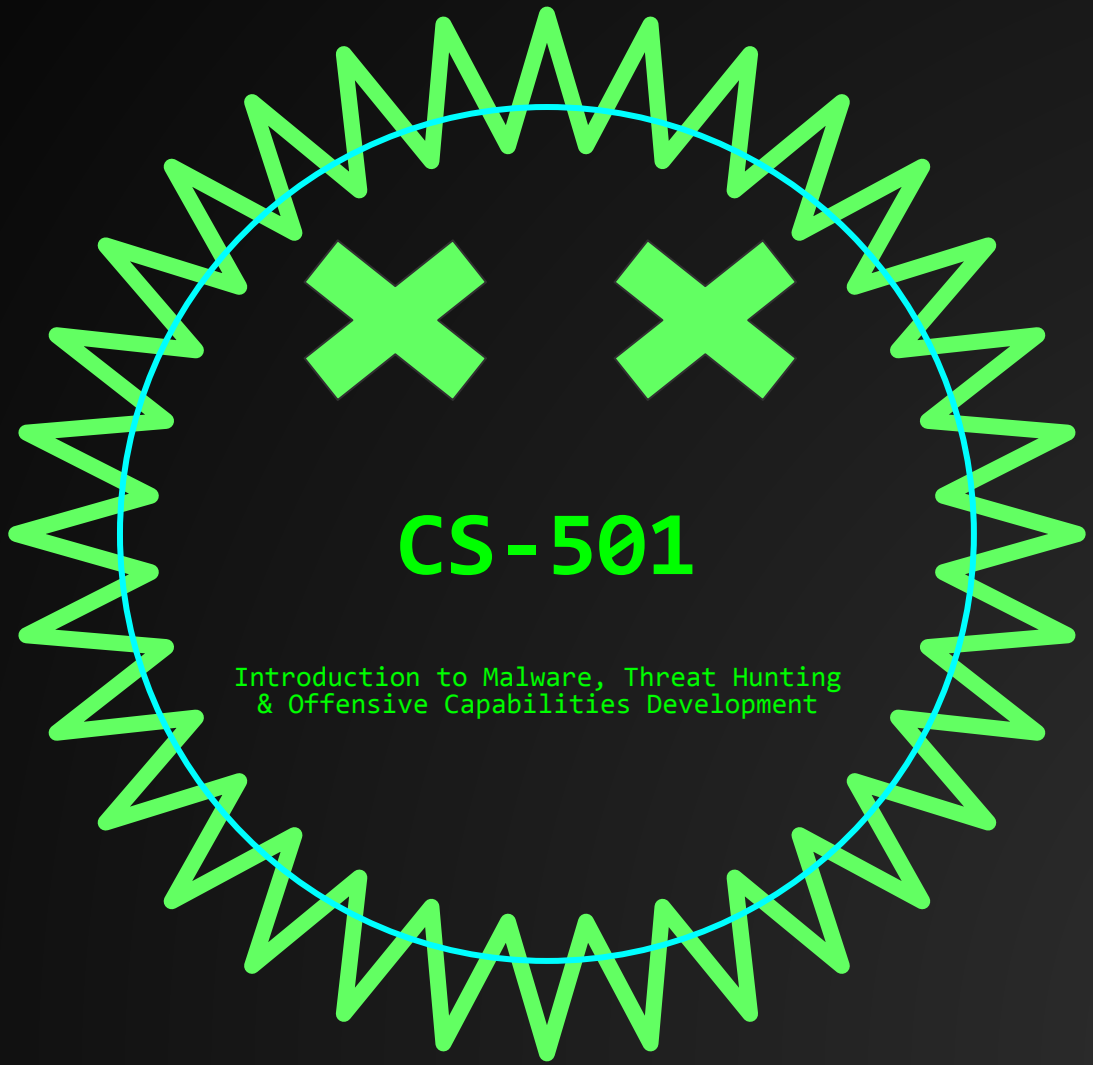


CS-501

Introduction to Malware, Threat Hunting
& Offensive Capabilities Development

Lecture 0x02:
Basic Windows Internals

Agenda for Today: Windows Basics

- Core OS Concepts
- Windows Internals

The Windows Operating System

What is an Operating System

A fancy resource manager.

