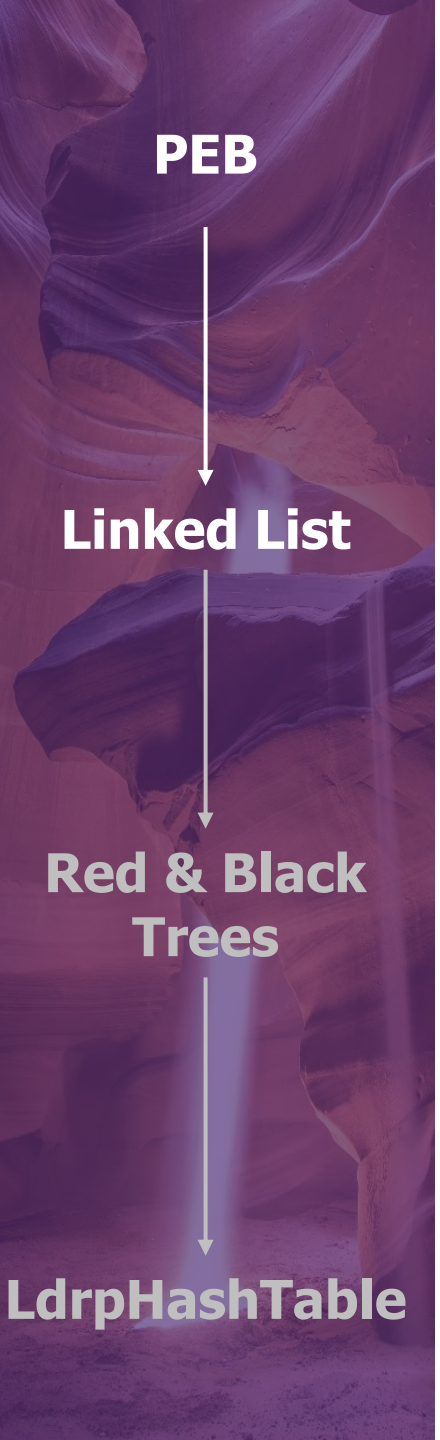


The *PEB* (*Process Environment Block*) is a data structure in the operating system that contains information about a process, such as its loaded modules, environment variables, and process startup parameters

LDR_DATA and LDR_ENTRY

- These structures contain **detailed information about each loaded module**, including its base address, size, entry points, and various other attributes.
- It contains the **main pointer to access to the different internal structures** such as the linked lists or the *Red&Black* tree nodes
  - *LIST_ENTRY InLoadOrderLinks;*
  - *LIST_ENTRY InMemoryOrderLinks;*
  - *LIST_ENTRY InInitializationOrderLinks*
  - *RTL_BALANCED_NODE BaseAddressIndexNode;*
  - *RTL_BALANCED_NODE MappingInfoIndexNode;*

Load the DLL in the Linked Lists
> Make the process aware of the loaded DLL



The *LinkedList* were used in the first *Windows* version to locate the *DLL* loaded by the process. When doing a *GetModuleHandle* on the previous *Windows* version, the function would parse these lists a retrieve the *LDR ENTRY* associated

InLoadOrderModuleList

> This list contains the **modules in the order they were loaded** into the process. It is useful for iterating through modules in the sequence they were initially loaded
> Contains the *DLL LDR_ENTRY* structure

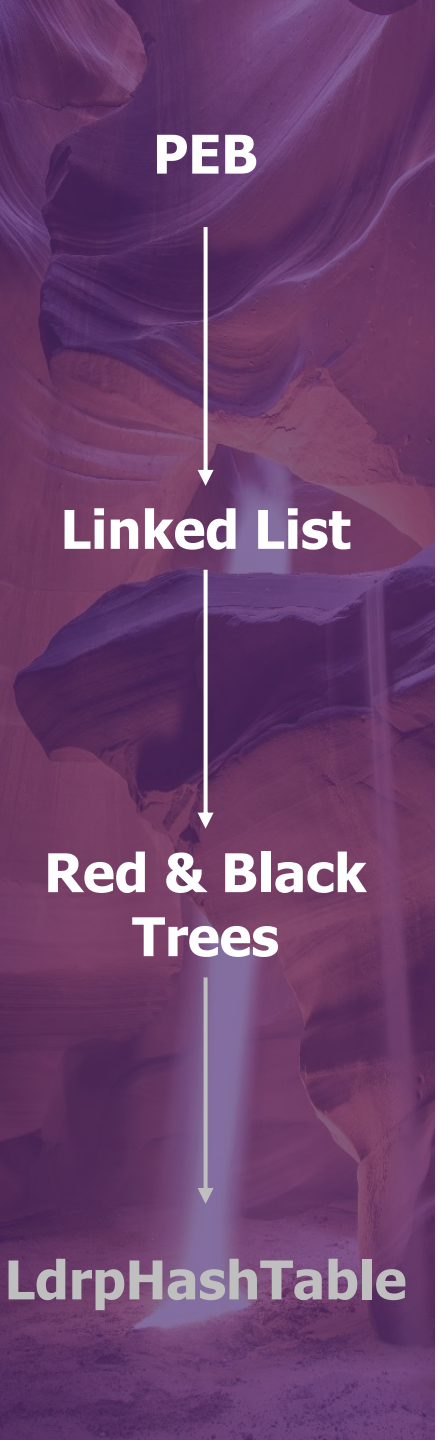
InMemoryOrderModuleList

> This list orders the **modules based on their memory locations**. It helps in scenarios where you need to perform operations based on the memory layout of the modules.
> Contains the *DLL LDR_ENTRY* structure

InInitializationOrderModuleList

> This list orders the **modules based on the sequence of their initialization**. This order is particularly relevant during process startup when the initialization routines (e.g., *DllMain*) are called.
> Contains the *DLL LDR_ENTRY* structure

Load the DLL in the Red & Black binary trees
> New Windows version, new search algorithm



To speed up *DLL* lookup, *Windows* implemented two *Red&Black* binary trees. Today, the implementation of *GetModuleHandle* does not look on the linked list, **but on the *Red&Black* trees**

LdrpModuleBaseAddressIndex (Ldr.BaseAddressIndexNode) :

> This red-black tree **indexes loaded modules by their base address**. It allows for quick lookups of modules based on their starting address in memory. This is useful for operations where the base address is known or when needing to find a module residing at a particular memory location.

