2012

Deep Dive into OS Internals with Windbg

Malware and OS Internals

An approach towards reversing malwares, shellcodes and other malicious codes to understand the ways in which they use the OS Internals for their functionality.

Sudeep Pal Singh
Honeywell
1/26/2012

Table of Contents

Preface

There is more than one reason to reverse malwares these days. As time passes by, the awareness about Reverse Engineering is spreading. However, there are few obstacles encountered for a person new in the field of Reversing Viruses. Unlike other domains of security where you can make your way through with the reliance on some security tools, this field demands a strong understanding of the Operating System Internals and Assembly Language Programming.

Reversing Windows Internals

As the main aim of the document is to correlate the Operating Systems Internals and the approach of Reverse Engineering a Malware at code level, I have started the document by giving a brief overview of Windows OS Internals.

Since this subject requires hands on experience and it is not possible to visualize all the data structures on our own, it's important to have a debugger at hand. Any portable executable file can be used to deep dive into the OS internals. We do not need complicated tools to get an insight into the internal data structures of the operating systems. For the purpose of this document, I make use of the Debugging Tool provided by Microsoft called, Windbg.

The reader is introduced to methods that will allow them to practise along with using this document as a reference.

Portable Executable Anatomy

We are going to understand the Portable Executable structure, the concepts and various data directories inside it. To summarize, I will explain the OS internals with the help of Windbg.

At first, set up the Symbols of your Windbg to point to the Microsoft Symbols Server.

You can set the Environment Variable **_NT_SYMBOL_PATH** to symsrv*symsrv.dll*C:\Symbols*http://msdl.microsoft.com/download/symbols where, C:\Symbols is the folder on my file system where these symbols are cached as and when they are downloaded.

Let's open up an executable like notepad.exe in windbg.

Go to File->Open Executable and open up notepad.exe from %systemroot%\system32 folder.

Once this is done, windbg will present a list of all the modules which were loaded along with the main module, notepad.exe

Let us take a look at this output and try to make some sense out of it:

```
ModLoad: 01000000 01014000   notepad.exe
ModLoad: 7c900000 7c9b2000   ntdll.dll
ModLoad: 7c800000 7c8f6000   C:\WINXP\system32\kernel32.dll
ModLoad: 763b0000 763f9000   C:\WINXP\system32\comdlg32.dll
ModLoad: 77dd0000 77e6b000   C:\WINXP\system32\ADVAPI32.dll
ModLoad: 77e70000 77f03000   C:\WINXP\system32\RPCRT4.dll
ModLoad: 77fe0000 77ff1000   C:\WINXP\system32\Secur32.dll
ModLoad: 773d0000 774d3000   C:\WINXP\WinSxS\x86_Microsoft.Windows.Common-
Controls_6595b64144ccf1df_6.0.2600.6028_x-ww_61e65202\COMCTL32.dll
ModLoad: 77c10000 77c68000   C:\WINXP\system32\msvcrt.dll
ModLoad: 77f10000 77f59000   C:\WINXP\system32\GDI32.dll
ModLoad: 7e410000 7e4a1000   C:\WINXP\system32\USER32.dll
ModLoad: 77f60000 77fd6000   C:\WINXP\system32\SHLWAPI.dll
ModLoad: 7c9c0000 7d1d8000   C:\WINXP\system32\SHELL32.dll
ModLoad: 73000000 73026000   C:\WINXP\system32\WINSPOOL.DRV
(15c.ab8): Break instruction exception - code 80000003 (first chance)
```

When a PE is opened up, the OS loader will load the modules from which our executable imports the functions. Each of these DLLs will occupy a specific memory address range.

For instance, notepad.exe makes use of some Windows API functions exported by ntdll.dll. So, ntdll.dll was loaded along with notepad.exe. Also, this dll will occupy the address range: 7c900000 to 7c9b2000

By looking at this range, we get the Base Address of ntdll.dll

It's important to know the base address since as we investigate further into the internal structures of the operating system, we will discover that usually the value at hand is an RVA (Relative Virtual Address) which must be added to the image base address to fetch the values.

Next, enter .cls to clear the screen.

Now, we will display the list of all loaded modules using lm command.

Note: lm command will give us the same output as above with the address range of each module.

Every Portable Executable begins with a DOS Header having the structure of type, **IMAGE_DOS_HEADER**. We can view it in windbg using the Image Base Address of the PE image.

```
0:002> dt _IMAGE_DOS_HEADER 01000000
ntdll!_IMAGE_DOS_HEADER
   +0x000 e_magic          : 0x5a4d // MZ Signature of the PE
   +0x002 e_cblp           : 0x90
   +0x004 e_cp             : 3
   +0x006 e_crlc           : 0
   +0x008 e_cparhdr        : 4
   +0x00a e_minalloc       : 0
   +0x00c e_maxalloc       : 0xffff
   +0x00e e_ss             : 0
   +0x010 e_sp             : 0xb8
   +0x012 e_csum           : 0
   +0x014 e_ip             : 0
   +0x016 e_cs             : 0
   +0x018 e_lfarlc         : 0x40
   +0x01a e_ovno           : 0
   +0x01c e_res            : [4] 0
   +0x024 e_oemid          : 0
   +0x026 e_oeminfo        : 0
   +0x028 e_res2           : [10] 0
   +0x03c e_lfanew         : 224 // Decimal value of the PE File Header Offset
```

Here the two fields of interest to us are:

E_magic: This has the MZ signature hex value, 0x5a4d. Any portable executable will begin with the characters 'MZ' which are present in the DOS Header.

E_lfanew: This field is of importance to us since it holds the offset to the Portable Executable File Header. Please note that this value is in decimal and hence we must convert it to hexadecimal before locating the PE File Header using it.

In the above case, e_lfanew is 224 in decimal which is E0 in hex. Therefore, the PE File Header is at an offset E0 from the Image Base Address of notepad.exe.

Data Directories of Interest

Let us see how the PE header of our main executable, notepad.exe looks like.

The basic syntax is: !dh <image base address> <options>

In our case, image base address of notepad.exe is 01000000

I use the option -f along with this command, so that it does not display the Section Headers for .data, .text, .rsrc, .reloc and various other sections which might be present.

What I want to look into right now is something called as Data Directories, let's see how they look like:

!dh 01000000 –f

```
File Type: EXECUTABLE IMAGE
FILE HEADER VALUES
     14C machine (i386)
       3 number of sections
48025287 time date stamp Mon Apr 14 00:05:51 2008

       0 file pointer to symbol table
       0 number of symbols
      E0 size of optional header
     10F characteristics
            Relocations stripped
            Executable
            Line numbers stripped
            Symbols stripped
            32 bit word machine

OPTIONAL HEADER VALUES
     10B magic #
    7.10 linker version
    7800 size of code
    A600 size of initialized data
       0 size of uninitialized data
    739D address of entry point
    1000 base of code
         ----- new -----
01000000 image base
    1000 section alignment
     200 file alignment
       2 subsystem (Windows GUI)
    5.01 operating system version
    5.01 image version
    4.00 subsystem version
   14000 size of image
     400 size of headers
   18700 checksum
00040000 size of stack reserve
00011000 size of stack commit
00100000 size of heap reserve
00001000 size of heap commit
    8000  DLL characteristics
            Terminal server aware
       0 [       0] address [size] of Export Directory
    7604 [      C8] address [size] of Import Directory
    B000 [    8948] address [size] of Resource Directory
       0 [       0] address [size] of Exception Directory
       0 [       0] address [size] of Security Directory
       0 [       0] address [size] of Base Relocation Directory
    1350 [      1C] address [size] of Debug Directory
       0 [       0] address [size] of Thread Storage Directory
    18A8 [      40] address [size] of Load Configuration Directory
    1000 [     348] address [size] of Import Address Table Directory
```

Do you see the array of directories at the bottom highlighted in green?

This is the data directory array. It's an array of structures of **IMAGE_DATA_DIRECTORY** type each having two fields:

VirtualAddress and Size.

The Virtual Address is going to give the Relative Virtual Address (RVA) corresponding to the image base address. This RVA will point to another structure.

The data directories I am interested in are:

Import Directory
Import Address Table Directory
Export Directory

Import Directory

The second entry in the Data Directory Array is of the Import Table.

```
7604 [      C8] address [size] of Import Directory
```

We get the relative virtual address of the Import Table from this entry as 7604

Image Base Address = 01000000

Hence, the Virtual Address of Import Table = 01000000+7604= 1007604

Import Table contains an array of data structures of type, **_IMAGE_IMPORT_DESCRIPTOR**

This structure has the following form:

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

Each DLL which is loaded along with the main module will have its own _IMAGE_IMPORT_DESCRIPTOR structure.

There is no field that will give us an idea about how many _IMAGE_IMPORT_DESCRIPTOR structures are present in the array; however the end of this array is denoted by a structure who’s all the fields are set to 0.