It abstracts away the managing of physical devices (CPU, Memory, I/O devices…etc)

Operating System Kernel

Responsible for servicing requests and translating them into instructions compatible with computer hardware including the CPU, Memory, I/O devices…etc

Often provides an interface for userland applications to interact indirectly with resources

The Windows Operating System

- Proprietary Operating System Developed by Microsoft
- Multi Arch: 32/64 bit, preemptive multitasking operating system
- Primary flavors are Client, Server
  - Others include IOT, CE, Xbox, Some defunct targeting mobile…etc

We will focus on Windows (10) Client/Server

Devices that run Windows

- Desktop computers
- Servers
- Laptops
- Tablets
- XBox (technically a fork of Windows 2000)

Different devices have different considerations (battery, screen size, memory, storage …etc)

(Some) Windows Design Principles

- Extensibility
- Portability
- Reliability
- Performance
- Compatibility
- Security
- $$$$$

Extensibility

- The bulk of system services are provided by the Executive
- The Windows OS is technically a monolithic OS
  - I.e., One driver can mess with another driver. Kernel memory is shared
- However, it separates userland execution environments into “subsystems”
- Each subsystem has the ability to execute a type of application
- This provides a modular setup where changes to execution environments don’t necessarily necesitate changes to the executive

Portability

- Windows will run on a wide variety of systems.
- The bulk of the OS is written in C and C++
  - The rest is bundles of code used for processor specific instructions (Eg Arm vs intel/amd)
- Platform dependent code is implemented in the Hardware Abstraction Layer (HAL)

This is less magical than it was 10 years ago as most modern OSes support variety in hardware/architecture

Reliability

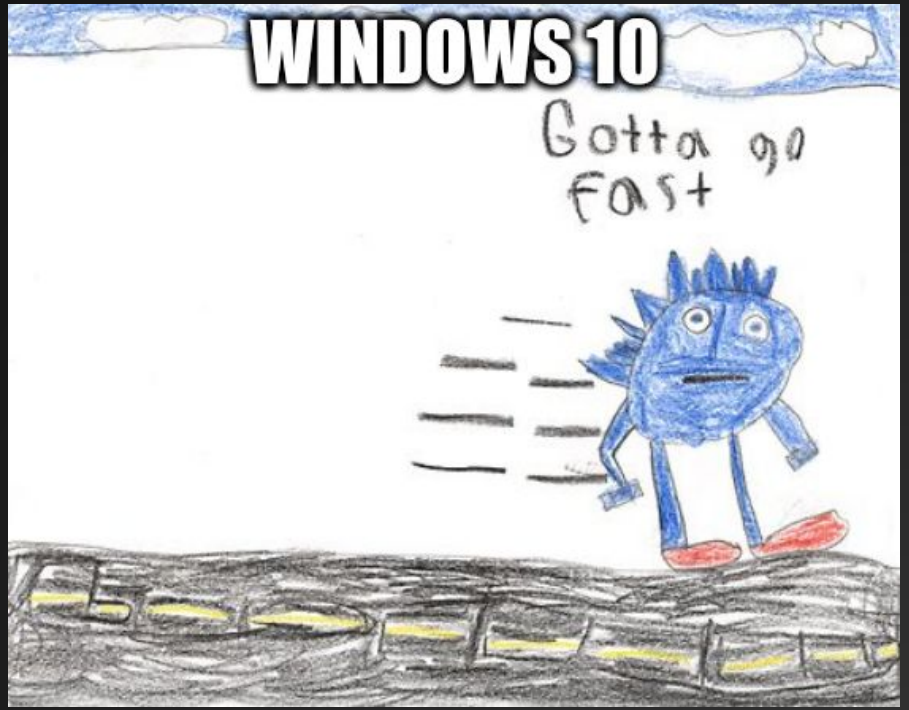
- User applications should not be able to corrupt the operating system

Windows audits most drivers and has rigorous standards for which drivers are officially signed

That said, malicious drivers can muck around with kernel memory. There are countermeasures for this. (eg Driver Guard)

Performance

- Should be fast.