LdrpMappingInfoIndex (Ldr.MappingInfoIndexNode)

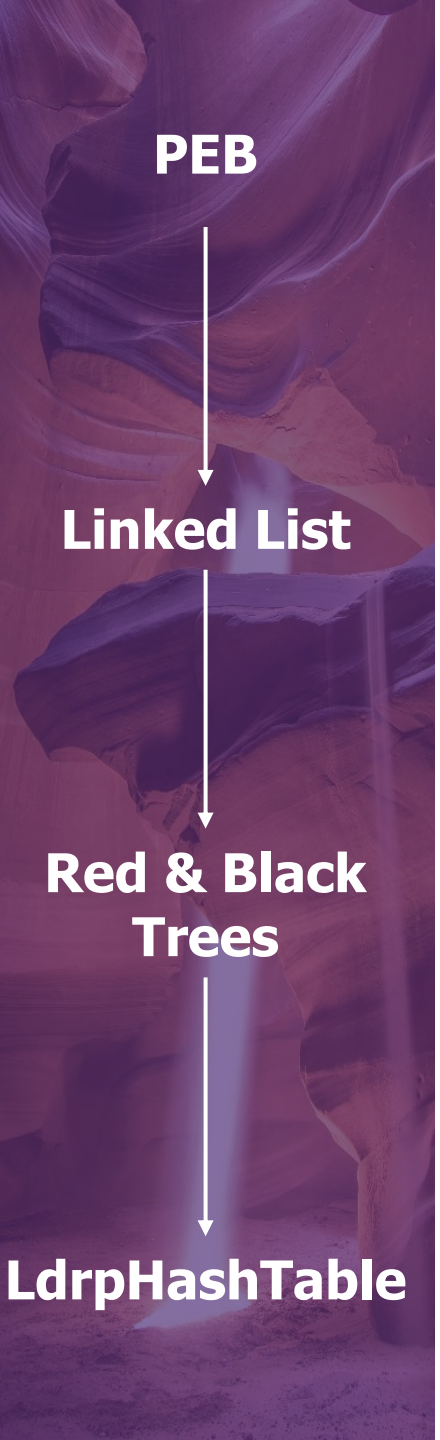
> This red-black tree **indexes loaded modules by their mapping information**, specifically the sections of the module mapped into memory. This index helps in scenarios where the detailed mapping of the module (such as sections and their attributes) is required.

Adding element to these trees

> The addresses of these trees are not exported from the *NTDLL*. It **can be retrieved by getting the *LDR_ENTRY* of *NTDLL***, and rewinding the tree from the node pointed by the *BaseAddressIndexNode* attribute

> The node can be added using the ***RtlRbInsertNodeEx Win32 API***

You may not like it, but this is what peak performance looks like
> Increase lookup performance with LdrpHashTable



LdrpHashTable is a data structure used by the Windows loader to **manage and quickly access loaded modules** (*DLLs*). The hash table allows for efficient lookups of modules based on their names.

Purpose of LdrpHashTable

> Provides a mechanism to **efficiently retrieve information about loaded modules by their names**. This is particularly useful for operations such as checking if a module is already loaded or resolving dependencies between modules.

Structure of LdrpHashTable

> The *LdrpHashTable* consists of an **array of linked lists (or buckets).** Each bucket contains entries that hash to the same value, allowing for collisions to be managed gracefully. Each entry in the bucket is a *LDR_DATA_TABLE_ENTRY* structure, which contains detailed information about a module.

Usage of LdrpHashTable

> When a module is loaded, its **name is hashed**, and the resulting hash value **determines the bucket** in the *LdrpHashTable* where the module's *LDR_DATA_TABLE_ENTRY* is stored through its *Hashlink* address. To find a module, the loader hashes the module's name and searches the corresponding bucket for a matching entry

You may not like it, but this is what peak performance looks like
> Increase lookup performance with LdrpHashTable