Two important fields at this point are, **OriginalFirstThunk** and **FirstThunk**. Each of these will point to a Table.

OriginalFirstThunk -> **Imports Name Table**
FirstThunk -> **Import Address Table**

Both these tables are identical to each other till the point the PE is mapped to memory by the OS Loader. The reason I say they are similar to each other will become evident as we view these structures in debugger.

Let us start by viewing the memory at the Import Table Virtual Address:

```
0:001> dd 01000000+7604
01007604  00007990 ffffffff ffffffff 00007aac
01007614  000012c4 00007840 ffffffff ffffffff
01007624  00007afa 00001174 00007980 ffffffff
01007634  ffffffff 00007b3a 000012b4 000076ec
01007644  ffffffff ffffffff 00007b5e 00001020
01007654  000079b8 ffffffff ffffffff 00007c76
01007664  000012ec 000076cc ffffffff ffffffff
01007674  00007d08 00001000 00007758 ffffffff
```

Let us relate these fields (highlighted in blue) to the _IMAGE_IMPORT_DESCRIPTOR structure.

```
OriginalFirstThunk = 00007990
TimeDateStamp = ffffffff
ForwarderChain = ffffffff
Name = 00007aac
FirstThunk = 000012c4
```

For now, we will focus on the two fields, OriginalFirstThunk and FirstThunk.

Let us view the memory pointed by the OriginalFirstThunk:

```
0:001> dd 01000000+00007990
01007990  00007a7a 00007a5e 00007a9e 00007a50
010079a0  00007a40 00007a8a 00007a6a 00007a14
010079b0  00007a2c 00000000 00007bdc 00007bd4
010079c0  00007bca 00007bc2 00007bb6 00007bea
010079d0  00007ba0 00007b8c 00007b84 00007b7a
010079e0  00007b6c 00007bf4 00007bfc 00007c06
010079f0  00007c16 00007c22 00007c36 00007c46
01007a00  00007c56 00007c64 00007c82 00007bac
```

And we get a list of RVAs. Each of these RVAs is also called the **_IMAGE_THUNK_DATA** which in turn points to the structure of type, **_IMAGE_IMPORT_BY_NAME** structure:

```
typedef struct _IMAGE_IMPORT_BY_NAME {
    WORD    Hint;
    BYTE    Name[1];
} IMAGE_IMPORT_BY_NAME, *PIMAGE_IMPORT_BY_NAME;
```

Here, Name[1] is the Name of the Imported Function which can have a variable length.

So, there is a one to one correspondence between _IMAGE_THUNK_DATA and _IMAGE_IMPORT_BY_NAME structure.

Let us view it:

```
0:001> dc 01000000+00007a7a
01007a7a  6150000f 65536567 44707574 0057676c  ..PageSetupDlgW.
01007a8a  6547000a 65704f74 6c69466e 6d614e65  ..GetOpenFileNam
01007a9a  00005765 72500012 44746e69 7845676c  eW....PrintDlgEx
01007aaa  6f630057 676c646d 642e3233 00006c6c  W.comdlg32.dll..
01007aba  68530103 416c6c65 74756f62 001f0057  ..ShellAboutW...
01007aca  67617244 696e6946 00006873 72440023  DragFinish..#.Dr
01007ada  75516761 46797265 57656c69 001e0000  agQueryFileW....
01007aea  67617244 65636341 69467470 0073656c  DragAcceptFiles.
```

As you can see, the Names of Functions are stored in sequence. Also, if we compare with the _IMAGE_IMPORT_BY_NAME structure, we can observe that the HINT field is set to a NULL value. This value is set by the linker.

In this way, the Names of Functions are populated in the Import Names Table pointed by the OriginalFirstThunk.

Now, let us view the FirstThunk.

```
0:001> dd 01000000+000012c4
010012c4  763d4906 763c85ce 763d9d84 763cc3e1
010012d4  763b2306 763c7b9d 763c8602 763c0036
010012e4  763c7c2b 00000000 77c32dae 77c39e9a
010012f4  77c39ece 77c4aecf 77c4ab69 77c39eb6
01001304  77c1d036 77c35c94 77c1ce77 77c4802f
01001314  77c3fb0c 77c39e7e 77c617ac 77c1eeeb
01001324  77c39d67 77c4d695 77c623d8 77c1f1a4
01001334  77c1f1db 77c3537c 77c4ee4f 77c4806b
```

We see that it is already populated with virtual addresses. Let us view them,

```
0:001> ln 763d4906
(763d4906)   comdlg32!PageSetupDlgW

0:001> ln 763c85ce
(763c85ce)   comdlg32!FindTextW
```

So, these are the virtual address of the functions imported by our PE from the comdlg32.dll loaded module.

The reason we see this table populated with the virtual addresses already is that our PE is already loaded by the OS Loader and the Import Address Table is already filled with function pointers.

We will focus now on the Names field of the _IMAGE_IMPORT_DESCRIPTOR. This field is important to us since it gives information about the Name of the DLL.

```
0:001> dc 01000000+00007aac
01007aac  646d6f63 3233676c 6c6c642e 01030000  comdlg32.dll....
```

Similarly we can parse the Import Table to locate the next, _IMAGE_IMPORT_DESCRIPTOR structure as given below:

```
0:001> dd 01000000+7604
01007604  00007990 ffffffff ffffffff 00007aac
01007614  000012c4 00007840 ffffffff ffffffff
01007624  00007afa 00001174 00007980 ffffffff
01007634  ffffffff 00007b3a 000012b4 000076ec
01007644  ffffffff ffffffff 00007b5e 00001020
01007654  000079b8 ffffffff ffffffff 00007c76
01007664  000012ec 000076cc ffffffff ffffffff
01007674  00007d08 00001000 00007758 ffffffff
```

```
OriginalFirstThunk = 00007840
TimeDateStamp = ffffffff
ForwarderChain = ffffffff
Name = 00007afa
FirstThunk = 00001174
```

Let us view the name of the next loaded module:

```
0:001> dc 01000000+00007afa
01007afa  4c454853 2e32334c 006c6c64 6c43001b  SHELL32.dll...Cl
```

So, the next loaded module is shell32.dll

As mentioned before, the end of the _IMAGE_IMPORT_DESCRIPTOR array is denoted by a structure filled with all NULL values as shown below in the screenshot:

```
0:001> dd 01000000+7604 L50
01007604  00007990 ffffffff ffffffff 00007aac
01007614  000012c4 00007840 ffffffff ffffffff
01007624  00007afa 00001174 00007980 ffffffff
01007634  ffffffff 00007b3a 000012b4 000076ec
01007644  ffffffff ffffffff 00007b5e 00001020
01007654  000079b8 ffffffff ffffffff 00007c76
01007664  000012ec 000076cc ffffffff ffffffff
01007674  00007d08 00001000 00007758 ffffffff
01007684  ffffffff 000080ec 0000108c 000076f4
01007694  ffffffff ffffffff 0000825e 00001028
010076a4  00007854 ffffffff ffffffff 0000873c
010076b4  00001188 00000000 00000000 00000000
010076c4  00000000 00000000 00007ca2 00007cb6
```

Another important point worth discussing at this moment is how API Calls made in a program are replaced by bytecode by a compiler.

Let's say there is an API Call to **GetSystemTimeAsFileTime**()in our program. This function is exported by kernel32.dll

However the above API Call is replaced by the following instruction by our compiler:

```
CALL DWORD PTR DS:[010010EC]
```

The reason being, instead of hard coding the function pointer of the API in the bytecode, we give a pointer to the memory location where this function pointer will be stored.

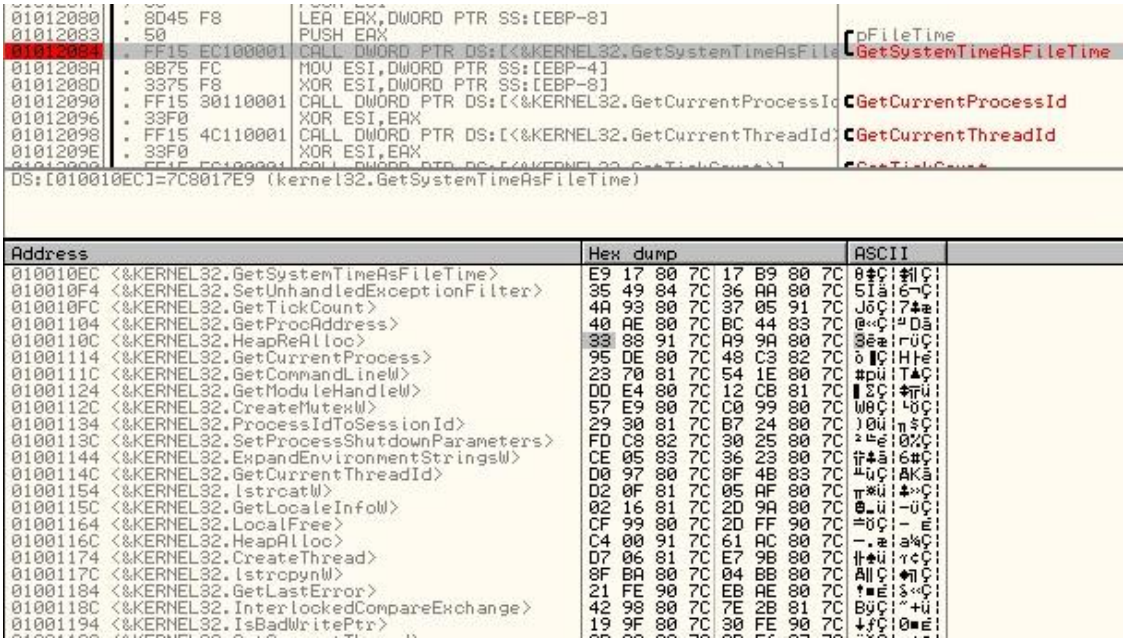
The advantage of doing so is that if we invoke this API in multiple locations in our program, we need not modify the addresses in all those locations if the function pointer happens to change in a newer version of the DLL.