Compatibility

Backwards compatibility and extending of existing technology

This is partially why the Windows API is so stable

Security

- Kernel/User separation
- User separation
- Process (and associated resources) isolation

Windows Kernel

- Windows is an Object based kernel
- Resources are called objects, and can only be directly accessed by the kernel
- To indirectly interact with an object, a *handle* is used

Kernel Objects

- Kernel object (KOs): a single run-time instance of a statically defined object type
- Object types are system-defined data types.
- Each object type has its own attributes and functions to interact with it
- For example, an object of type *process* is an instance of a process object.
- A *file* object is an instance of a file. Note, *file!= a thing on disk*

Objects and Handles

A handle is an abstract reference to an object. This could be an actual pointer to the object, or a reference to a per-process GUID that references an object

This allows us to abstract away direct management of objects in memory, and instead work with with references. **This is a security control. If something goes wrong in kernel space, you get a BSOD.**

APIs are used to interact with system resources, share resources among processes, and protect resources from unauthorized access.

Windows System Architecture

- User Processes
- Subsystem DLLs
- NTDLL.dll
- Service Processes
- Executive
- Kernel
- Device Drivers
- Win32k.sys
- (sometimes) Hyper-V

Interacting with the Windows OS

- Windows API functions (win32): Documented, callable functions in the Windows API.
  - For example, MessageBox, CreateFile, GetMessage
- Native system services (sys calls): Undocumented (officially) underlying serivices in the OS that are callable from user mdode. For example
  - NtAllocateVirtualMemory is the internal service used for VirtualAlloc
  - NtCreateUserProcess is the internal service used by CreateProcess
- Direct Syscalls
- Other: (WinRT, COM,...etc)
- Kernel support functions: functions inside the Windows OS that can only be called from in kernel mode

Windows Executables and Shared Libraries

Executable File Formats

- Bundles of machine code and associated data needed to run a program
- Usually requires an OS to *load* the executable
- Code is organized according to a convention that the programer and the kernel agree on.

It is just a convention though!

PE File Format Basic Definitions and Concepts

- Portable Executable (PE) is an executable file format used by Windows NT
- It contains information about code to execute, and how it should be executed
- In this discussion, we will use an open source tool PE-Bear to look at the structure of a PE file.

PE File Format

- PE file format is used for both userland and kernel mode executables
  - Userland: file.exe, file.dll, file.obj
  - Kernel mode: driver.sys, ntoskrnl.exe
- PE is based on the Common Object File Format (COFF).
- PE format is not architecture specific (hence “portable”)
  - Note this means the format can be used across multiple different architectures. The target architecture is still specified inside of the PE though
- Data is grouped together in blocks called *sections*, identified by *headers*

PE: Libraries

- Windows shared libraries are called Dynamically Linked Libraries
- They are PE files with a special characteristic set
- They can export functions from their code
- Other PEs can load these libraries and access the exported code
- This allows for modular programs

We will talk more about this next lecture when we review linking.

PE

PEs are composed of sections and headers

Sections are data/code

Headers contain information about how to load the PE, and where the data is, and how to process it