PEB

Linked List

Red & Black Trees

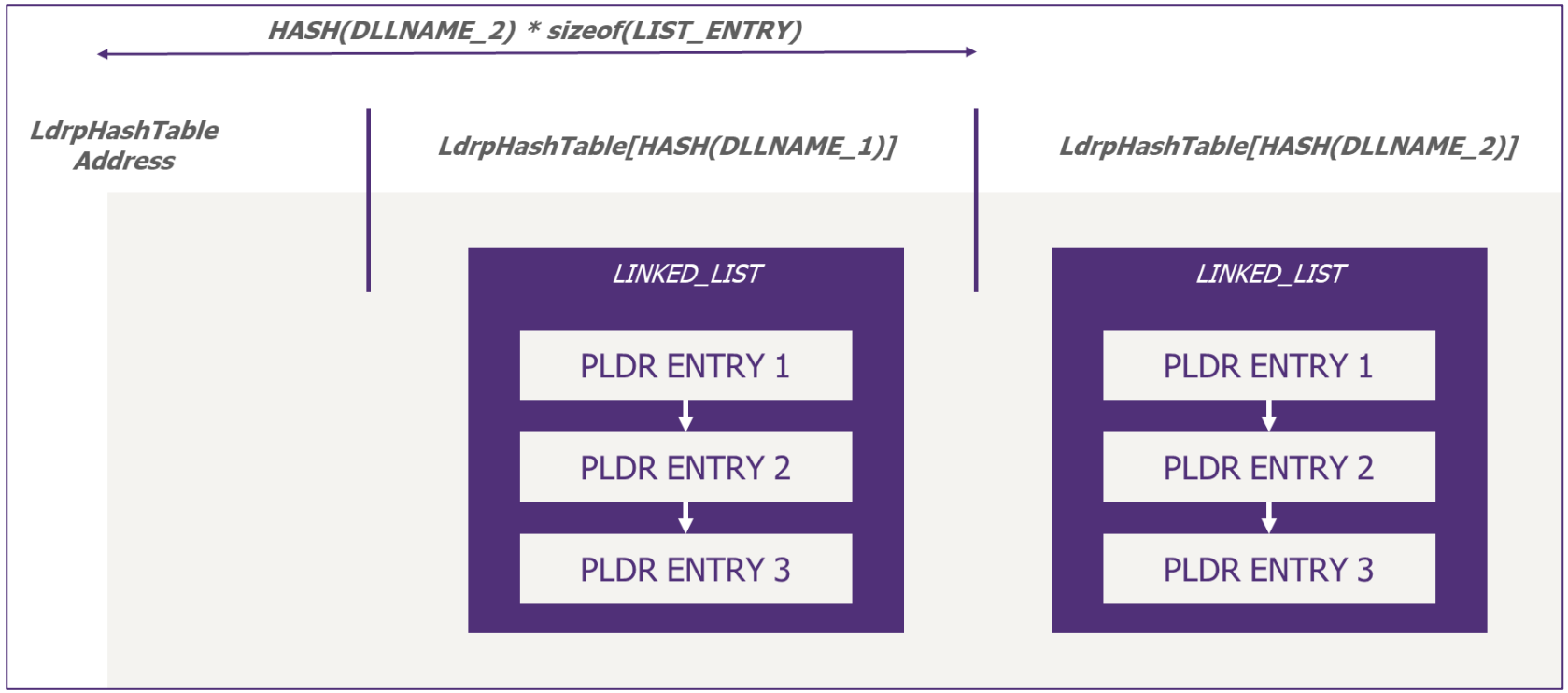
LdrpHashTable

LdrpHashTable is a data structure used by the Windows loader to manage and quickly access loaded modules (*DLLs*). The hash table allows for efficient lookups of modules based on their names.

Finding the LdrpHashTable

> The *LdrpHashTable* is not an exported symbol and **cannot be accessed from outside the *NTDLL***

> Its **address can be recomputed** by getting the *Hashlink* list element and rewinding the list

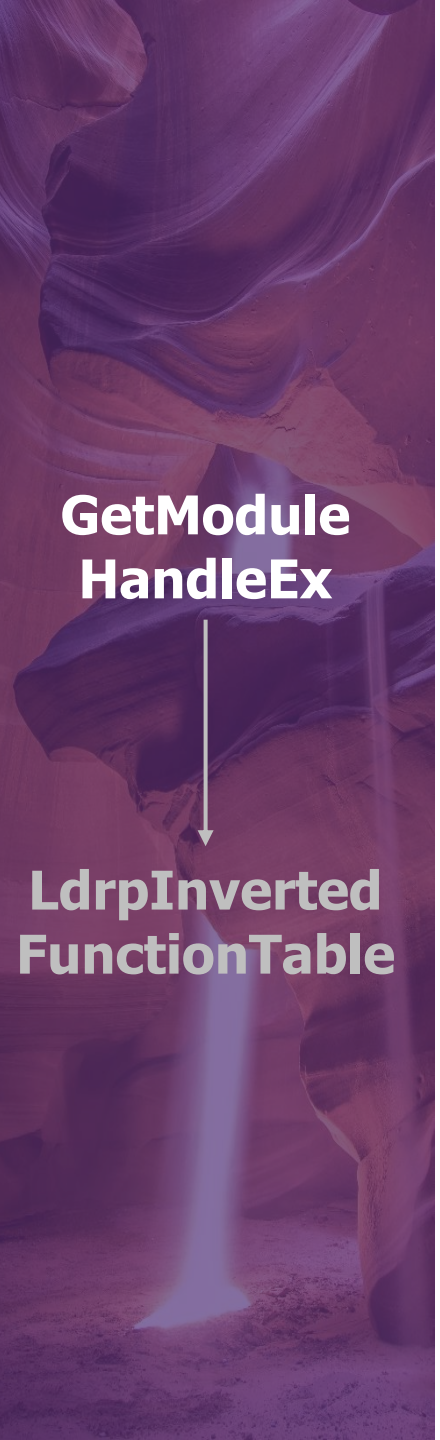


Each bucket does not directly contain the PLDR_ENTRY but the Hashlink address that can be used to make the link with the PLDR ENTRY

/ 03

Limit of PEB Integration

The devil is in the detail
> A forgotten option highlighting a limit



The *GetModuleHandleEx* function **can use several additional options**. Among these options, some are not supported by the loader making the *DLLMain* of some DLL crashing.

For example, with the previous loader, it is not possible to load the *Bcrypt DLL*.