In the above CALL Instruction, 010010EC is a memory location which has the address of GetSystemTimeAsFileTime API imported from kernel32.dll

```
DS:[010010EC] = 7C8017E9
```

Here, 010010EC is a memory address inside the Import Address Table of the main module.

This is shown in the screenshot below:



Import Address Table

When we look at the theory, it can be quite a complicated task to visualize. But why visualize when we have Windbg?

IAT is the Import Address Table which consists of the mappings between the absolute virtual addresses and the function names which are exported from different loaded modules.

As we saw in starting, there are various DLLs which are loaded along with notepad.exe. IAT gives us a list of function names which are imported from these loaded modules.

Let us see how I can grab that info. Looking at the above data directory we can see the following line:

```
1000 [     348] address [size] of Import Address Table Directory
```

Here, 1000 is the RVA which is the offset from the image base address of notepad.exe

image base address + RVA are going to point to the Import Address Table. We also have the size of this structure as 348 bytes.

I will display this structure as follows:

dps 01000000+1000 L348/4

here, the L parameter passed to the dps command is used to denote the size and I have divided by 4 to take steps of 4 bytes while displaying the addresses.

A subsection of the output is shown below:

```
     01001000  77dd6fff ADVAPI32!RegQueryValueExW
```

```
01001004  77dd6c27 ADVAPI32!RegCloseKey
01001008  77dfba55 ADVAPI32!RegCreateKeyW
0100100c  77dfbd35 ADVAPI32!IsTextUnicode
01001010  77dd7abb ADVAPI32!RegQueryValueExA
01001014  77dd7852 ADVAPI32!RegOpenKeyExA
01001018  77ddd767 ADVAPI32!RegSetValueExW
0100101c  00000000
01001020  773dd270 COMCTL32!CreateStatusWindowW
01001024  00000000
01001028  77f2dc61 GDI32!EndPage
0100102c  77f44cd2 GDI32!AbortDoc
01001030  77f2def1 GDI32!EndDoc
01001034  77f16e5f GDI32!DeleteDC
01001038  77f2f49e GDI32!StartPage
```

So, we get a nice clean list of the function names, the module name from which they are imported and the absolute virtual addresses.

It is good to know how to fetch an IAT of a PE image since we can use this output to detect any sort of IAT Hooks. IAT hooking a technique used by rootkits to take control of the functions in a DLL by overwriting the function pointers in the IAT.

The question now is, how did the IAT get filled up with these values at run time?

When the OS Loader starts the PE, it will fill the IAT with function pointers to the imported functions from various loaded modules. We will try to find out how it got to know these function pointers.

Export Directory

Let's take an example to understand this better. My PE image makes use of a function called GetCurrentProcess from kernel32.dll. The kernel32.dll file exports these functions to our PE image.

PE image imports the functions from (<-) kernel32.dll
kernel32.dll exports the functions to (->) our PE image.

So, there's a handshake taking place.

This is a complicated concept to grasp but it should become clear with an example.

Each loaded module will have its own Export Directory. This export directory is a structure called, **IMAGE_EXPORT_DIRECTORY**, having the following form:

```
Private Type IMAGE_EXPORT_DIRECTORY
   Characteristics As Long
   TimeDateStamp As Long
   MajorVersion As Integer
   MinorVersion As Integer
   lpName As Long
   Base As Long
   NumberOfFunctions As Long
   NumberOfNames As Long
   lpAddressOfFunctions As Long
   lpAddressOfNames As Long
   lpAddressOfNameOrdinals As Long
End Type
```

This structure will help us understand the export directory. I would recommend the article by Iczelion on Export Directory to understand it better.

A summary of the Structure: This structure contains pointers to 3 Arrays. The pointers are in the form of RVAs relative to the base address of the Image.

What are these arrays?

AddressOfFunctions: It is an array of RVAs of the functions in the module.

AddressOfNames: It is an array of RVAs each corresponding to the function name strings of exported functions.

AddressOfNameOrdinals: This array is in sync with the AddressOfNames array and there is a one to one correspondence between the two. It gives an index or an offset into the AddressOfFunctions array to get the Address of the Function Name.

So, the flow is like this:

The function name is fetched using Export Names Table and the corresponding entry in the Export Ordinals Table is looked up to get the index or offset. This index is used to parse the Export Address Table and fetch the Function Address.

Export Names Table > Export Ordinals Table -> Export Address Table = Function Address (VA).

This should make it clear how the OS Loader gets to know the addresses of Functions which are imported by the main module from these loaded modules.

Manual Walkthrough of Export Directory

Let's say, I want to view all the functions exported by the GDI32.dll module.

I need to first view the data directory of gdi32.dll:

!dh gdi32 -f

```
File Type: DLL
FILE HEADER VALUES
     14C machine (i386)
       4 number of sections
4900717E time date stamp Thu Oct 23 18:13:42 2008

       0 file pointer to symbol table
       0 number of symbols
      E0 size of optional header
    210E characteristics
            Executable
            Line numbers stripped
            Symbols stripped
            32 bit word machine
            DLL

OPTIONAL HEADER VALUES
     10B magic #
    7.10 linker version
   42C00 size of code
    3000 size of initialized data
       0 size of uninitialized data
    6587 address of entry point
    1000 base of code
         ----- new -----
77f10000 image base
    1000 section alignment
     200 file alignment
       3 subsystem (Windows CUI)
    5.01 operating system version
```

```
    5.01 image version
    4.10 subsystem version
   49000 size of image
     400 size of headers
   52F15 checksum
00040000 size of stack reserve
00001000 size of stack commit
00100000 size of heap reserve
00001000 size of heap commit
       0  DLL characteristics
    1CA0 [    3CD6] address [size] of Export Directory // We need to explore this
   42F24 [      50] address [size] of Import Directory
   46000 [     3D0] address [size] of Resource Directory
       0 [       0] address [size] of Exception Directory
       0 [       0] address [size] of Security Directory
   47000 [    1870] address [size] of Base Relocation Directory
   43A10 [      38] address [size] of Debug Directory
       0 [       0] address [size] of Description Directory
       0 [       0] address [size] of Special Directory
```

We are interested in the Export Directory:

```
1CA0 [    3CD6] address [size] of Export Directory
```

So, the offset of the Export Directory is 1CA0 relative to the Image Base Address.

Let's view the IMAGE_EXPORT_DIRECTORY structure in the memory:

The base address of gdi32.dll is: 77f10000

dd 77f10000+1CA0

```
77f11ca0  00000000 48ff5bdd 00000000 0000349c
77f11cb0  00000001 00000262 00000262 00001cc8
77f11cc0  00002650 00002fd8 00034cd2 00036009
77f11cd0  0002f7d5 00019415 0001942f 0002ffc5
77f11ce0  0002fe78 0002ffab 00035069 0002d182
77f11cf0  00041fda 0003511b 00035207 000204cc
77f11d00  00042b64 00042b78 0001cd62 0001d4b0
77f11d10  00006f79 0001f0a6 0001f0ba 00042b16
```

Focus on first 3 rows. The first column has the memory address. We need to focus on the values in the next 4 columns for the first 3 rows.

Parsing these values and comparing them with the IMAGE_EXPORT_DIRECTORY structure definition we get:

```
Characteristics = 00000000
TimeDateStamp = 48ff5bdd
MajorVersion = 0000
MinorVersion = 0000
lpName = 0000349c
Base = 00000001
NumberOfFunctions = 00000262
NumberOfNames = 00000262
lpAddressOfFunctions = 00001cc8
lpAddressOfNames = 00002650
lpAddressOfNameOrdinals = 00002fd8
```

From this, we get the pointers to the 3 arrays.

