

Windows Hello World

Thomas Bitzberger
bitz@lse.epita.fr

But why ?

- Realized I actually knew nothing about Windows internals
- Every executable including the kernel is a PE
- Understanding the binary format is important, whatever OS
- Let's begin by that !

What are we doing

- Let's begin with a very simple LSE exercise : calling 'printf()'
- But let's not do it the ez way
- We don't want to use the 'printf' symbol
- We want to find it's address at runtime and jump there

Where do we start

Let's think about what a dynamic linker basically does:

- Loading shared libraries
- Making relocations

To do it's work, it maintains a state of the loaded binaries:

→ The Link Map

On Windows, the dynamic linking is done by the binary loader.

How to find the link map

We can find it using a pretty useful structure:

→ *Thread Information Block*

- Per-thread structure
- Accessed via %fs/%gs in 32/64 bits + an offset
- Contains a lot of informations:
 - *LastError*
 - [... lots of stuff ...]
 - PEB address (offset 0x60)

Process Environment Block

```
typedef struct _PEB {
    BYTE Reserved1[2];
    BYTE BeingDebugged; /* :) */
    BYTE Reserved2[1];
    PVOID Reserved3[2];
    PPEB_LDR_DATA Ldr; /* link map */
    PRTL_USER_PROCESS_PARAMETERS ProcessParameters;
    BYTE Reserved4[104];
    PVOID Reserved5[52];
    PPS_POST_PROCESS_INIT_ROUTINE PostProcessInitRoutine;
    BYTE Reserved6[128];
    PVOID Reserved7[1];
    ULONG SessionId;
} PEB, *PPEB;
```

Microsoft beauty

As you may have noticed, we don't have a lot of infos about structures exported fields.

This is the case for the PEB and many other structures we'll need.

Fortunately, Microsoft gives us a way to access the real structures definitions ! (That's not MSDN).

Debugger to the rescue

Using *WinDbg* that is included in Windows SDK, we can get the informations we want.

We can configure *WinDbg* to use Microsoft symbol server.

Then, using the '*dt*' (display type) we get the real structure definition !

Using WinDbg

```
0:007> dt -t _PEB
ntdll!_PEB
+0x000 InheritedAddressSpace : UChar
+0x001 ReadImageFileExecOptions : UChar
+0x002 BeingDebugged : UChar
+0x003 BitField : UChar
+0x003 ImageUsesLargePages : Pos 0, 1 Bit
+0x003 IsProtectedProcess : Pos 1, 1 Bit
+0x003 IsImageDynamicallyRelocated : Pos 2, 1 Bit
+0x003 SkipPatchingUser32Forwarders : Pos 3, 1 Bit
+0x003 IsPackagedProcess : Pos 4, 1 Bit
+0x003 IsAppContainer : Pos 5, 1 Bit
+0x003 IsProtectedProcessLight : Pos 6, 1 Bit
+0x003 IsLongPathAwareProcess : Pos 7, 1 Bit
+0x004 Padding0 : [4] UChar
+0x008 Mutant : Ptr64 Void
+0x010 ImageBaseAddress : Ptr64 Void
+0x018 Ldr : Ptr64 _PEB_LDR_DATA
+0x020 ProcessParameters : Ptr64 _RTL_USER_PROCESS_PARAMETERS
+0x028 SubSystemData : Ptr64 Void
+0x030 ProcessHeap : Ptr64 Void
```

Real structure size is almost 2 Ko !

Process Environment Block

```
void *GetPEB(void)
{
    return (void *)__readgsqword(0x60);
}
```

Microsoft compiler has no support for 64 bits inline assembly. Yay...

We can use a set of compiler builtins that will do the job.

This is sometimes not enough (no *lgdt* builtin for example).

PEB_LDR_DATA

```
0:007> dt -t _PEB_LDR_DATA
ntdll!_PEB_LDR_DATA
+0x000 Length : Uint4B
+0x004 Initialized : UChar
+0x008 SsHandle : Ptr64 Void
+0x010 InLoadOrderModuleList : _LIST_ENTRY
+0x020 InMemoryOrderModuleList : _LIST_ENTRY
+0x030 InInitializationOrderModuleList : _LIST_ENTRY
+0x040 EntryInProgress : Ptr64 Void
+0x048 ShutdownInProgress : UChar
+0x050 ShutdownThreadId : Ptr64 Void
```

Ok so we have lists... but where are the loading infos ?

Windows is definitely intrusive

```
typedef struct _LIST_ENTRY {  
    struct _LIST_ENTRY *Flink; /* next */  
    struct _LIST_ENTRY *Blink; /* prev */  
} LIST_ENTRY, *PLIST_ENTRY;
```

The list is also circular. This kind of list is also used in the Linux Kernel.

Getting loaded DLLs infos

Reading the Windows documentation about PEB_LDR_DATA:

“InMemoryOrderModuleList

The head of a doubly-linked list that contains the loaded modules for the process. Each item in the list is a pointer to an **LDR_DATA_TABLE_ENTRY** structure.”

