

OFFENSIVE SECURITY WRITE-UP

Retrieving Native API addresses and Syscall Ids at runtime

Code:- <https://github.com/prakashyadav008/Offensive-Security>

Overview:- Native api address serve its importance in the implementation of shellcode. The syscall Ids and api addresses can also be used in implementing payloads to hide from EDR detection. The writeup demonstrates how PEB and PE file structure can be used to dynamically retrieve address of API and Syscall Ids at runtime.

Retrieving Base Address Of Ntdll

The first address to retrieve is the pointer to TEB(Thread environment block). TEB structure stores various values relating to the state of a process and also contains the address PEB(Process environment block) structure. The TEB is retrieved using gs:[30] in 64 bit systems. The address of PEB is located at offset 0x60.

```
//0x1000 bytes (0x1000)
struct _TEB64
{
    struct _NT_TIB64 NtTib; //0x0
    ULONGLONG EnvironmentPointer; //0x38
    struct _CLIENT_ID64 ClientId; //0x40
    ULONGLONG ActiveRpcHandle; //0x50
    ULONGLONG ThreadLocalStoragePointer; //0x58
    ULONGLONG ProcessEnvironmentBlock; //0x60
    ULONG LastErrorValue; //0x68
    ULONG CountOfOwnedCriticalSections; //0x6c
    ULONGLONG CsrClientThread; //0x70
    ULONGLONG Win32ThreadInfo; //0x78
};
```

Image: TEB structure

Inside PEB the address of _PEB_LDR_DATA structure is located at offset 0x18.

```
};
};
UCHAR Padding0[4]; //0x4
ULONGLONG Mutant; //0x8
ULONGLONG ImageBaseAddress; //0x10
ULONGLONG Ldr; //0x18
ULONGLONG ProcessParameters; //0x20
ULONGLONG SubSystemData; //0x28
ULONGLONG ProcessHeap; //0x30
```

Image: PEB structure

Inside _PEB_LDR_DATA structure, the InLoadOrderModuleList field is located at offset 0x10. This field points to linked list of a structure of type _LDR_DATA_TABLE_ENTRY.

```

struct _PEB_LDR_DATA
{
    ULONG Length; //0x0
    UCHAR Initialized; //0x4
    VOID* SsHandle; //0x8
    struct _LIST_ENTRY InLoadOrderModuleList; //0x10
    struct _LIST_ENTRY InMemoryOrderModuleList; //0x20
    struct _LIST_ENTRY InInitializationOrderModuleList; //0x30
}

```

Image: _PEB_LDR_DATA structure

At offset 0x60 inside _LDR_DATA_TABLE_ENTRY structure the FullDllName structure stores the name of the dll in wide character format.

```

{
    struct _LIST_ENTRY InLoadOrderLinks; //0x0
    struct _LIST_ENTRY InMemoryOrderLinks; //0x10
    struct _LIST_ENTRY InInitializationOrderLinks; //0x20
    VOID* DllBase; //0x30
    VOID* EntryPoint; //0x38
    ULONG SizeOfImage; //0x40
    struct _UNICODE_STRING FullDllName; //0x48
    struct _UNICODE_STRING BaseDllName; //0x58
    union
    {

```

Image: _LDR_DATA_TABLE_ENTRY structure

The code below shows implementing this in assembly.

```

xor r11, r11

mov r8, gs:[30h] ;retrieve address of TEB
mov r9, [r8+60h] ; store PEB address
mov r10, qword ptr [r9+18h] ; pointer to _PEB_LDR_DATA
mov r8, qword ptr [r10+10h] ; pointer to InLoadOrderModuleList
mov r9, rcx ; pointer ntdll string

DO_AGAIN:
mov r10, qword ptr [r8+60h] ; address of Buffer inside struct FullDllName
mov rsi, r9

```

Image: Assembly Implementation

At this point we can match the name found with ntdll or any other dll we wish to find. If the name does not match we can get the pointer to the next structure in the linked list located at offset 0x0 in the _LDR_DATA_TABLE_ENTRY structure pointed to by InLoadOrderModuleList.