Let's get the list of function names now:

The RVA of the pointer to AddressOfNames array is: 00002650

to dump the contents of this array, let's add it to the base address and display:

dd 77f10000+00002650

```
77f12650  000034a6 000034af 000034b9 000034ce
77f12660  000034df 000034f2 00003505 0000351d
77f12670  0000352e 00003537 00003546 00003555
77f12680  00003559 0000355f 0000357b 00003592
77f12690  000035a7 000035c0 000035ca 000035d1
77f126a0  000035df 000035f2 00003605 0000360e
77f126b0  00003621 00003633 00003639 0000364f
77f126c0  00003664 00003675 00003681 0000368f
```

So, we got the list of RVAs now. Each of these RVAs when added to the base address of gdi32.dll will point to the Function Name string. Let's check by taking the first RVA from this list: 000034a6

da gdi32+000034a6

```
77f134a6  "AbortDoc"
```

To get the next function name:

da gdi32+000034af

```
77f134af  "AbortPath"
```

This way, we can get the list of function names.

The **NumberOfNames** field in the **IMAGE_EXPORT_DIRECTORY** structure had a value of 262. It means that there are 262 function names exported by gdi32.dll. We cannot list all of them manually, that would be a very tedious task.

So, I will make use of a windbg script to iterate through these RVAs and grab the corresponding Function Name String.

Before we write the script, let's make sure that our expression evaluator is set to MASM:

Current Expression Evaulator:

.expr /q

Current expression evaluator: MASM - Microsoft Assembler expressions

Now, the windbg script:

```
r? @$t0 = ((int *) (0x77f12650))

.for (r @$t1 = 0; @$t1 < 100; r @$t1 = @$t1 + 1)
{da gdi32+(@@c++(@$t0[@$t1]));}
```

0x77f12650 is a pointer to the first element of the AddressOfNames array.

It is obtained by adding the base address of gdi32.dll (0x77f10000) to the RVA of AddressOfNames array (00002650) as given in the IMAGE_EXPORT_DIRECTORY structure.

I save this script as parser.wds in the path: C:\Scripts\parser.wds and execute as follows:

$$><C:\Scripts\parser.wds

It gives me the list of function names as follows:

```
77f134a6  "AbortDoc"
77f134af  "AbortPath"
77f134b9  "AddFontMemResourceEx"
```

```
77f134ce  "AddFontResourceA"
77f134df  "AddFontResourceExA"
77f134f2  "AddFontResourceExW"
77f13505  "AddFontResourceTracking"
77f1351d  "AddFontResourceW"
77f1352e  "AngleArc"
77f13537  "AnimatePalette"
77f13546  "AnyLinkedFonts"
77f13555  "Arc"
77f13559  "ArcTo"
77f1355f  "BRUSHOBJ_hGetColorTransform"
```

In this way, we can walk through the list of function names using the Export Directory structure of a loaded module.

Important thing to note is, if we use the x command to list the export symbols, that's going to be more verbose and different from the above output:

x gdi32!* = All the exported symbols.

Process Environment Block

Process Environment Block is an important data structure from an exploiter's perspective. A shellcode executes a set of function APIs and for this it must locate and load them.

A common way of doing this is to make use of LoadLibraryA function which is exported by kernel32.dll OS module.

I showed in my previous article about IATs and EATs, that we can parse the EAT of a loaded module to find out the address of a function API exported by the module.

Similarly, if we can find the base address of kernel32.dll then we can parse its Export Address Table and locate LoadLibraryA or LoadLibraryW functions which can further be used to load and execute the functions used by shellcode.

This is not a new technique and there are snippets available on the net which show you how to do this using assembly language code. What I felt after reading them is, there is not a good explanation provided along with those codes to describe what they are doing.

So, I will make use of windbg to walk through the assembly language code which is used to locate the base address of kernel32.dll

First let's explore the Process Environment Block. I attach my windbg to notepad.exe on a Win XP SP3 platform.

PEB is located within the virtual address space of the loaded process. This address is most often, 7ffda000 but not always. There are different ways to get the PEB base address in the process VA space.

Different methods to locate the PEB

Method 1: Windbg provides pseudo registers like **$peb** which point to the base address of PEB data structure within the process VA space. Let's read this value by prefixing it with @ symbol.

```
0:001> dt @$peb
output: 7ffda000
```

Method 2: You can use the !peb command to read this value. !peb will display the complete PEB data structure with values.

The first few lines of output:

```
0:001> !peb
PEB at 7ffda000 <-------- The base address of PEB
    InheritedAddressSpace:    No
    ReadImageFileExecOptions: No
    BeingDebugged:            Yes
```

Method 3:

Now let's say I am debugging in kernel mode and I want to inspect the user mode state of the processes and grab the address of PEB. So, here's how to do it:

```
lkd> !process -0 0

**** NT ACTIVE PROCESS DUMP ****
.....................
PROCESS 8290b020  SessionId: 0  Cid: 081c    Peb: 7ffda000  ParentCid: 058c
    DirBase: 0e6c0240  ObjectTable: e23f1388  HandleCount:  41.
    Image: notepad.exe
.....................
```

This will display the information about all the processes running on the local system right now. Along with other pieces of useful information like DirBase, it also displays the location of PEB.

Method 4:

PEB is a data structure in the user mode and specific to an application process running in the user mode. Similarly in the kernel mode, we have the **_EPROCESS** data structure which points to the PEB.

Using the Process number grabbed from the above output, I can display the **_EPROCESS** structure for our notepad.exe user mode process.

```
lkd> dt nt!_EPROCESS 8290b020

  +0x000 Pcb              : _KPROCESS
  +0x06c ProcessLock      : _EX_PUSH_LOCK
  .....................
  +0x1b0 Peb              : 0x7ffda000 _PEB  <--------- Pointer to PEB
  .....................
```

Method 5:

We have so far seen how to do all of this using Windbg. However, when we are writing a shellcode, we have to find a way to reference the base address of PEB using assembly language code. This is done using the concept that, in any

Windows NT operating system, PEB is always located at an offset 30 to fs segment register.

Hence,

fs:[30] -> points to PEB.

So, by using a simple MOV instruction, we can get a pointer to PEB in a register as follows:

```
mov edx, fs:[30]
```

More on this later.

Method 6:

Yet another way to locate the Process Environment Block in the user mode is by using the Thread Environment Block.

Using the !teb command will display the thread environment block for the current executing thread in our process. This has a pointer to the PEB as shown below:

```
0:001> !teb
TEB at 7ffdc000
    ExceptionList:        0096ffe4
    StackBase:            00970000
    StackLimit:           0096f000
    SubSystemTib:         00000000
    FiberData:            00001e00
    ArbitraryUserPointer: 00000000
    Self:                 7ffdc000
    EnvironmentPointer:   00000000
    ClientId:             00000a14 . 00000a44
    RpcHandle:            00000000
    Tls Storage:          00000000
    PEB Address:          7ffda000    <------------ Pointer to PEB
    LastErrorValue:       0
    LastStatusValue:      0
    Count Owned Locks:    0
    HardErrorMode:        0
```

Method 7:

Similar to Method 1 above, we have the pseudo register, $teb which stores the address of Thread Environment Block.

```
0:001> dt @$teb
output: 7ffdc000
```

Now using this, I can display the complete TEB structure as follows:

```
0:001> dt nt!_TEB @$teb
ntdll!_TEB
   +0x000 NtTib            : _NT_TIB
   +0x01c EnvironmentPointer : (null)
   +0x020 ClientId         : _CLIENT_ID
   ...................
   +0x030 ProcessEnvironmentBlock : 0x7ffda000 _PEB <---------- Pointer to PEB
```

So, at offset 0x030 in the TEB, we have a pointer to PEB.

This also means that we can reference the TEB using, fs:[0] and since we have PEB at offset 0x030 in the TEB, so PEB can be located using, fs:[30]. Putting all the pieces of information together, this makes more sense now.

Understanding an Example Shellcode

Now, let's look at the shellcode which is used to retrieve the base address of kernel32.dll

```
1. xor ebx, ebx              ; clear ebx
2. mov ebx, fs:[ 0x30 ]      ; get a pointer to the PEB
3. mov ebx, [ ebx + 0x0C ]
4. mov ebx, [ ebx + 0x1C ]
5. mov ebx, [ ebx ]
6. mov ebx, [ ebx + 0x08 ]
```

The first two instructions are straight forward and so a comment is enough to explain what they do. For the second instruction, I have elaborated in the methods above that how we derive that PEB is at offset 0x30 to fs segment register.

Using _PEB_LDR_DATA

For the next 4 instructions, I will explain in detail since they are not so easy to understand.

After instruction 2, I have the pointer to PEB in ebx register.