Tools

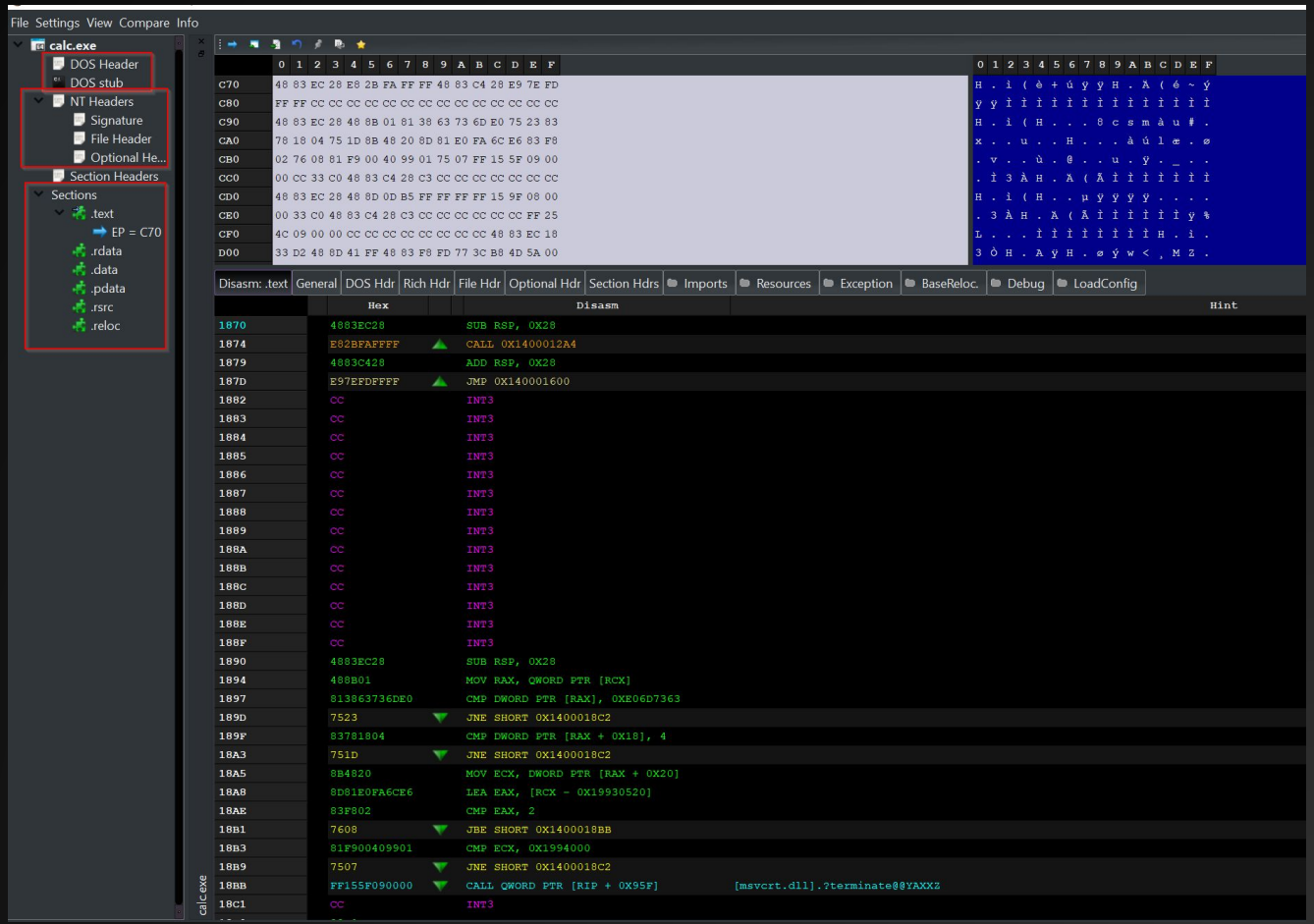
In this lecture, we will use PE-Bear to explore the PE file format.

As a sample, lets use Calc.exe (64 bit)

Run $path = Get-Command calc.exe to find the path to calc.exe on your machine

Run PE-Bear.exe $path.Source (in powershell)

