Finding and Displaying Symbols

dt *!_IMAGE_DOS_HEADER: Finding a known symbol within an unknown module. Result is the module(s) that have that symbol.

dt ntdll!_IMAGE_DOS_HEADER kernelbase: Applying the type to the base address of the module starting at RVA 0x00. Result is the struct fields with values.

.formats 0n248: Take one base and show all formats of that number in other bases like hex: 0xf8.

? kernelbase + 0xf8: Adding the base address of a module with an RVA. Result is the Virtual Address.

dx -r1 (*((combase!_IMAGE_DATA_DIRECTORY(*)[16]) 0x<some_virtual_address>)): Using the DX command to create a C++ expression to display 16 entries of type _IMAGE_DATA_DIRECTORY starting at the given virtual address.

dx -r1 (*((combase!_IMAGE_DATA_DIRECTORY*) (0x<some_virtual_address> + 0x08))): At the start of the _IMAGE_DATA_DIRECTORY table, dump the data at offset 0x08. This will be the struct for imports. Offset 0x00 would be for the exports.

Important PE-Related Structures

_IMAGE_DOS_HEADER:
- The first struct found at RVA 0x00
- Important fields: *e_magic*, *e_lfanew*
- Identified by *e_magic* field value: **MZ**
- Represents the beginning of a PE image, the infamous MZ signature is often searched for in memory to find PE images

_IMAGE_NT_HEADERS64:
- Pointed to by *e_lfanew* RVA
- Important fields: *Signature, FileHeader*, *OptionalHeader*
- Identified by *Signature* field value: **PE**

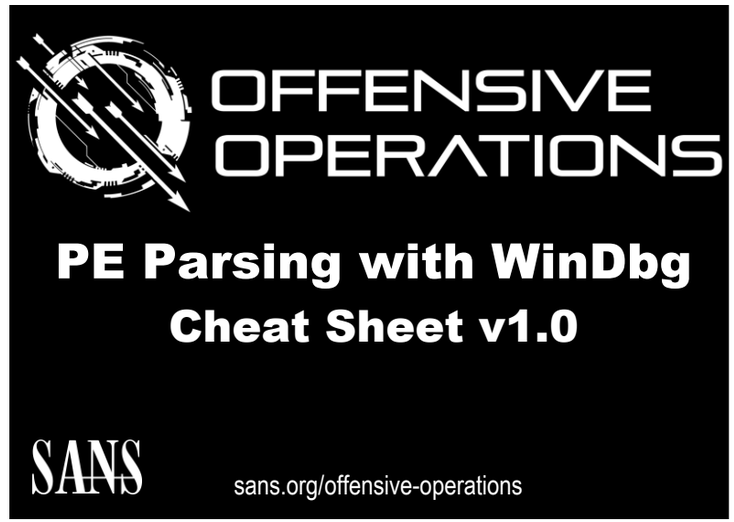
_IMAGE_FILE_HEADER:
- Contained within _IMAGE_NT_HEADERS64
- Important fields: *NumberOfSections*, *SizeOfOptionalHeader*
- The *NumberOfSections* is useful when iterating over all sections
- The *SizeOfOptionalHeader* is useful to jump to the first section

_IMAGE_OPTIONAL_HEADER64:
- contained within _IMAGE_NT_HEADERS64
- Important fields: *AddressOfEntryPoint*, *ImageBase*, *DataDirectory*
- Identified by *Magic* field value: **0x20b**. Don't be fooled by *AddressOfEntryPoint*, it is simply an RVA that must be added to the module's base address.

_IMAGE_DATA_DIRECTORY:
- Contained within _IMAGE_OPTIONAL_HEADER64
- Important fields: *VirtualAddress*, *Size*
- This holds an array of 16 entries (arrays) that store information about imports, exports, debug info, etc. Don't be fooled by *VirtualAddress*, it is simply an RVA that must be added to the module's base address.

_IMAGE_EXPORT_DESCRIPTOR:
- Found in the 1st entry in the _IMAGE_DATA_DIRECTORY (index 0).
- Important fields: *Name*, *Base, NumberOfFunctions, NumberOfNames, AddressOfFunctions, AddressOfNames, AddressOfNameOrdinals*
- Represents the exports for the image, if any

_IMAGE_IMPORT_DESCRIPTOR:
- Found in the 2nd entry in _IMAGE_DATA_DIRECTORY (index 1).
- Important fields; *OriginalFirstThunk*, *Name*, *FirstThunk*
- Represents the imports for the image



Manually parsing PE images in WinDbg lays the foundation to parsing them programmatically using C++, which is covered in SEC670: Red Teaming Tools

Getting Started

To get started, WinDbg must be attached to a native 64-bit target process like notepad.exe. You can either "Launch executable" or "Attach to process"; the choice is yours.