Instruction 3:

mov ebx, [ebx+0x0c]

I am moving the value stored at offset 0x0c of PEB into the register ebx. Using windbg, let's understand it.

```
0:001> dt nt!_PEB @$peb
ntdll!_PEB
   +0x000 InheritedAddressSpace : 0 ''
   +0x001 ReadImageFileExecOptions : 0 ''
   +0x002 BeingDebugged    : 0x1 ''
   +0x003 SpareBool        : 0 ''
   +0x004 Mutant           : 0xffffffff
   +0x008 ImageBaseAddress : 0x01000000
   +0x00c Ldr              : 0x001a1e90 _PEB_LDR_DATA <-- We are storing this value in ebx
```

As we can see, at offset, 0x00c we have a pointer to the **_PEB_LDR_DATA** structure.

So, register ebx now has 0x001a1e90 memory address stored in it.

Instruction 4:

We are moving the value stored at offset, 0x1c in the _PEB_LDR_DATA structure into the register ebx.

We can view the structure along with the values by passing it the address, 0x001a1e90

```
0:001> dt nt!_PEB_LDR_DATA 0x001a1e90
ntdll!_PEB_LDR_DATA
   +0x000 Length           : 0x28
   +0x004 Initialized      : 0x1 ''
```

```
   +0x008 SsHandle         : (null)
   +0x00c InLoadOrderModuleList : _LIST_ENTRY [ 0x1a1ec0 - 0x1a2e90 ]
   +0x014 InMemoryOrderModuleList : _LIST_ENTRY [ 0x1a1ec8 - 0x1a2e98 ]
   +0x01c InInitializationOrderModuleList : _LIST_ENTRY [ 0x1a1f28 - 0x1a2ea0 ]
   +0x024 EntryInProgress  : (null)
```

We can see 3 linked lists stored in this structure.

Windows OS Loader maintains information about how the DLLs were loaded into the memory in 3 ways:

- Based on the order in which they were loaded: **InLoadOrderModuleList**
- Based on the order in which they appear in memory: **InMemoryOrderModuleList**
- Based on the order in which they were initialized: **InInitializationOrderModuleList**

In our case, we are referencing offset, 0x1c into the _PEB_LDR_DATA structure which points to the InInitializationOrderModuleList.

Before we look further into these entries and what is stored in them, let's first understand these lists.

All these lists are of type, _LIST_ENTRY.

Using windbg, I can see the structure of _LIST_ENTRY as:

```
0:001> dt nt!_LIST_ENTRY
ntdll!_LIST_ENTRY
   +0x000 Flink            : Ptr32 _LIST_ENTRY
   +0x004 Blink            : Ptr32 _LIST_ENTRY
```

So it is a set of Forward Pointer and Backward Pointer. It's a double linked list keeping track of both the previous node and the next node. However, we have to understand what data items are they pointing to.

Let's expand the InInitializationOrderModuleList field of _PEB_LDR_DATA structure as follows:

```
0:001> dt nt!_PEB_LDR_DATA 0x001a1e90 InInitializationOrderModuleList.Flink /r1
ntdll!_PEB_LDR_DATA
   +0x01c InInitializationOrderModuleList      : [ 0x1a1f28 - 0x1a2ea0 ]
      +0x000 Flink                             : 0x001a1f28 _LIST_ENTRY [ 0x1a1fc8 - 0x1a1eac ]
```

I am expanding the Flink of this List which gives me the first memory address as, 0x001a1f28

This memory address is moved into the ebx register.

Instruction 5:

Now, we are reading the data stored at this memory address. Let's dump this data using dd command as follows:

```
dd 0x001a1f28
```

```
0:001> dd 0x001a1f28
001a1f28  001a1fc8 001a1eac 7c900000 7c912afc
001a1f38  000b2000 02080036 7c980048 00140012
...................
```

We are storing the address, 001a1fc8 into ebx

Instruction 6:

We are moving the value stored at offset 0x8 from 001a1fc8 into the register ebx.

Let's dump the contents of 001a1fc8 address.

```
0:001> dd 001a1fc8
001a1fc8  001a2248 001a1f28 7c800000 7c80b64e
001a1fd8  000f6000 003e003c 001a1f70 001a0018
................
```

At offset 0: 001a2248
At offset 4: 001a1f28
At offset 8: 7c800000

So, we are moving the address 7c800000 into ebx register and this should be the base address of kernel32.dll

Let us confirm this using lm command.

```
0:001> lm
start    end        module name
01000000 01014000   notepad    (deferred)
5ad70000 5ada8000   UxTheme    (deferred)
5cb70000 5cb96000   ShimEng    (deferred)
6f880000 6fa4a000   AcGenral   (deferred)
73000000 73026000   WINSPOOL   (deferred)
74720000 7476c000   MSCTF      (deferred)
755c0000 755ee000   msctfime   (deferred)
76390000 763ad000   IMM32      (deferred)
763b0000 763f9000   comdlg32   (deferred)
769c0000 76a74000   USERENV    (deferred)
76b40000 76b6d000   WINMM      (deferred)
77120000 771ab000   OLEAUT32   (deferred)
773d0000 774d3000   COMCTL32   (deferred)
774e0000 7761e000   ole32      (deferred)
77be0000 77bf5000   MSACM32    (deferred)
77c00000 77c08000   VERSION    (deferred)
77c10000 77c68000   msvcrt     (deferred)
77dd0000 77e6b000   ADVAPI32   (deferred)
77e70000 77f03000   RPCRT4     (deferred)
77f10000 77f59000   GDI32      (deferred)
77f60000 77fd6000   SHLWAPI    (deferred)
77fe0000 77ff1000   Secur32    (deferred)
```

From the above list of loaded modules, we can confirm that, 7c800000 is indeed the base address of kernel32.dll

Using _LDR_DATA_TABLE_ENTRY

In the above method, I have dumped the contents of memory addresses (Flinks) and used the offsets to see what is there. But to understand better, we need to look
deeper into the double linked lists.

The Flinks of the lists stored in _PEB_LDR_DATA structure actually point to a data structure, **_LDR_DATA_TABLE_ENTRY**.

Let's view the structure.

```
0:001> dt nt!_LDR_DATA_TABLE_ENTRY
ntdll!_LDR_DATA_TABLE_ENTRY
   +0x000 InLoadOrderLinks : _LIST_ENTRY
   +0x008 InMemoryOrderLinks : _LIST_ENTRY
   +0x010 InInitializationOrderLinks : _LIST_ENTRY
   +0x018 DllBase          : Ptr32 Void
   +0x01c EntryPoint       : Ptr32 Void
   +0x020 SizeOfImage      : Uint4B
   +0x024 FullDllName      : _UNICODE_STRING
   +0x02c BaseDllName      : _UNICODE_STRING
   +0x034 Flags            : Uint4B
   +0x038 LoadCount        : Uint2B
   +0x03a TlsIndex         : Uint2B
   +0x03c HashLinks        : _LIST_ENTRY
   +0x03c SectionPointer   : Ptr32 Void
   +0x040 CheckSum         : Uint4B
   +0x044 TimeDateStamp    : Uint4B
   +0x044 LoadedImports    : Ptr32 Void
   +0x048 EntryPointActivationContext : Ptr32 Void
   +0x04c PatchInformation : Ptr32 Void
```

As can be seen, the Lists in the data structure _PEB_LDR_DATA are pointing to Links in the _LDR_DATA_TABLE_ENTRY data structure.

We saw before that, _LIST_ENTRY is a set of two pointers, flink and blink. But we could not see the data item of the double linked list. We will use the concept of macro **CONTAINING_RECORD** of linked lists to read the real data elements of the list.

Let's look only at the fields of type, _LIST_ENTRY of the two structures:

```
0:001> dt nt!_PEB_LDR_DATA
ntdll!_PEB_LDR_DATA
   +0x00c InLoadOrderModuleList : _LIST_ENTRY
   +0x014 InMemoryOrderModuleList : _LIST_ENTRY
   +0x01c InInitializationOrderModuleList : _LIST_ENTRY

0:001> dt nt!_LDR_DATA_TABLE_ENTRY
ntdll!_LDR_DATA_TABLE_ENTRY
   +0x000 InLoadOrderLinks : _LIST_ENTRY
   +0x008 InMemoryOrderLinks : _LIST_ENTRY
   +0x010 InInitializationOrderLinks : _LIST_ENTRY
```

There is a one to one correspondence between the _LIST_ENTRY fields of the two structures. For instance, the InLoadOrderModuleList.Flink points to InLoadOrderLinks entry of _LDR_DATA_TABLE_ENTRY structure at offset 0.

InMemoryOrderModuleList.Flink field of _PEB_LDR_DATA points into InMemoryOrderLinks field of _LDR_DATA_TABLE_ENTRY at offset 0x08

InInitializationOrderModuleList.Flink field of _PEB_LDR_DATA points into InInitializationOrderLinks field of _LDR_DATA_TABLE_ENTRY at offset 0x010.

In windbg, if you want to display the offsets of fields in a structure you can use the following command,

#FIELD_OFFSET(Structure Name, Field Name)

Now, let's again take the above example shellcode and look into it:

Instruction 4:

We got that the Flink of InInitializationOrderModuleList points to 0x001a1f28 which is stored in ebx.