NOTE:- The fields InLoadOrderModuleList, InMemoryOrderModuleList point to the same Linked list structure but at different offsets. Therefore the offset to the FullDllName field will change depending on which list was used to get to the _LDR_DATA_TABLE_ENTRY linked list.

After finding the right DLL, the offset 0x30 inside LDR_DATA_TABLE_ENTRY can be used to retrieve the base address of the dll as seen in the image of the structure.

```

x64 Native Tools Command Prompt for VS 2022 - Temp_project.exe
C:\Program Files\Microsoft Visual Studio\2022\Community>cd C:\Users\bond008\Desktop\Temp_projects\Project0

C:\Users\bond008\Desktop\Temp_projects\Project0>compile.bat
Microsoft (R) Macro Assembler (x64) Version 14.34.31933.0
Copyright (C) Microsoft Corporation. All rights reserved.

Assembling: Temp_project_.asm
Temp_project.c

C:\Users\bond008\Desktop\Temp_projects\Project0>Temp_project.exe
ntdll.dll Base: 00007FF9F70A0000
NtAllocateVirtualMemory Base: 00007FF9F713C3B0

C:\Users\bond008\Desktop\Temp_projects\Project0>Temp_project.exe
ntdll.dll Base: 00007FF9F70A0000
NtAllocateVirtualMemory Base: 00007FF9F713C3B0

```

00007FF9F70A0000	00000000000001000	ntdll.dll				
00007FF9F70A1000	000000000000116000	".text"	Executable code	IMG	-R---	ERWC-
00007FF9F71B7000	00000000000001000	".rt"		IMG	ER---	ERWC-
00007FF9F71B8000	000000000000047000	".rdata"	Read-only initialized data	IMG	-R---	ERWC-
00007FF9F71FF000	0000000000000C000	".data"	Initialized data	IMG	-RW--	ERWC-
00007FF9F7208000	0000000000000F000	".pdata"	Exception information	IMG	-R---	ERWC-
00007FF9F721A000	00000000000004000	".mrdata"		IMG	-R---	ERWC-
00007FF9F721E000	00000000000001000	".00cfg"		IMG	-R---	ERWC-
00007FF9F721F000	00000000000007000	".rsrc"	Resources	IMG	-R---	ERWC-
00007FF9F728F000	00000000000001000	".reloc"	Base relocations	IMG	-R---	ERWC-

Image: Retrieved ntdll base

Retrieving Base Address Of API By Parsing PE Headers

Below is the layout of the PE headers to get to the Export Directory which contains the export address table.