Basic Commands

!dh: displays all PE headers
lm v m: lists detailed module info
db/w/c: display hex & ASCII: BYTE/WORD/DWORD
dx: Natvis command, can handle C++ expressions
dt: display a symbol type
dl: display singly/doubly linked lists
ds/dS: display STRING/UNICODE_STRING structs
?: used to evaluate an expression

Parsing a module from its base address

Show the value of *e_lfanew* and only *e_lfanew* given the base address of a module like kernelbase.dll. Find the base address by running lm m kernelbase. Convert the output to hex, then add it to the base address of the module

> dx (*(ntdll!_IMAGE_DOS_HEADER *)0x7ffed31b0000).e_lfanew
> .formats 0n248 = 0xf8
> ? kernelbase + 0xf8 = 00007ffe`d31b00f8

> dt ntdll!_IMAGE_NT_HEADERS 00007ffe`d31b00f8
Symbol ntdll!_IMAGE_NT_HEADERS not found

Show the values of the NT Headers struct
> dx (*(combase!_IMAGE_NT_HEADERS*)0x7ffe`d31b00f8)

Use the dx command to grab the File and Optional headers with dot notation.
> dx &(*((combase!_IMAGE_NT_HEADERS*)0x7ffe`d31b00f8)).FileHeader

> dx &(*((combase!_IMAGE_NT_HEADERS*) 0x7ffe`d31b00f8)).OptionalHeader

From the *OptionalHeader*, show the data directory.
> dx -r1 &(*((combase!_IMAGE_DATA_DIRECTORY *) 0x00007ffe`d31b0180))
[+0x000] VirtualAddress : 0x344bc0
[+0x004] Size : 0xf9b0

Add the *VirtualAddress* RVA value to the base address to determine the true virtual address.
> ? kernelbase + 0x344bc0 = 0x00007ffe`3d424bc0

Pull out the RVA for the *AddressOfNames* then add it to the base address
> dx &(*(combase!PIMAGE_EXPORT_DIRECTORY) 0x00007ffe`3d424bc0).AddressOfNames
> ? kernelbase + 0x346af8 = 00007ffe`3d426af8

Dump the data found here and limit output to 1
> dc 00007ffe`3d426af8 l1 = 003499b7

Take the RVA from previous command, add it to base address
> ? kernelbase + 0x003499b7 = 00007ffe`3d4299b7

Display ASCII text using that address and we have the first exported function
> da 00007ffe`3d4299b7 = "AccessCheck"

Walking the PEB for DLL Hashes

Dump the PEB using the pseudo register *@$peb*, which is the pointer to the PEB for the process you are debugging.
dt nt!_PEB @$peb

Save the pointer to the loader data inside a variable. We can use arrow notation just like in C++.
dx @$ldr = @$peb->Ldr

Grab the first link from the *InMemoryOrderModuleList* and save it off.
dx @$headlist = @$ldr->InMemoryOrderModuleList.Flink

The DX command offers many features from the Collections library. One such feature is to walk a linked list using the FromListEntry found under Utility.Collections. The documentation can be shown by running the following command.
dx -v debugger.Utility.Collections.FromListEntry

To use this properly, there are three arguments that must be supplied: the LIST_ENTRY for the head of the list, the typename to cast against, and finally the name of the field itself from within the struct.

Since the head of the list has been saved in a local variable, it can be used to walk the linked list of *InMemoryOrderLinks* to see any values of interest. Dump the entire list from the start.
dx Debugger.Utility.Collections.FromListEntry(*(nt!_LIST_ENTRY*)@$headlist, "ntdll!_LDR_DATA_TABLE_ENTRY", "InMemoryOrderLinks")

Dump the entire list from the start, but this time display the results in grid form. The grid will have the name of the DLL and its hash value. The hash value is calculated by Windows when the system loader maps it into memory. This is part of a special linked list sometimes called hash links.
dx -g Debugger.Utility.Collections.FromListEntry(*(nt!_LIST_ENTRY*)@$headlist, "ntdll!_LDR_DATA_TABLE_ENTRY", "InMemoryOrderLinks").Select(e => new { Module = e->BaseDllName, HashValue = e.BaseNameHashValue })

From here, a few tweaks could be made to display the base address of the modules. Once those are in hand, manual PE parsing can begin as previously shown in the other column.

Useful Commands

dtx: the perfect combo of dt and dx
dtx nt!_PEB @$peb

sxe: set an exception for an event
sxe ld mpclient.dll

k: dump the callstack

.echo: print strings
.echo "WinDbg rocks!"

.printf: print a formatted string
.printf "peb %ma\n", @$peb

Useful Extensions

!dlls: show table entries of loaded modules

!dlls -i/l/m: show init, load, memory order of modules

!dlls -c <some_dll_address>: will show the module where the specified address is found

!analyze -v: analyze a crash dump

!peb/teb/heap: show the PEB/TEB/Heap of the current process

!exchain: show the SEH chain