Instruction 5:

We take the next Flink entry, 001a1fc8 of InInitializationOrderModuleList and move it into ebx register.

Now, let's use the above structures to see what data element of the linked list it points to.

```
0:001> dt nt!_LDR_DATA_TABLE_ENTRY (001a1fc8 -
@@(#FIELD_OFFSET(_LDR_DATA_TABLE_ENTRY,InInitializationOrderLinks)))

ntdll!_LDR_DATA_TABLE_ENTRY
   +0x000 InLoadOrderLinks : _LIST_ENTRY [ 0x1a2058 - 0x1a1f18 ]
   +0x008 InMemoryOrderLinks : _LIST_ENTRY [ 0x1a2060 - 0x1a1f20 ]
   +0x010 InInitializationOrderLinks : _LIST_ENTRY [ 0x1a2248 - 0x1a1f28 ]
   +0x018 DllBase          : 0x7c800000
   +0x01c EntryPoint       : 0x7c80b64e
   +0x020 SizeOfImage      : 0xf6000
   +0x024 FullDllName      : _UNICODE_STRING "C:\WINXP\system32\kernel32.dll"
   +0x02c BaseDllName      : _UNICODE_STRING "kernel32.dll"
   +0x034 Flags            : 0x80084004
   +0x038 LoadCount        : 0xffff
   +0x03a TlsIndex         : 0
   +0x03c HashLinks        : _LIST_ENTRY [ 0x7c97e2d0 - 0x7c97e2d0 ]
   +0x03c SectionPointer   : 0x7c97e2d0
   +0x040 CheckSum         : 0x7c97e2d0
   +0x044 TimeDateStamp    : 0x49c4f2bb
   +0x044 LoadedImports    : 0x49c4f2bb
   +0x048 EntryPointActivationContext : (null)
   +0x04c PatchInformation : (null)
```

At offsets 0x024 and 0x02c, we can see **FullDllName** and **BaseDllName** respectively which give us the name of DLL. This shows us that the second entry in the InInitializationOrderModuleList contains information about the kernel32.dll module.

Practical Example with Rustock.B Rootkit

This time, we will apply the knowledge gained from previous topics about IAT, EAT of a PE and also about Process Environment Block.

I have taken the Rustock.B rootkit as an example and this rootkit makes use of a myriad of function APIs exported by kernel32.dll for its working. It has to get the function pointers to memory addresses of these exported Function APIs. And we are going to see how exactly, the rootkit does this.

Knowledge of the previous topics is highly recommended before you read further.

I have added my comments to the assembly language instructions to make it easier for you to understand.

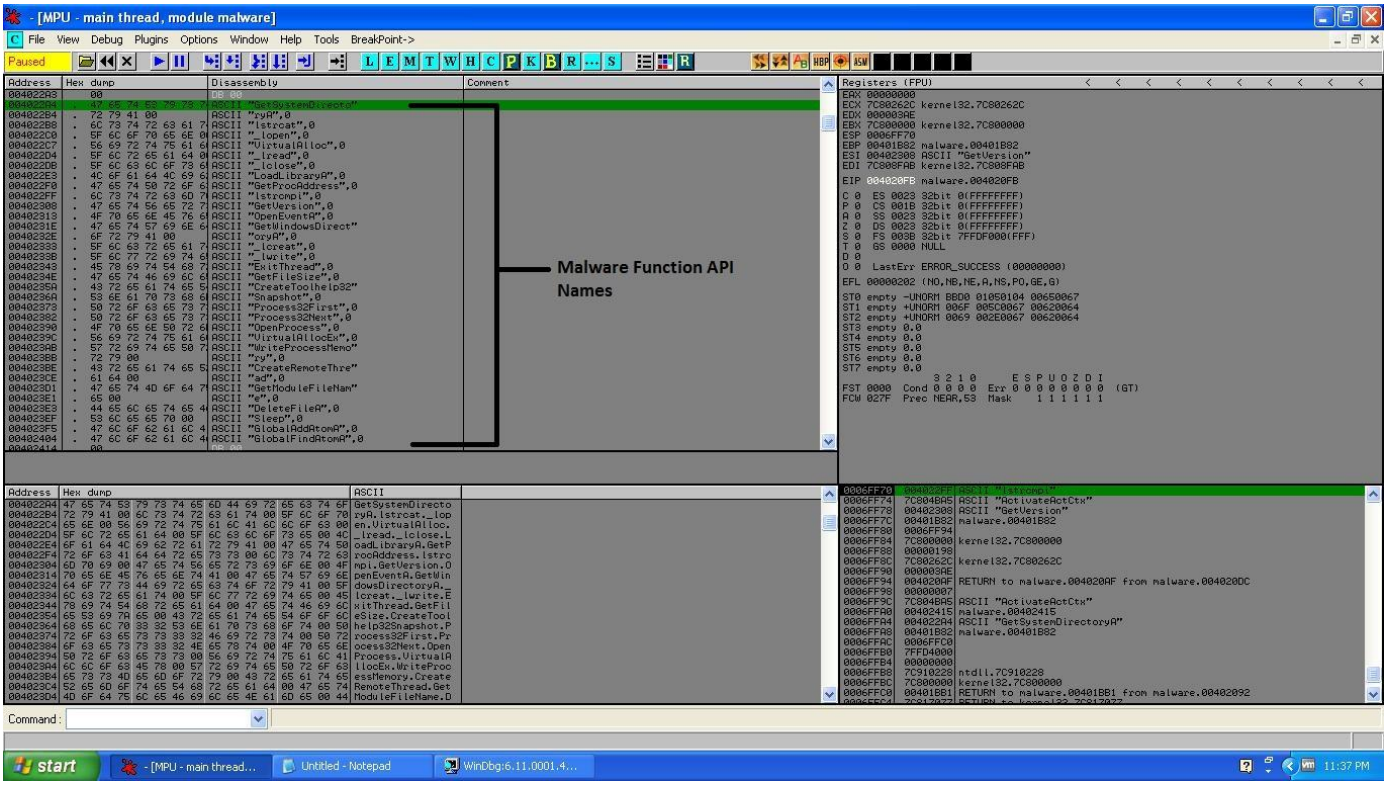
After unpacking the rootkit, this is what we see:

Function I:

```
pop ebp
sub ebp, 9
mov eax, fs:[30]   ; Pointer to Process Environment Block
mov eax, [eax+c]   ; Pointer to _PEB_LDR_DATA structure
mov eax, [eax+1c]  ; Pointer to InInitializationOrderModuleList.Flink
mov eax, [eax]        ; The second entry
mov eax, [eax+8]   ; The base address of kernel32.dll
lea esi, [ebp+722] ; At memory address, 004022A4, we have the function name GetSystemDirectoryA
lea edi, [ebp+893] ; 00402415
```

The above set of instructions will grab a pointer to Process Environment Block and then use it to find the base address of kernel32.dll. I have covered this in detail in my previous article.

Also, in the above instructions, we have grabbed the name of Function API, GetSystemDirectoryA from memory location, 004022A4. The Rootkit stores the array of names of all the Function APIs which it uses, at memory address, 004022A4 as shown in the screenshot below.



Then it parses this list and grabs the function pointers to each of the above APIs and stores them in another array.

Function II:

```
1. pushad                ; save the contents of all the registers on the stack
2. mov ebx, eax          ; kernel32.dll base address is placed in ebx
3. mov ecx, [ebx+3c]     ; e_lfanew field in IMAGE_DOS_HEADE is at offset 0x3c and it points to the PE header
4. mov ecx, [ebx+ecx+78]; at offset 78h from IMAGE_NT_HEADER, we have the RVA of Export Directory Table of
kernel32.dll
5. add ecx,ebx           ; add image base address
6. mov edx, [ecx+20]     ; At offset 20h in the Export Directory table, we have the RVA of AddressOfNames
array
7. add edx, ebx          ; add the image base address
8. mov edi, [edx]        ; move the RVA of first function name in kernel32.dll in edi
9. add edi, ebx          ; add the image base address
a. push edi              ; push the First Function Name, ActivateActCtx to the stack
b. push -1
c. call 004020dc
```

In the above set of instructions, we have used the IMAGE_NT_HEADER to locate the Export Data Directory of the Rootkit. In a Portable Executable you can find the Export Table at Offset 0x78 from the PE File Header.

We have got a pointer to the first function name in the AddressOfNames array inside the Export Directory. This function name is, ActivateActCtx.

We have to find the function pointer to GetSystemDirectoryA. To do this, we have to first find out the index of API function in the AddressOfNames Array in EAT.