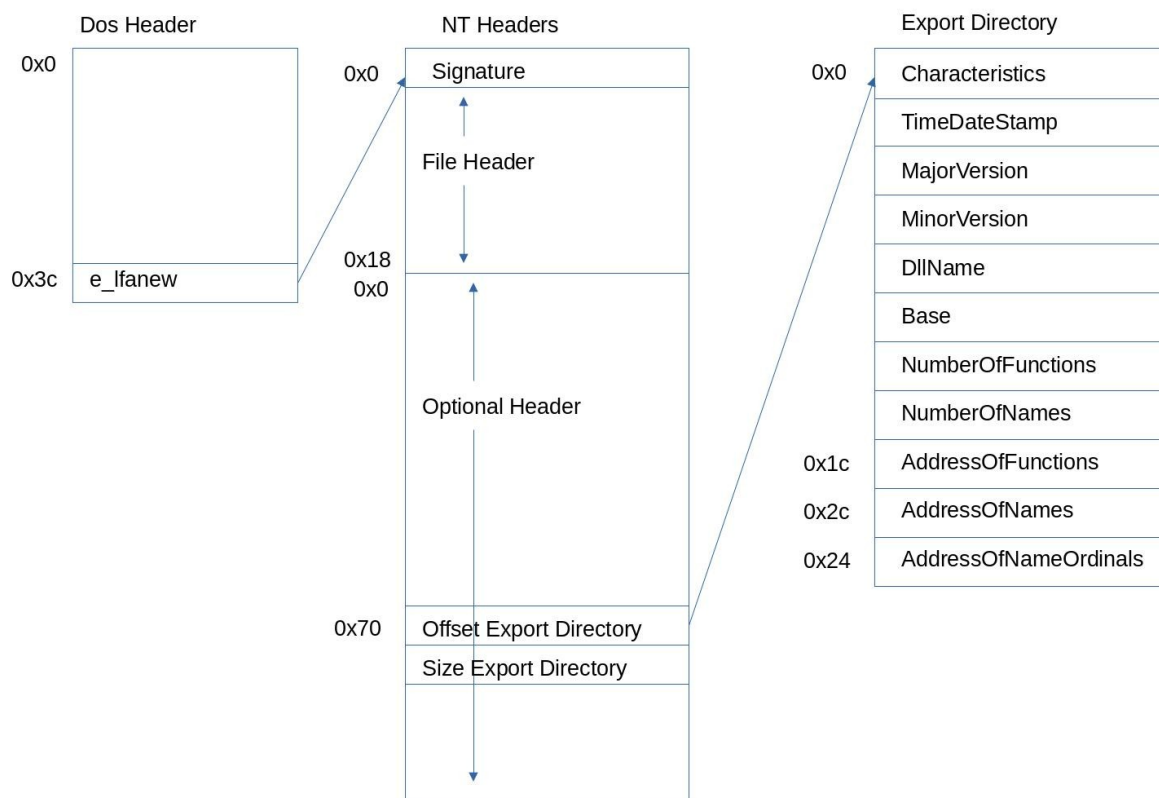


Image: PE headers layout

Taking a look at the above image the DOS header contains the offset to NT Headers at offset 0x3c in the `e_lfanew` field. At offset 0x70 from start of the Optional Header is the offset to Export Directory which contains the Export Address Table. The assembly code below shows the implementation.

```

57     lea r9, [rcx]                ;save DllBaseAddr
58     lea r13, [rdx]               ;save pointer NtApiName
59     mov rbx, r8                  ;save Len
60
61     mov r10d, dword ptr [r9+03ch] ;r8+3ch = e_lfanew (offset to NT Header)
62
63     mov r11d, dword ptr [r9+r10+018h+070h] ;base + [base+3ch] + 18h + 70h = pointer to export table
64     lea r11, [r9+r11]            ;Virtual address Export Table
65     mov r12d, dword ptr [r11+020h] ;pointer to function names array
66     lea r12, [r9+r12]
67     xor rdx, rdx
68

```

Image: Assembly Implementation

Inside Export Directory the important fields are:-

- Offset 0x1c points to array of AddressOfFunctions(Export Address Table).
- Offset 0x2c points to array of AddressOfNames.
- Offset 0x24 points to AddressOfNameOrdinals.

We first use the address of names array to loop through all the names of the API's to find the matching API. Below implementation shows how to achieve this.

```

69 RepeatCheck:
70     mov edi, dword ptr [r12+rdx*4] ;address of name of the first API
71     lea rdi, [r9+rdi]              ;address of API name
72     lea rsi, [r13]                 ;reload rsi with API name
73
74     mov rcx, r8
75     cld
76     repe cmpsb                     ;compare passed API name with API name in export table -> Address of Names Array
77     jrcxz ApiFound
78     inc rdx
79     jmp RepeatCheck
80

```

Image: assembly implementation

The index at which the name is found is used as an index in the address of name ordinals array. The value retrieved from the name ordinals array is then used as an index in the address of functions array to find the address of the API.

```

83 ApiFound:
84     mov r10d, dword ptr [r11+024h] ;Virtual offset Address of name ordinals field
85     lea r10, [r9+r10]              ;pointer to array address of name ordinals
86     mov r12w, word ptr [r10+rdx*2] ;ordinal number
87     movzx r12, r12w
88
89     mov r10d, dword ptr [r11+01ch] ;Virtual offset Address of Functions name field
90     lea r10, [r9+r10]              ;pointer to function array
91
92     mov r10d, dword ptr [r10+r12*4] ; Api Address VirtualOffset
93     lea r10, [r9+r10]
94     mov rax, r10

```

Image: assembly implementation

The below diagram shows the relation between all the arrays.