LDR_DATA_TABLE_ENTRY

```
0:007> dt -t _LDR_DATA_TABLE_ENTRY
ntdll!_LDR_DATA_TABLE_ENTRY
+0x000 InLoadOrderLinks : _LIST_ENTRY
+0x010 InMemoryOrderLinks : _LIST_ENTRY
+0x020 InInitializationOrderLinks : _LIST_ENTRY
+0x030 DllBase           : Ptr64 Void
+0x038 EntryPoint       : Ptr64 Void
+0x040 SizeOfImage      : Uint4B
+0x048 FullDllName      : _UNICODE_STRING
+0x058 BaseDllName      : _UNICODE_STRING
```

Using this we can access the loaded DLLs !

Getting the entries

- 1) Get the head of the loaded DLLs list (in PEB_LDR_DATA)
- 2) Iterate on list
 - a) You can retrieve the entry using classic 'CONTAINER_OF' macro

Ok so now we can access the DLLs, let's see how to lookup functions !

Let's look up the PE header format ...

PE File format

offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x00000000	0x5A4D (MZ)		lastsize		PagesInFile		relocations		headerSizeInParagraph		MinExtraParagraphNeeded		MaxExtraParagraphNeeded		Initial (relative) SS	
0x00000010	Initial (relative) SP		checksum		Initial IP		Initial (relative) CS		FileAddOfRelocTable		OverLayNumber		reserved		reserved	
0x00000020	reserved		reserved		OEMIdentifier		OEMInformation		reserved		reserved		reserved		reserved	
0x00000030	reserved		reserved		reserved		reserved		reserved		reserved		0x80 (offset to PE signature)			
0x00000040	This block contains instructions to display the message "This program cannot be run in DOS mode" when run in MS-DOS															
0x00000050																
0x00000060																
0x00000070																
0x00000080																
0x00000090	NumberOfSymbols (0 for image)				SizeOfOptionalHeaders		Characteristics		0x10B (exe)		lnMajVer	lnMnrVer	SizeOfCode			
0x000000A0	SizeOfInitializedData				SizeOfUninitializedData				AddressOfEntryPoint				BaseOfCode			
0x000000B0	BaseOfData				ImageBase				SectionAlignment				FileAlignment			
0x000000C0	MajorOSVersion		MinorOSVersion		MajorImageVersion		MinorImageVersion		MajorSubsystemVersion		MinorSubsystemVersion		Win32VersionValue			
0x000000D0	SizeOfImage				SizeOfHeaders				Checksum				Checksum		DllCharacteristics	
0x000000E0	SizeOfStackReserve				SizeOfStackCommit				SizeOfHeapReserve				SizeOfHeapCommit			
0x000000F0	LoaderFlags				NumberOfRVAandSizes				.edata offset				.edata size			
0x00000100	.idata offset				.idata size				.rsrc offset				.rsrc size			
0x00000110	.pdata offset				.pdata size				attribute certificate offset (image)				attribute certificate size (image)			
0x00000120	.reloc offset (image)				.reloc size (image)				.debug offset				.debug size			
0x00000130	Architecture (reserved - 0x0)				Architecture (reserved - 0x0)				Global Ptr offset				must be 0x0			
0x00000140	.tls offset				.tls size				Load config table offset (image)				Load Config table size (image)			
0x00000150	Bound import table offset				Bound import table size				IAT (Import address table) offset				IAT (Import address table) size			
0x00000160	Delay import descriptor offset (image)				Delay import descriptor size (image)				CLR runtime header offset (object)				CLR runtime header size (object)			
0x00000170	Reserved (must be 0x0)				Reserved (must be 0x0)				Section header - Name							
0x00000180	VirtualSize				VirtualAddress				SizeOfRawData				PointerToRawData			
0x00000190	PointerToRelocations				PointerToLineNumbers				NumberOfRelocations		NumberOfLineNumbers		Characteristics			
0x000001A0	Section header - Name								VirtualSize				VirtualAddress			
0x000001B0	SizeOfRawData				PointerToRawData				PointerToRelocations				PointerToLineNumbers			
0x000001C0	NumberOfRelocations		NumberOfLineNumbers		Characteristics				Section header - Name..							