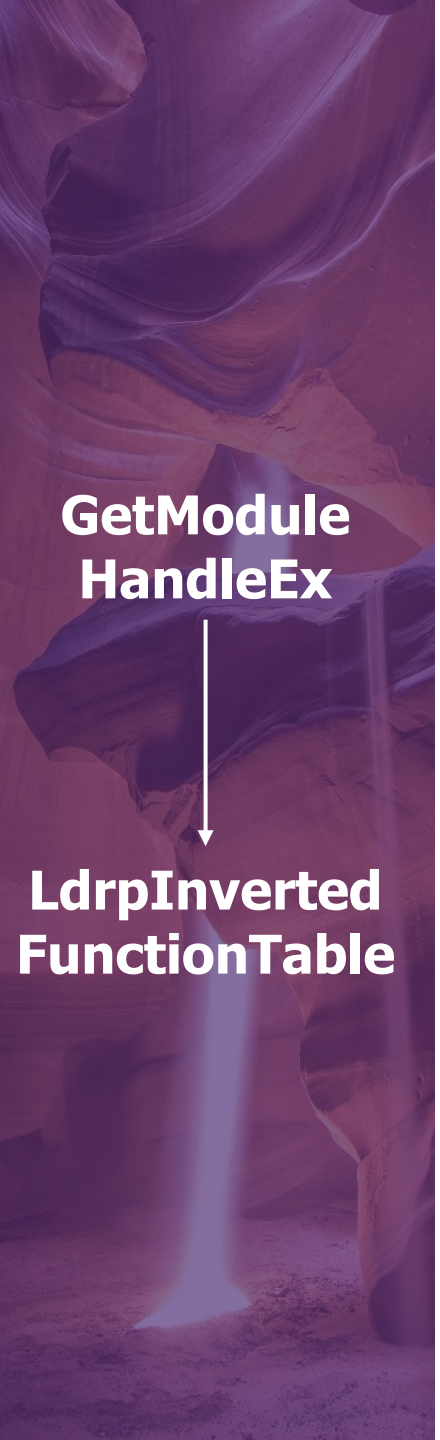
GET_MODULE_HANDLE_EX_FLAG_FROM_ADDRESS

- When this flag is used, the *GetModuleHandleEx* function will search the module not by name but from the address of the function given as parameter
- *GetModuleHandleEx(4, ReadFile)* will return *Kernel32* as *ReadFile* is contained in Kernel32

How does it work ?

- It does not use any of the previous structure
- By reversing the *NTDLL*, it **is possible to retrieve the *RtlpInvertedFunctionTable***
- It uses the **table to retrieve the *DLL* base address**

Discovering a new internal structure
> The use of LdrpInvertedFunctionTable



The *LdrpInvertedFunctionTable* is an *NTDLL* internal structure that is filled during the *DLL* loading. It is then used for quick lookup and use by, at least, the *GetModuleHandleEx* function to retrieve the *DLL* loaded in the process

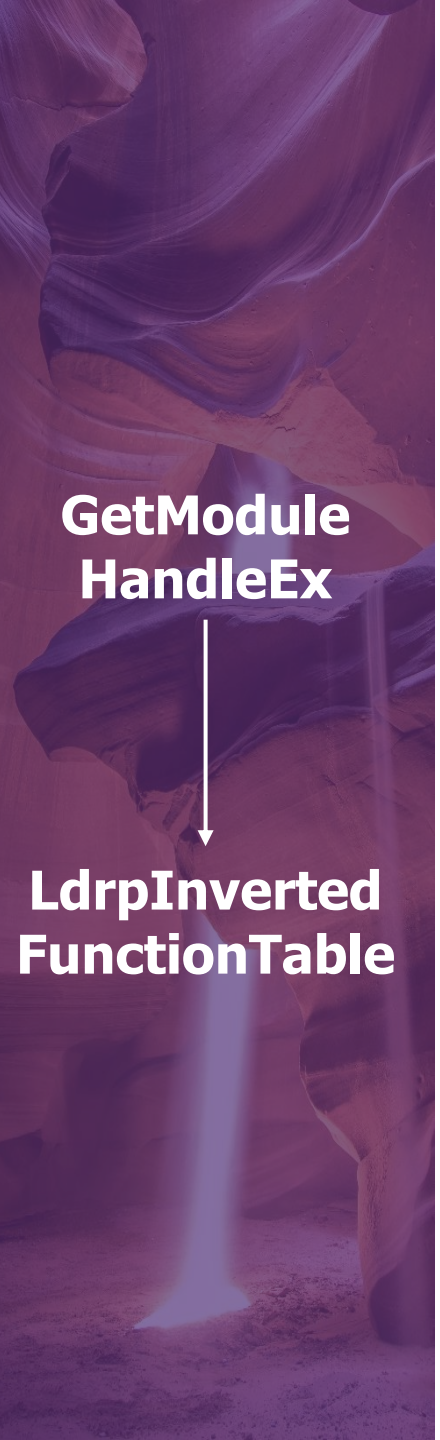
Usage for LdrpInvertedFunctionTable

> It contains the **last *0x200 DLL* loaded by the process** sorted by base address
> It is used for **a quick lookup of *DLL* base address**, size, and exception directory
> It is a **linear array** containing the *DLL* information

Locate the LdrpInvertedFunctionTable

> The table is **located in the *.mrdata*** section of the *NTDLL*
> The structure **contains the field *Count* and *MaxCount*** where *MaxCount* is always lower than *0x200* and *Count* is always more than 0
> The element of the table represents values from *DLL* such as the exception table, the base address and the size. It is possible to check the consistency of this information
> Finally, it is possible to **cross-check the element of the table with the different *DLL* loaded** in the process through the previous *PEB* structures

Discovering a new internal structure
> The use of LdrpInvertedFunctionTable



The *LdrpInvertedFunctionTable* is an *NTDLL* internal structure that is filled during the *DLL* loading. It is then used for quick lookup and use by, at least, the *GetModuleHandleEx* function to retrieve the *DLL* loaded in the process

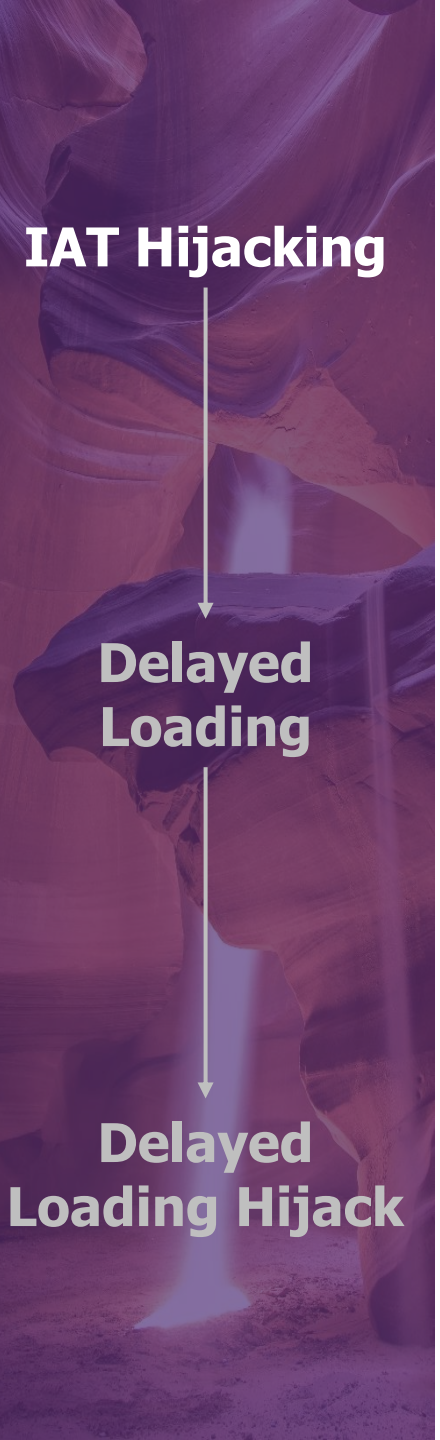
Write into the LdrpInvertedFunctionTable