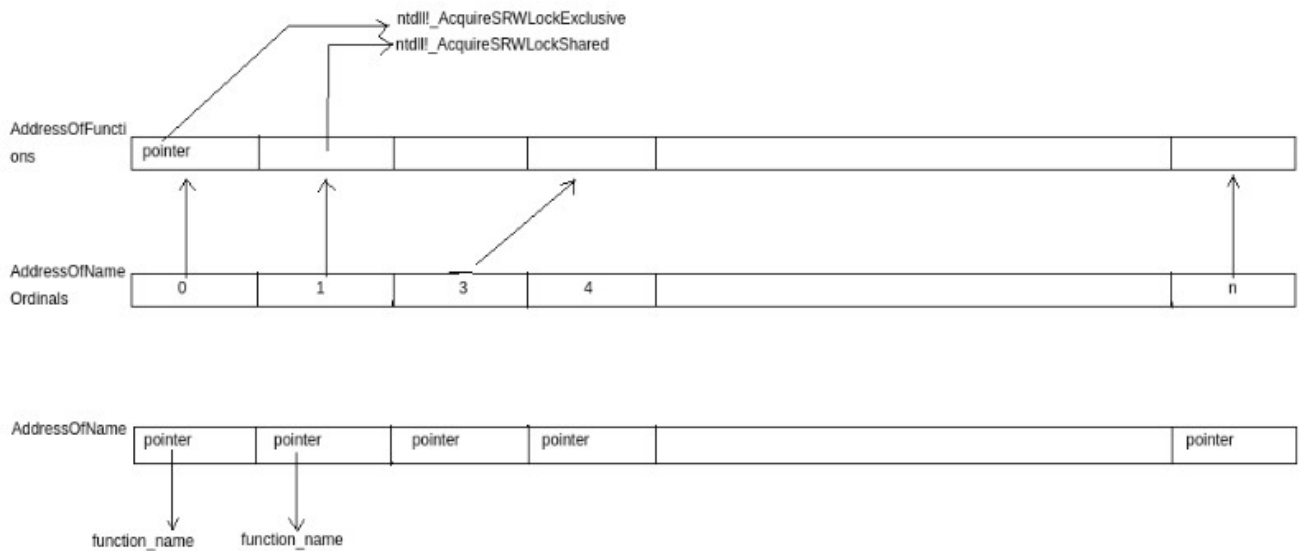


Image: Relation between different array tables inside Export Directory

The screenshot shows the Visual Studio 2022 interface. The assembly view displays the following code:

```

00007FF9F713C3B0 4C:8BD1 mov r10,rcx
00007FF9F713C3B3 B8:18000000 mov eax,18
00007FF9F713C3B8 F60425 0803FE7F 01 test byte ptr ds:[7FFE0308],1
00007FF9F713C3C0 75:03 jne ntdll.7FF9F713C3C5
00007FF9F713C3C2 0F05 syscall
00007FF9F713C3C4 C3 ret
00007FF9F713C3C5 CD:2E int 2E
00007FF9F713C3C7 C3 ret
00007FF9F713C3C8 0F1F8400 00000000 nop dword ptr ds:[rax+rax],eax
00007FF9F713C3D0 4C:8BD1 mov r10,rcx
00007FF9F713C3D3 B8:19000000 mov eax,19
00007FF9F713C3D8 F60425 0803FE7F 01 test byte ptr ds:[7FFE0308],1
  