			Size in bytes
MS-DOS header			64
PE Signature			4
COFF header			20
Standard fields	Optional header	File header	28
Windows-Specific fields			68
Data directories			variable
Section table (each section header is 40 bytes)			variable

DOS Header

```
0:007> dt -t _IMAGE_DOS_HEADER
ntdll!_IMAGE_DOS_HEADER
+0x000 e_magic           : Uint2B
+0x002 e_cblp           : Uint2B
+0x004 e_cp             : Uint2B
+0x006 e_cr1c          : Uint2B
+0x008 e_cparhdr       : Uint2B
+0x00a e_minalloc      : Uint2B
+0x00c e_maxalloc      : Uint2B
+0x00e e_ss            : Uint2B
+0x010 e_sp            : Uint2B
+0x012 e_csum          : Uint2B
+0x014 e_ip            : Uint2B
+0x016 e_cs            : Uint2B
+0x018 e_lfarlc        : Uint2B
+0x01a e_ovno          : Uint2B
+0x01c e_res           : [4] Uint2B
+0x024 e_oemid         : Uint2B
+0x026 e_oeminfo       : Uint2B
+0x028 e_res2         : [10] Uint2B
+0x03c e_lfanew        : Int4B
```

PE Header

```
typedef struct _IMAGE_NT_HEADERS64 {  
    DWORD Signature;  
    IMAGE_FILE_HEADER FileHeader;  
    IMAGE_OPTIONAL_HEADER64 OptionalHeader;  
} IMAGE_NT_HEADERS64, *PIMAGE_NT_HEADERS64;
```

OptionalHeader will lead us to exported functions.

Optional Header

```
typedef struct _IMAGE_OPTIONAL_HEADER {  
    WORD                Magic;  
    BYTE                MajorLinkerVersion;  
    BYTE                MinorLinkerVersion;  
    DWORD               SizeOfCode;  
    /* ... */  
    IMAGE_DATA_DIRECTORY DataDirectory[MAX_ENTRIES];  
} IMAGE_OPTIONAL_HEADER, *PIMAGE_OPTIONAL_HEADER;
```

MSDN tells us that the export table is the first DataDirectory entry.

Getting export table

```
PVOID LookupDll(PVOID DllBase, PCHAR func_name)
{
    PIMAGE_DOS_HEADER dosHdr = (PIMAGE_DOS_HEADER)DllBase;
    PIMAGE_NT_HEADERS64 peHdr = ((PCHAR)DllBase + dosHdr->e_lfanew);
    ULONG exportAddr = peHdr->OptionalHeader.DataDirectory[0].VirtualAddress;
    PIMAGE_EXPORT_DIRECTORY exportTable = ((PCHAR)DllBase + exportAddr);

    /* ... */
}
```

Retrieving function address

```
PULONG names = (PCHAR)Dllbase + exportTable->AddressOfNames;  
PULONG funcs = (PCHAR)Dllbase + exportTable->AddressOfFunctions;  
for (ULONG i = 0; i < exportTable->NumberOfNames; ++i) {  
    PCHAR name = (PCHAR)DllBase + names[i];  
    if (!Strcmp(name, func_name))  
        return (PCHAR)DllBase + funcs[i];  
}
```

Calling printf

We have everything we need to call 'printf' so let's go !

Let's just loop over loaded DLLs and look for 'printf'.

So I wrote the program, tested and ...

'printf' is not found on any loaded DLL !

I find *puts*, *fputs*, *fwrite*, *__stdio_common_vprintf* and tons of others ...

BUT I WANT { 'P', 'R', 'I', 'N', 'T', 'F' } !!!

Where is printf

So I started investigating ... and found a Microsoft Blog article:

“The Great C Runtime (CRT) Refactoring”

Windows CRT has evolved during time.

It was ‘MSVCRT.DLL’ for a long time.

Then it moved to ‘MSVCR*.DLL’ (one per MSVC version)

Starting for version 14.0 there is now ‘UCRTBASE.DLL’ + another one still depending on MSVC version.

Looking for printf

'printf' is defined in ucrt/stdio.h:

```
'_CRT_STDIO_INLINE __CRTDECL printf('
```

On another Microsoft Blog article about UCRT:

“The **printf** and **scanf** functions are now defined inline”

Nice ...

Let's cheat

'The msvcrt.dll is now a system component owned and built by Windows.'

from [Microsoft documentation](#)

This CRT exports the 'printf' function.

However, this library isn't loaded in our address space.

At this point, this is not a problem :)

Let's cheat

So in order to do complete our mission, we need to load 'MSVCRT.DLL' in memory.

We have a function lookup mechanism in any loaded DLL ...

Let's retrieve 'LoadLibrary' address in KERNEL32.DLL (loaded in every process).

As we're here, let's also get 'GetProcAddress' ...

Eventually winning

```
PVOID l = LookupDll(entry->DllBase, "LoadLibraryA");
```

```
PVOID g = LookupDll(entry->DllBase, "GetProcAddress");
```

```
PHANDLE h = ((load_library_t)l)("msvcrt.dll");
```

```
printf_t print = (printf_t)((get_proc_addr_t)g)(h, "printf");
```

```
print("Windows Hello World - LSE\n");
```

```
C:\Users\zionlion\printf_lse_lt> dumpbin /IMPORTS /SYMBOLS printf.exe |findstr printf
Dump of file printf.exe
```

```
C:\Users\zionlion\printf_lse_lt>
```

```
C:\Users\zionlion\printf_lse_lt>printf.exe
ZionLion - Windows Hello World - LSE
```

Questions ?

Thank you !