Function III:

```
1. pushad                    ; save the contents of registers to the stack
2. xor edx, edx              ; clear edx. It will store index number of API in the AddressOfNames array
3. push esi                  ; push the name of the function to the stack
4. cmp byte ptr [esi], 0     ; check whether the current byte is null or not
5. je short 004020f2         ; if the byte is 0 then exit the function
6. cmps byte ptr [esi], byte ptr es:[edi]    ; compare the strings pointed to by esi and edi, byte by byte
7. je short 004020e0         ; if the byte matches then compare the next byte
8. inc edx                   ; increment the counter
9. xor eax, eax              ; clear the eax register which is used to check for string termination
a. dec edi                   ; set the edi register to point to the API name
b. scas byte ptr es:[edi]    ; compare the byte pointed to by edi with al register. in other words,
compare the current char with null byte
c. jnz short 004020ec        ; keep checking till null byte is reached. at the end of this loop, we will
have the next API name pointer in the edi register
d. pop esi                   ; store the function name in the esi register
e. jmp short 004020df        ; check with the next function name in the AddressOfNames array
```

The above set of instructions will walk through the AddressOfNames array of kernel32.dll's Export Directory till it locates the function API whose address we want to find. In our case, GetSystemDirectoryA function as an example.

At the end of the above function, edx register will have the index of the Function API whose address we are trying to find.

Function IV:

```
1. inc esi            ; point esi to the next function name whose address we have to fetch
2. mov [esp+8], esi   ; store this function name on the stack
3. mov [esp+20], edx  ; store the index number of function API on the stack
4. pop eax            ; eax is set to function API name
5. popad              ; restore all the registers using the stack contents
6. ret
```

At the end of the above set of instructions, we have the following stack layout.

eax: index number of function API in the AddressOfNames array
ecx: virtual address of kernel32.dll's export address table
edx:
ebx: base address of kernel32.dll
esp: stack pointer
ebp: base pointer
esi: pointer to next function name whose address we have to find
edi: pointer to the first function name in kernel32.dll's EAT's AddressOfNames array

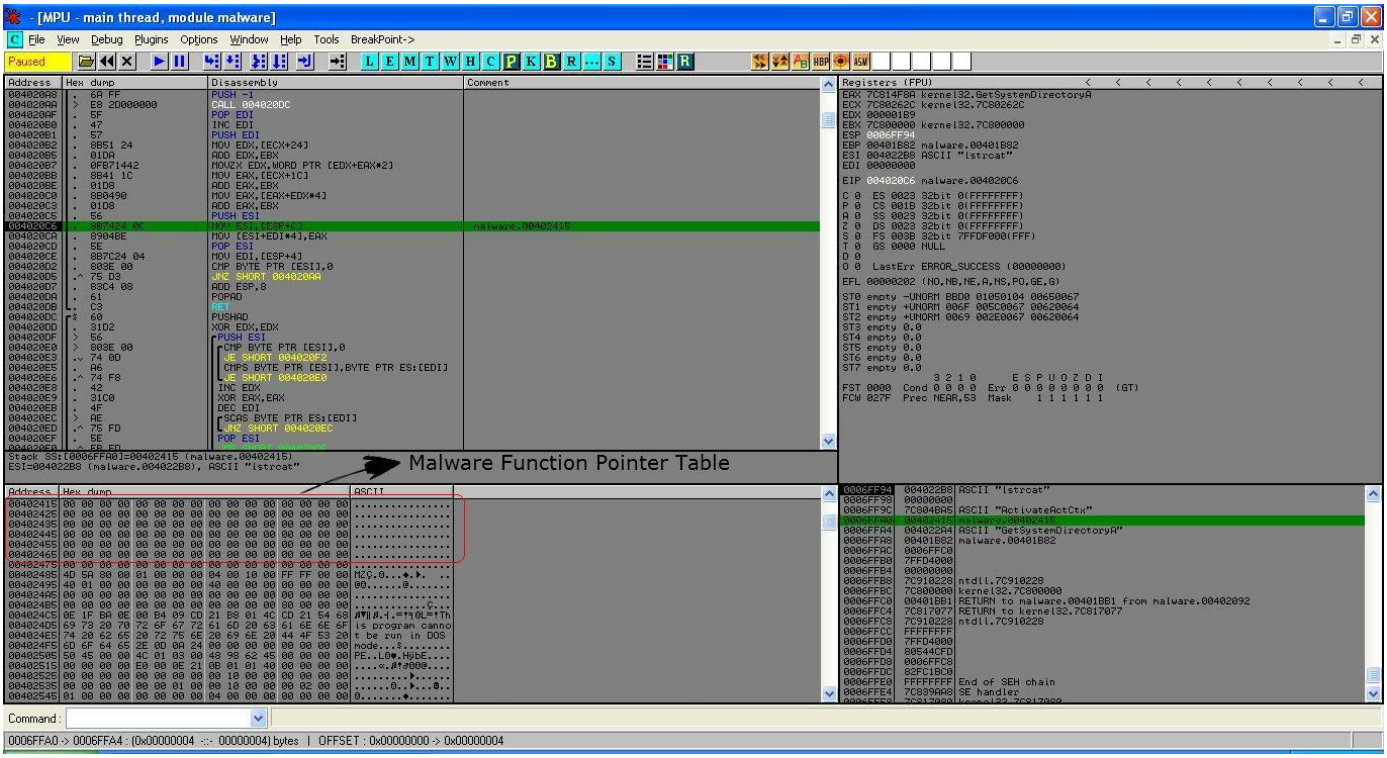
This stack layout remains the same for all the Function API Address Resolutions.

Function V:

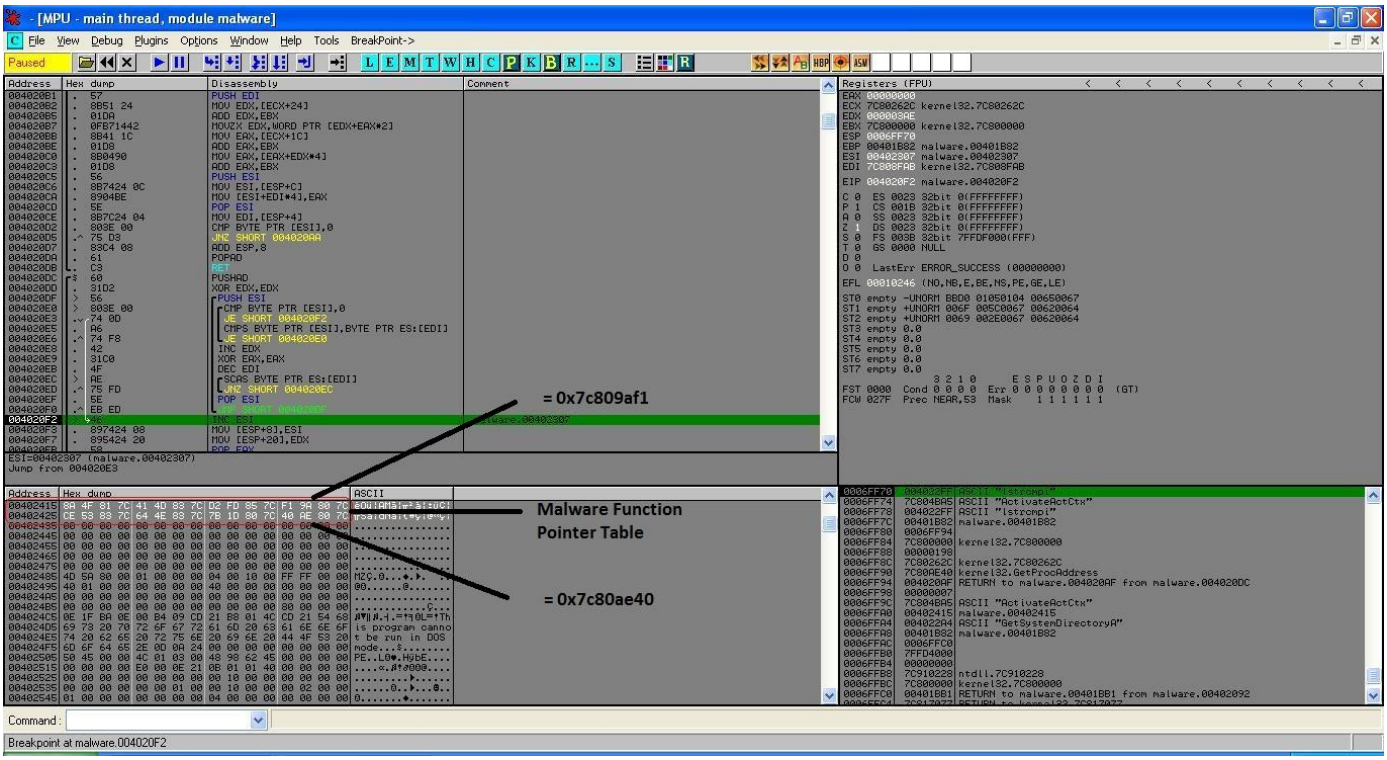
```
1. pop edi                   ; store FFFFFFFF in the edi register
2. inc edi                   ; set edi to 0 and therefore initialize the counter
3. push edi                  ; save it to stack
4. mov edx, [ecx+24]         ; store the RVA of AddressOfOrdinals array of Export Directory. It is at
offset, 0x24 from the base address of Export Directory
```

```
5. add edx, ebx              ; add the base address of kernel32.dll to this RVA to get the Virtual Address
of AddressOfOrdinals Array
6. movzx edx, word ptr [edx+eax*2]    ; using index of function name from AddressOfNames array, we get the
corresponding ordinal from AddressOfOrdinals array. store                                    in edx
7. mov eax, [ecx+1c]         ; store the RVA of AddressOfFunctions array of Export Directory which is at
offset 0x1c from base of EAT in eax.
8. add eax, ebx              ; add the base address of kernel32.dll to eax to get the VA of
AddressOfFunctions array
9. mov eax, [eax+edx*4]      ; using the ordinal of function api from AddressOfOrdinals array, we get the
RVA from AddressOfFunctions array and store in eax
a. add eax, ebx              ; add base address of kernel32.dll to the above RVA to get the Virtual
Address of Function API
b. push esi                  ; push the next function name whose address we have to find to the stack
c. mov esi, [esp+c]          ; get the pointer to the internal table where the malware stores the function
pointers to different APIs
d. mov [esi+edi*4], eax      ; store the function API virtual address in the malware's function pointer
table
e. pop esi                   ; store the next function API's name back in esi
f. mov edi, [esp+4]          ; point edi to the first function name in the AddressOfNames array
10. cmp byte ptr [esi],0     ; check whether esi is pointing to null byte or not
11. jnz short 004020aa       ; if esi is pointing to a function name then repeat the entire process to
find its virtual address
```