```

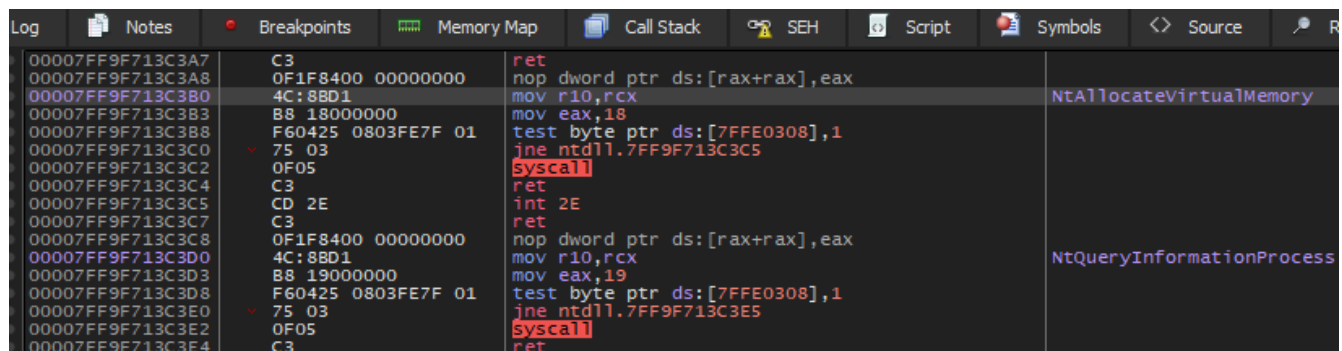
The command prompt window shows the following output:

```

C:\Program Files\Microsoft Visual Studio\2022\Community>cd C:\Users\bond008\Desktop\Temp_projects\Project0
C:\Users\bond008\Desktop\Temp_projects\Project0>Temp_project.exe
ntdll.dll Base: 00007FF9F70A0000
NtAllocateVirtualMemory Base: 00007FF9F713C3B0
  
```

Image: Retrieved API address

Retrieving Syscall Id's



Address	Disassembly	Comment
00007FF9F713C3A7	C3	ret
00007FF9F713C3A8	0F1F8400 00000000	nop dword ptr ds:[rax+rax],eax
00007FF9F713C3B0	4C:8BD1	mov r10,rcx
00007FF9F713C3B3	B8 18000000	mov eax,18
00007FF9F713C3B8	F60425 0803FE7F 01	test byte ptr ds:[7FFE0308],1
00007FF9F713C3C0	75 03	jne ntdll.7FF9F713C3C5
00007FF9F713C3C2	0F05	syscall
00007FF9F713C3C4	C3	ret
00007FF9F713C3C5	CD 2E	int 2E
00007FF9F713C3C7	C3	ret
00007FF9F713C3C8	0F1F8400 00000000	nop dword ptr ds:[rax+rax],eax
00007FF9F713C3D0	4C:8BD1	mov r10,rcx
00007FF9F713C3D3	B8 19000000	mov eax,19
00007FF9F713C3D8	F60425 0803FE7F 01	test byte ptr ds:[7FFE0308],1
00007FF9F713C3E0	75 03	jne ntdll.7FF9F713C3E5
00007FF9F713C3E2	0F05	syscall
00007FF9F713C3E4	C3	ret

Image: native api implementation inside ntdll.dll

The above image shows a typical native api definition in assembly. The value being moved to the eax register represents the Syscall Id. The Syscall Id is important as it represents an index into an array that contains the pointers to Syscall handler routines inside the kernel. But that's topic is out of scope for this write-up.

We can retrieve the syscall id after we have already retrieved the address of the api as shown below.


```
C:\Users\bond008\Desktop\Malware_development\Other_mal_dev\Projects\Project0>cd C:\Users\bond008\Desktop\Temp_projects
C:\Users\bond008\Desktop\Temp_projects>compile.bat
Microsoft (R) Macro Assembler (x64) Version 14.34.31933.0
Copyright (C) Microsoft Corporation. All rights reserved.

Assembling: Temp_project_.asm
Temp_project.c

C:\Users\bond008\Desktop\Temp_projects>Temp_project.exe
ntdll.dll Base: 0x00007FF9F70A0000
NtAllocateVirtualMemory Base: 0x00007FF9F713C3B0
NtAllocateVirtualMemory Syscall Id: 0x18
base address of the ntapi7
```

Image: Retrieved SyscallId