Calc.exe

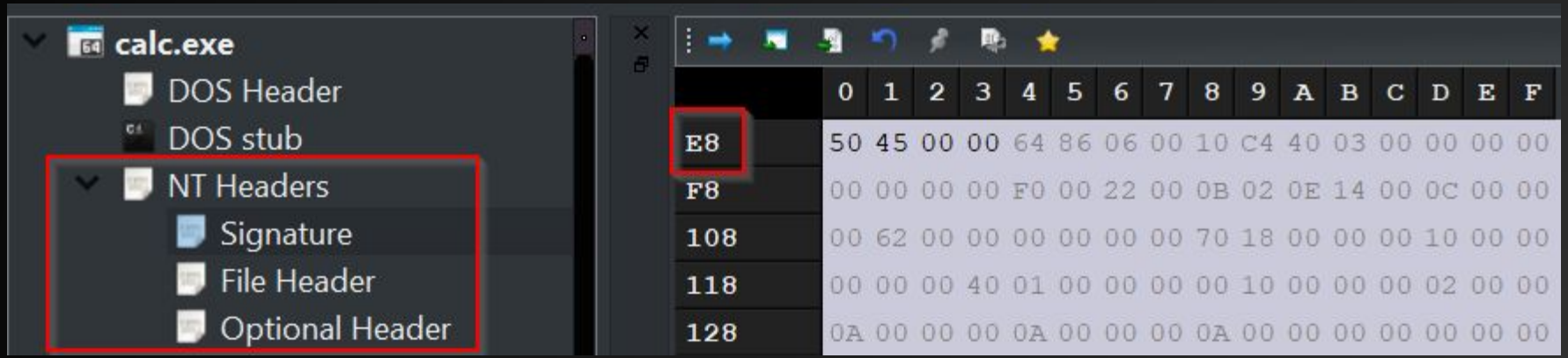

File Settings View Compare Info
calc.exe
DOS Header
DOS stub
NT Headers
Signature
File Header
Optional He...
Section Headers
Sections
.text
EP = C70
.rdata
.data
.pdata
.rsrc
.reloc
Disasm: .text
General
DOS Hdr
Rich Hdr
File Hdr
Optional Hdr
Section Hdrs
Imports
Resources
Exception
BaseReloc.
Debug
LoadConfig
Hex
Disasm
Hint
1870 4883EC28 SUB RSP, 0X28
1874 E82BFAFFFF CALL 0X1400012A4
1879 4883C428 ADD RSP, 0X28
187D E97EFDFFFF JMP 0X140001600
1882 CC INT3
1883 CC INT3
1884 CC INT3
1885 CC INT3
1886 CC INT3
1887 CC INT3
1888 CC INT3
1889 CC INT3
188A CC INT3
188B CC INT3
188C CC INT3
188D CC INT3
188E CC INT3
188F CC INT3
1890 4883EC28 SUB RSP, 0X28
1894 488B01 MOV RAX, QWORD PTR [RCX]
1897 813863736DE0 CMP DWORD PTR [RAX], 0XE06D7363
189D 7523 JNE SHORT 0X1400018C2
189F 83781804 CMP DWORD PTR [RAX + 0X18], 4
18A3 751D JNE SHORT 0X1400018C2
18A5 8B4820 MOV ECX, DWORD PTR [RAX + 0X20]
18A8 8D81E0FA6CE6 LEA EAX, [RCX - 0X19930520]
18AE 83F802 CMP EAX, 2
18B1 7608 JBE SHORT 0X1400018BB
18B3 81F9004099 01 CMP ECX, 0X1994000
18B9 7507 JNE SHORT 0X1400018C2
18BB FF155F090000 CALL QWORD PTR [RIP + 0X95F] [msvcrt.dll].?terminate@@YAXXZ
18C1 CC INT3
calc.exe

DOS Header

- The DOS header contains the magic bytes MZ that identify it as a PE
- The final entry (offset 0x3c referenced as ->e_lfanew) is the offset the of NT Headers