As you can see, the rootkit maintains a table of function Pointers at the memory address, 00402415, which initially looks like this:



After running through the above loop multiple times, this table is filled with the function pointers to all the APIs which the rootkit will be using:



Now, we will understand the above set of instructions much better using Windbg.

I will take AddConsoleAliasA as an example. By Walking through the above functions, I will highlight how the assembly language code is working.

I will not go into the details of Function I, since it is already covered in previous topic of Process Environment Block.

In Function II,

ebx = kernel32.dll base address

I will check the value at offset 3c from the base.

```
0:001> dd 7c800000+3c
7c80003c  000000f0 0eba1f0e cd09b400 4c01b821
```

So, the value is f0.

This is the offset of the PE File Header from the base address of kernel32.dll

Next, I will read the PE File Header by adding this offset to the base address,

```
0:001> dc 7c800000+f0
7c8000f0  00004550 0004014c 49c4f2bb 00000000  PE..L......I....
```

As can be seen, we have the PE File Header at the offset f0 from the base address of kernel32.dll. This marks the beginning of PE File Header.

Let's view this header now,

```
0:001> !dh kernel32 -f

File Type: DLL
FILE HEADER VALUES
     14C machine (i386)
       4 number of sections
49C4F2BB time date stamp Sat Mar 21 19:29:23 2009

       0 file pointer to symbol table
       0 number of symbols
      E0 size of optional header
    210E characteristics
            Executable
            Line numbers stripped
            Symbols stripped
            32 bit word machine
            DLL

OPTIONAL HEADER VALUES
     10B magic #
    7.10 linker version
   83A00 size of code
   70400 size of initialized data
       0 size of uninitialized data
    B64E address of entry point
    1000 base of code
         ----- new -----
7c800000 image base
    1000 section alignment
     200 file alignment
       3 subsystem (Windows CUI)
    5.01 operating system version
    5.01 image version
    4.00 subsystem version
   F6000 size of image
     400 size of headers
   F8F85 checksum
00040000 size of stack reserve
00001000 size of stack commit
00100000 size of heap reserve
00001000 size of heap commit
       0  DLL characteristics
    262C [    6D19] address [size] of Export Directory
   81EB4 [      28] address [size] of Import Directory
   8A000 [   65EE8] address [size] of Resource Directory
       0 [       0] address [size] of Exception Directory
       0 [       0] address [size] of Security Directory
   F0000 [    5CA0] address [size] of Base Relocation Directory
   847A4 [      38] address [size] of Debug Directory
       0 [       0] address [size] of Description Directory
       0 [       0] address [size] of Special Directory
       0 [       0] address [size] of Thread Storage Directory
   4E600 [      40] address [size] of Load Configuration Directory
       0 [       0] address [size] of Bound Import Directory
    1000 [     624] address [size] of Import Address Table Directory
       0 [       0] address [size] of Delay Import Directory
       0 [       0] address [size] of COR20 Header Directory
       0 [       0] address [size] of Reserved Directory
```

From this we can see that the Export Directory is at RVA, 262C from the base address.

In the Function II above, we are reaching the Export Directory by reading address at offset, 0x78 from the PE File Header.

So, let's check that,

```
0:001> dd 7c8000f0+78
7c800168  0000262c 00006d19 00081eb4 00000028
7c800178  0008a000 00065ee8 00000000 00000000
```

As can be seen, the value at offset 0x78 is, 262C which is the RVA of Export Directory.

Let's dump the export directory.

```
0:001> dd 7c800000+0000262c
7c80262c  00000000 49c4ce3f 00000000 00004b98
7c80263c  00000001 000003ba 000003ba 00002654
7c80264c  0000353c 00004424 0000a6e4 000354ed
```

We saw in Export Directory topic that how we can traverse this data structure and compare it with the IMAGE_EXPORT_DIRECTORY structure to get the RVAs of the 3 arrays.

In the function II above, we have read the RVA at offset 0x20 from the Export Directory Base.

```
0:001> dd 7c800000+0000262c+20
7c80264c  0000353c 00004424 0000a6e4 000354ed
7c80265c  000326c1 0007241f 000723e1 000593fa
7c80266c  000592de 0002bf11 00009011 00072a71
```

The value is 353c and this is the RVA of AddressOfNames array.

Let's view the contents of this array,

```
0:001> dd 7c800000+0000353c
7c80353c  00004ba5 00004bb4 00004bbd 00004bc6
7c80354c  00004bd7 00004be8 00004c07 00004c26
7c80355c  00004c33 00004c4f 00004c5c 00004c76
```

In instruction 8,9 and 10, we are moving the RVA of first function name, adding base address of kernel32.dll to it and then pushing the string found at that address to the stack.

Let's view this string,

```
0:001> da 7c800000+00004ba5
7c804ba5  "ActivateActCtx"
```

This is the first function name in the array.

Now, let's analyze Function III,

In function III, we traverse the **AddressOfNames** array, take each RVA and add it to the base address of kernel32.dll. The Function Name at that address is compared with AddConsoleAliasA. This loop is repeated till the matching RVA is found. A counter is incremented everytime the loop is executed and this is stored in edx.

So, after running this loop and searching for AddConsoleAliasA function, we get the value in edx as 3.

Let's analyze function V,

here ecx register will have the Export Directory Table base address, 7c80262c

At offset 0x24 from this address we have the RVA of AddressOfOrdinals array.

let's check it,

```
0:001> dd 7c80262c+24
7c802650  00004424 0000a6e4 000354ed 000326c1
7c802660  0007241f 000723e1 000593fa 000592de
7c802670  0002bf11 00009011 00072a71 0005fcb4
```

So, base address of the array is, 7c804424

In instruction 6, we use the index from AddressOfNames array and read the corresponding word from the AddressOfOrdinals array. This gives us, 0x03.

In instruction 7, I read the RVA of AddressOfFunctions array which is at offset 0x1c from the Export Directory's base address as shown below,

```
0:001> dd 7c80262c+1c
7c802648  00002654 0000353c 00004424 0000a6e4
7c802658  000354ed 000326c1 0007241f 000723e1
7c802668  000593fa 000592de 0002bf11 00009011
```

So, the base address of AddressOfFunctions Array is: 7c802654

Let's view this array,

```
0:001> dd 7c802654
7c802654  0000a6e4 000354ed 000326c1 0007241f
7c802664  000723e1 000593fa 000592de 0002bf11
7c802674  00009011 00072a71 0005fcb4 0003594f
```

We use the AddressOfOrdinals index and parse this array in steps of 4 bytes till index 0x3. This gives us the RVA of **AddConsoleAliasA** function as 0007241f

So, the base address of AddConsoleAliasA is: 7c800000+0007241f = 7c87241f

Let us verify the function name at this memory address,

```
0:001> ln 7c87241f
(7c87241f)   kernel32!AddConsoleAliasA
```

In this way, we saw a realistic application of the method to find base address of a function API by analyzing the rootkit.

Conclusion

After reading the above document, you should be able to explore OS Internals using a debugger and apply this knowledge while reversing malwares.

This will also help in understanding more advanced concepts like reconstructing the Import Table of a packed executable. This document shall serve as a reference to the experienced reversers as well.

References

1. http://blog.harmonysecurity.com/2009_06_01_archive.html
2. http://www.offensivecomputing.net/
3. http://www.phreedom.org/solar/code/tinype/