- The table contains the *RTL_INVERTED_FUNCTION_TABLE_ENTRY* structure
- It is a **static table with *0x200* elements**, the current count is saved in the structure, so it is possible to simply add the element at the right index
- The table is **sorted by base address**, so we might need to move the memory
- We can just **copy the *RtlpInsertInvertedFunctionTableEntry* function from the *NTDLL***

```
typedef struct _RTL_INVERTED_FUNCTION_TABLE_ENTRY
{
    PIMAGE_RUNTIME_FUNCTION_ENTRY ExceptionDirectory;
    PVOID                         ImageBase;
    ULONG                         ImageSize;
    ULONG                         ExceptionDirectorySize;
} RTL_INVERTED_FUNCTION_TABLE_ENTRY, * PRTL_INVERTED_FUNCTION_TABLE_ENTRY;
```

/ **04**

Delayed Loading and IAT



You load so you control
> Manipulating the DLL properties

The *IAT* is the table allowing a *DLL* to resolve external dependencies. This table is filled up during the *Snapping* step of the loading. When doing the loading, **we have to manually resolve the references to the external functions** using *GetProcAddress* and *GetModuleHandle*. What would happen if we **choose to not resolve on the right function** ?

IAT Table

> The *IAT* table is a **simple collection of function address and name**
> Setting a wrong address in it will compel the loaded library to use the wrong function

Impact

> You can **replace any external function used by the *DLL* by the function you want**
> It can be used, for example, to **replace the *LoadLibraryA*** by our own implementation or to replace some functions used to raise events (**such as the *AMSI Scanbuffer***) by a decoy, blinding the security tools based on these specific functions

We are loading all DLL at once…
> Understanding the Delayed Loading functionning

IAT Hijacking

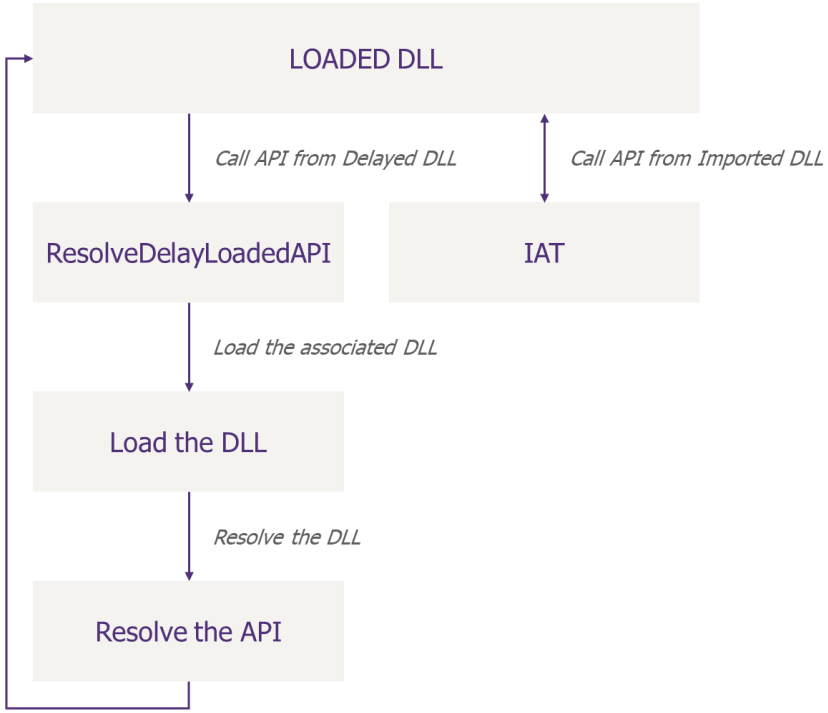
Delayed Loading

Delayed Loading Hijack

When we are loading our *DLL* by ourselves, we are loading the mandatory *DLL* as well as all the delayed DLL that will likely never be used. This can be an *IOC* cause we are loading way more *DLL* in the process that what is needed