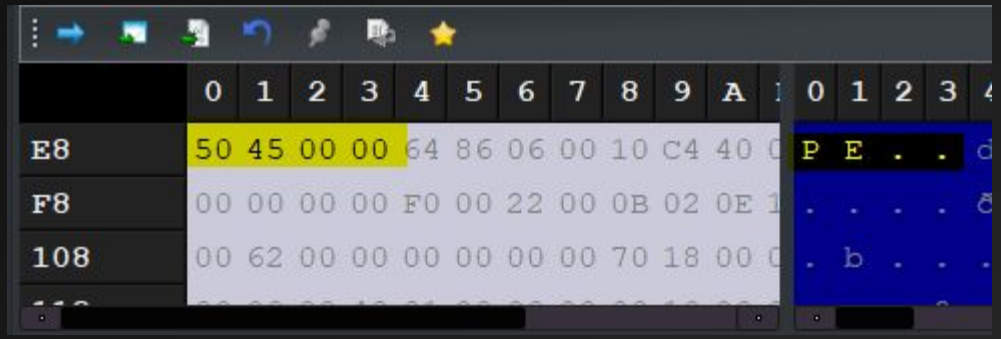
NT Headers

- Signatures
- File Header
- Optional Header



Signature

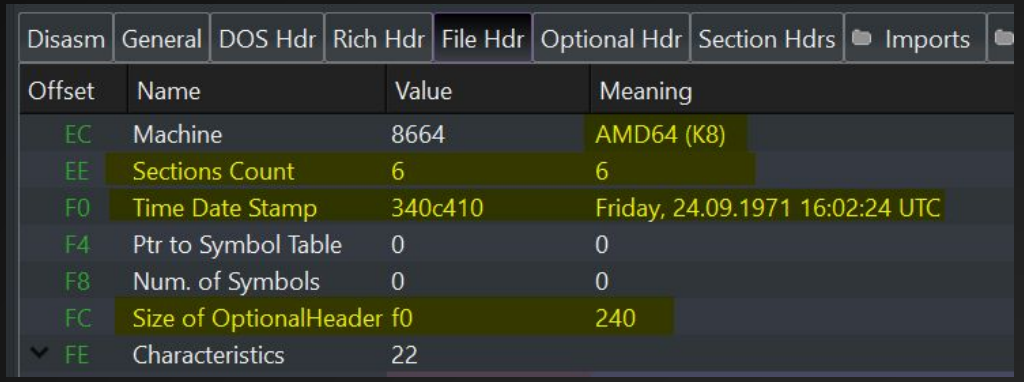
- Usually 4 bytes containing
- “PE\0\0”
- For our purposes, it is only used to verify the file format.



File Headers

Following the Signature, we have the File Headers. This gives us

- The number of sections (NumberOfSections)
- Whether or not we have a DLL/EXE (Characteristics)
- The Compilation timestamp
- A pointer to a symbol table if one exists

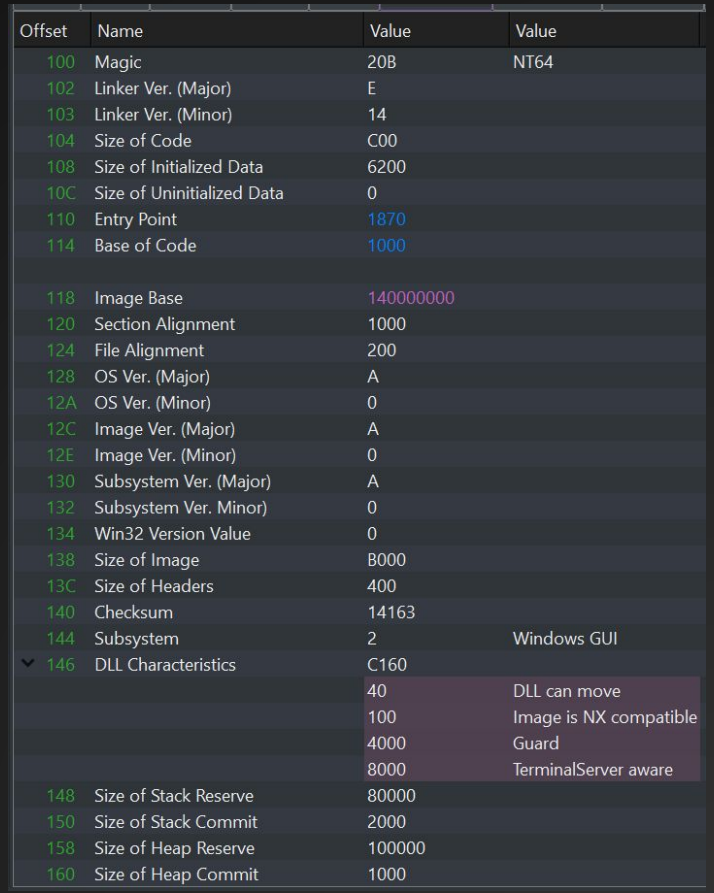


Optional Headers

I don’t know why it is listed as optional. I don’t think a PE can run without this section (but I could be wrong?)