How work the Delayed Loading

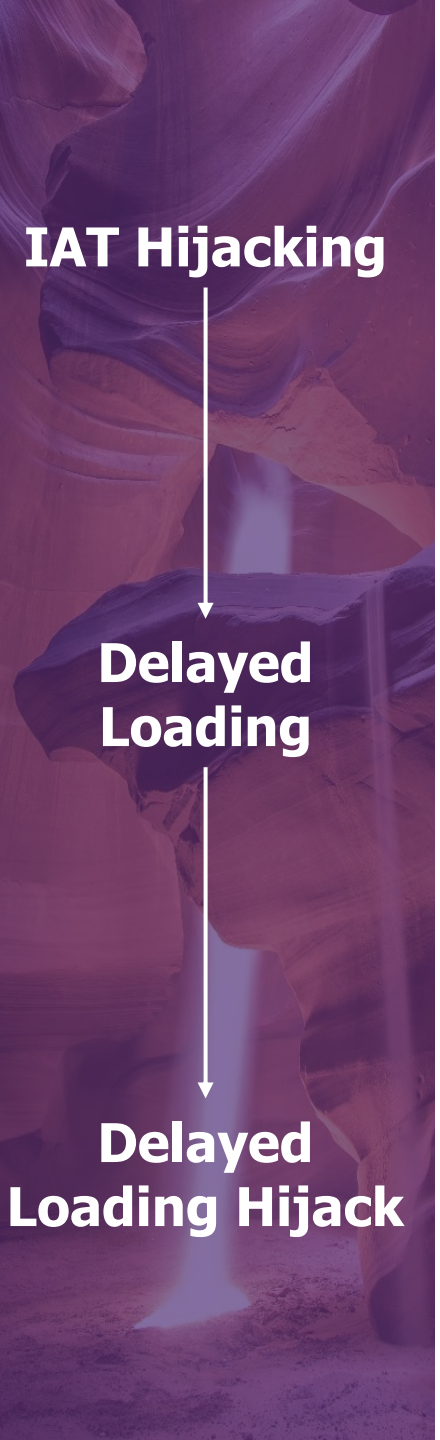
> When a process try to access to a function contained in a delayed *DLL*, it will **not directly look at the *IAT***, but it **will call the *ResolveDelayLoadedAPI***.
> This function will **load to *DLL* just in time** and resolve the *API* address



What if I don't load the delayed DLL

> If you don't load the delayed *DLL*, the process will do it by itself using the *ResolveDelayedLoadedAPI* and will raise an *ImageLoad* event

... But we have control over everything
> Use IAT hijacking on ResolveDelayedLoadedAPI



The real problem with *ResolveDelayedLoadedAPI* is that it **will likely not use our custom loader to load the delayed *DLL*** which will result in a *LoadImage* kernel callback and we will lose any control on the loading of the child *DLL*. For example, if we implement some ***IAT* hijacking on a *DLL* and it loads a child *DLL* through delayed loading**, the *IAT* hijacking will not be applied on this child *DLL*.

Hijack the ResolveDelayedLoadedAPI

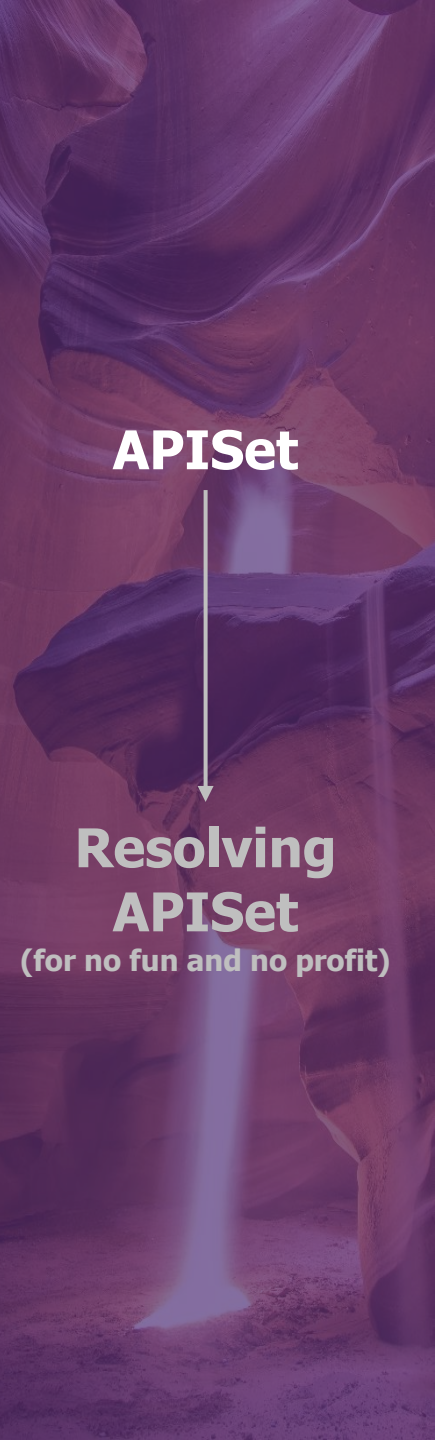
> The idea is to **perform an *IAT* hijacking when loading the *DLL***
> We **hijack the *ResolveDelayedLoadedAPI*** to reroute the execution flow to a controlled reimplementation of the function

Implement the IAT Hijacking

> The rogue function will just load the *DLL* needed and **forward the call to the real *ResolveDelayedLoadedAPI***
> The **name of the DLL to call can be computed** from the initial *ResolveDelayedLoadedAPI* arguments
> This function is then written in the *IAT* during the *DLL* loading snapping step

/ **05**

APISet : a DLL that is not a DLL



A DLL that is not a DLL
> A DLL indexer

Several *DLL* have a specific name starting with *api-* or *ext-*. These files are called *APISet* and are not *DLL*. They can be loaded as it is using *LoadLibrary* but they are not under *PE* format.

What are APISet

> To **ease portability among devices** (*PC*, *Xbox*, *smartphone*) *Windows* introduced the *APISet*
> The goal is to ensure that a **specific set of features will be loaded on each platform** even if they are in fact stored in different *DLL*
> Whatever the platform you are developing to, loading a specific *APISet* will ensure that a specific set of features is loaded

Impact

> The *APISet* are **glorified cross-reference table** that resolves to a specific *DLL* such as *KERNEL32*
> When an *APISet* is loaded, it must be first resolved to its DLL, and then, the target *DLL* can be loaded as usual

Dive into APISet implementation
> From the ApiSchema to the PEB

The *APISet* mapping is loaded into the process's memory at startup and can be used to resolve them

APISet

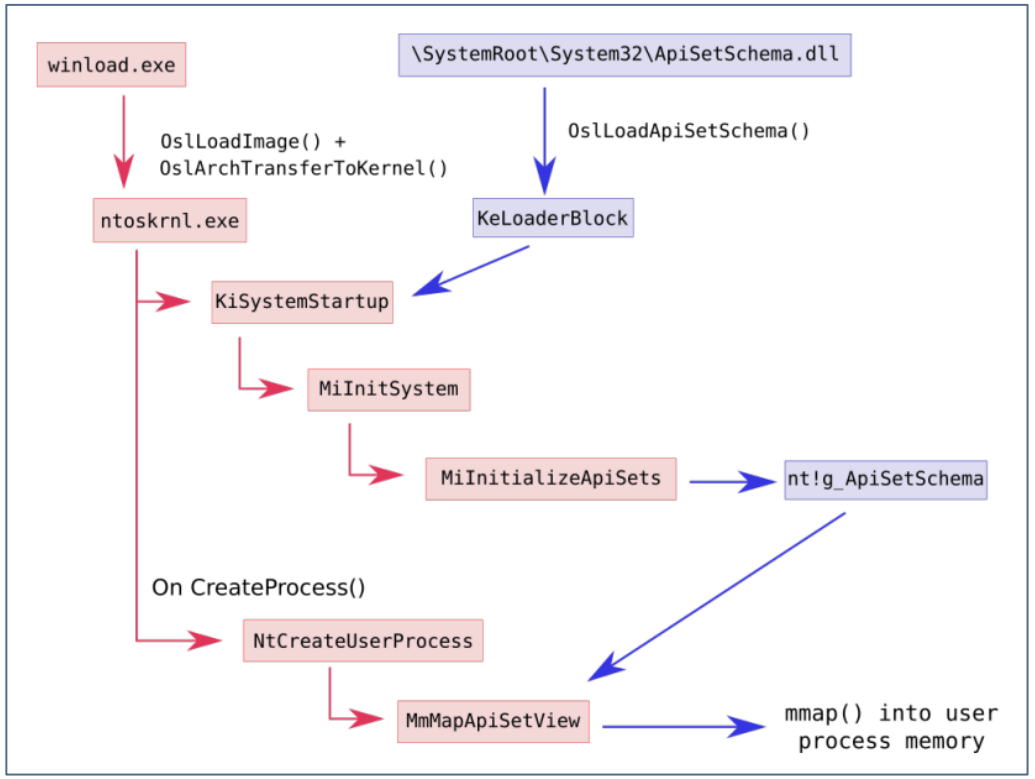
Resolving APISet
(for no fun and no profit)

Process creation

> The ***API* schema is stored** in the *C:|Windows|System32|ApiSetSchema.dll* **into the *.apiset* section.**

> During the Windows boot, the section is stored into a *KERNEL* structure and then **copied into the static variable *g_ApiSetSchema*** on kernel startups.

> When a process is created, the *MmMapApiSetView* **copy the *APISet* into the process's memory** and the address is **stored in the *PEB.ApiSetMap***



Stolen from DLL shell game and other misdirections article

Resolve the APISet
> Use the Win32 API to resolved the WIN32 nonsense

APISet

↓

Resolving APISet
(for no fun and no profit)

The *APISet* mapping is a hashtable of the correspondence between the *APISet* name and the host *DLL*. Internally, 4 functions are used to resolve an *APISet*

ApiSetQueryApiSetPresence

> High level API used to check if the *DLL* is associated with an *APISet*

ApiSetResolveToHost

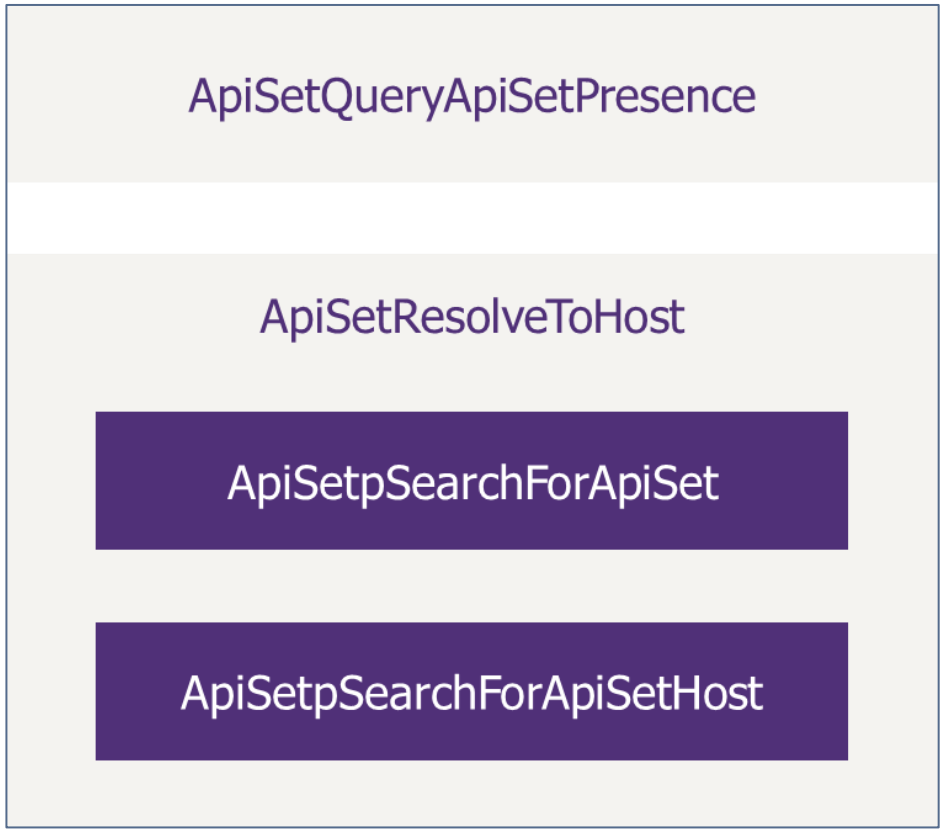
> High level *API* used to retrieve the host *DLL* associated with the *APISet*

ApiSetpSearchForApiSet

> Perform the lookup in the *APISetMap* hash table

ApiSetpSearchForApiSetHost

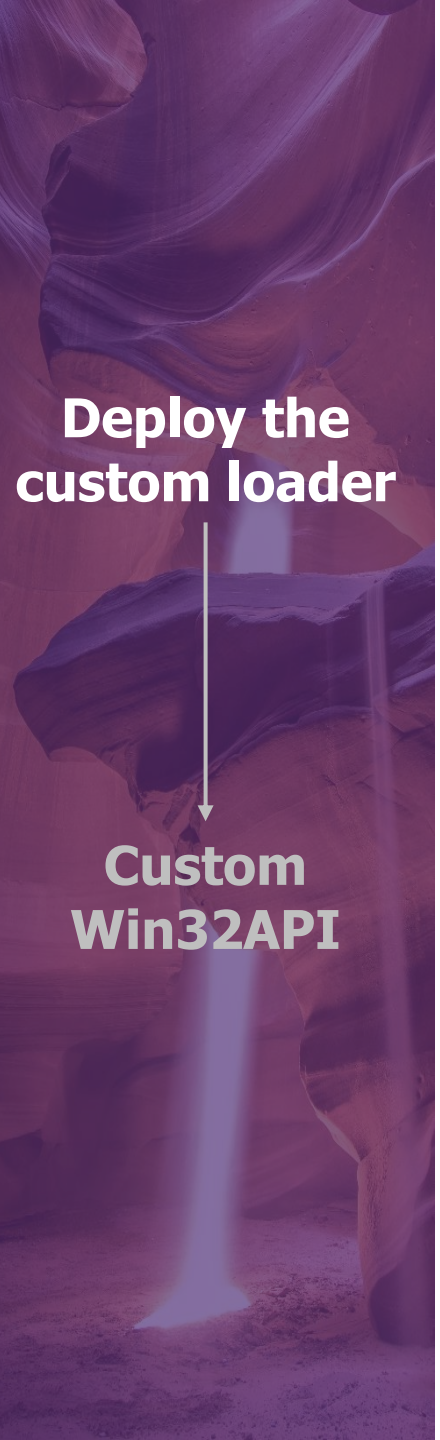
> In the case an *APISet* points to different hosts DLL, this function will handle the right *DLL* choice. I personally never seen this one in use



/ 06

WinHTTP

Going back to IAT Hijacking
> Deploy your loader on the whole loading chain



The loader is now fully functional but it is limited to the DLL we are loading ourselves. All the dependencies will be loaded using the *LoadLibrary* function

Go back to IAT Hijacking

> We can use the IAT hijacking to replace all *LoadLibrary** with our custom loader

> We have to reimplement all wrapper functions to handle the *Unicode* and the *Ex*

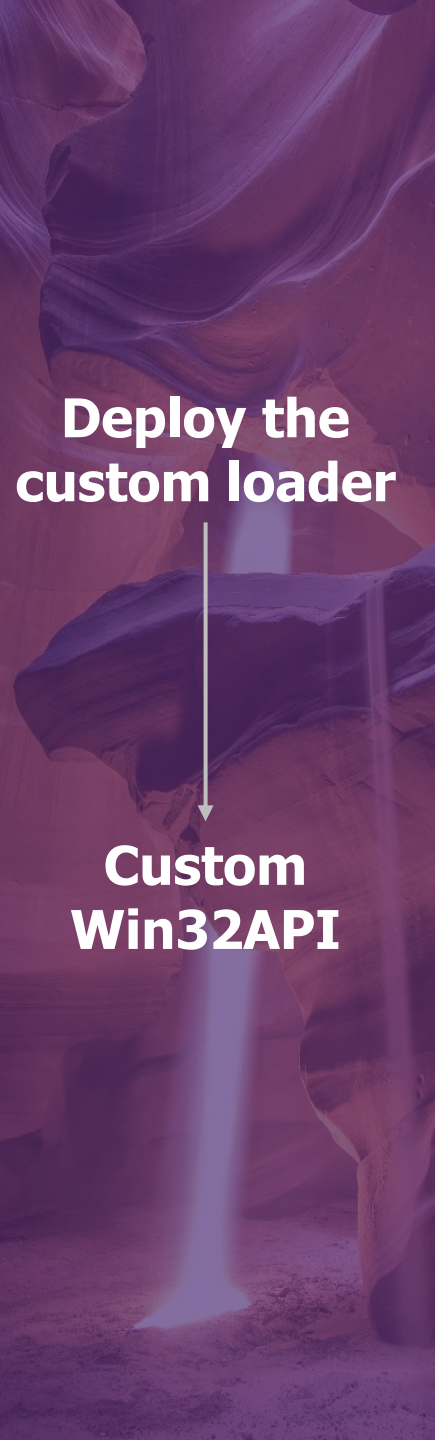
Replace all LoadLibrary in the code

> Remove all references to *LoadLibrary* by our specific loader

Implement the DLL path search order

> We have to implement a simple *DLL* name resolver to handle the different path name

Going back to IAT Hijacking
> Deploy your loader on the whole loading chain



The loader is now fully functional but it is limited to the DLL we are loading ourselves. All the dependencies will be loaded using the LoadLibrary function

GetModuleHandle

> Use the *PEB InLoadOrderModuleList* to locate the *DLL* and retrieve their base address (*HMODULE*)
> Iterate over the different modules and return the *BaseAddress*

GetProcAddress

> Retrieve the *exportDirectory* of the *DLL* and iterate over the functions
> Check for a specific forwarder, and follow the forwarder
> Handle ordinal lookup

DLL Forwarder

> Some *DLL* reference functions but does not implement them. This is mainly for portability when a function has been moved to another *DLL*
> The forwarder chain can be used to retrieve the *DLL* implementing the function


"That's all Folks!"

PARIS

LONDRES

NEW YORK

HONG KONG

SINGAPOUR *

DUBAI *

SAO PAULO *

LUXEMBOURG

MADRID *

MILAN *

BRUXELLES

GENEVE

CASABLANCA

ISTANBUL *

LYON

MARSEILLE

NANTES

* Partenariats