The optional headers contains most of the data required to load PE

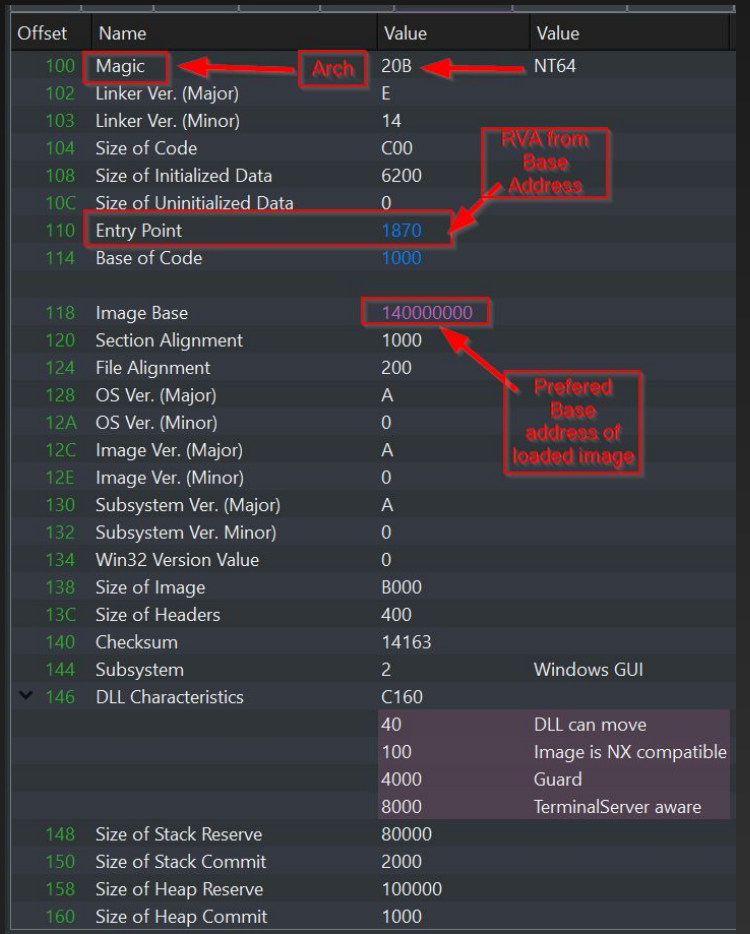
Specifically, values found here are used to build the *Import Address Table*, and perform *Base Relocations*



Offset	Name	Value	Value
100	Magic	20B	NT64
102	Linker Ver. (Major)	E	
103	Linker Ver. (Minor)	14	
104	Size of Code	C00	
108	Size of Initialized Data	6200	
10C	Size of Uninitialized Data	0	
110	Entry Point	1870	
114	Base of Code	1000	
118	Image Base	140000000	
120	Section Alignment	1000	
124	File Alignment	200	
128	OS Ver. (Major)	A	
12A	OS Ver. (Minor)	0	
12C	Image Ver. (Major)	A	
12E	Image Ver. (Minor)	0	
130	Subsystem Ver. (Major)	A	
132	Subsystem Ver. Minor)	0	
134	Win32 Version Value	0	
138	Size of Image	B000	
13C	Size of Headers	400	
140	Checksum	14163	
144	Subsystem	2	Windows GUI
146	DLL Characteristics	C160	
		40	DLL can move
		100	Image is NX compatible
		4000	Guard
		8000	TerminalServer aware
148	Size of Stack Reserve	80000	
150	Size of Stack Commit	2000	
158	Size of Heap Reserve	100000	
160	Size of Heap Commit	1000	

Optional Headers (pt 1)

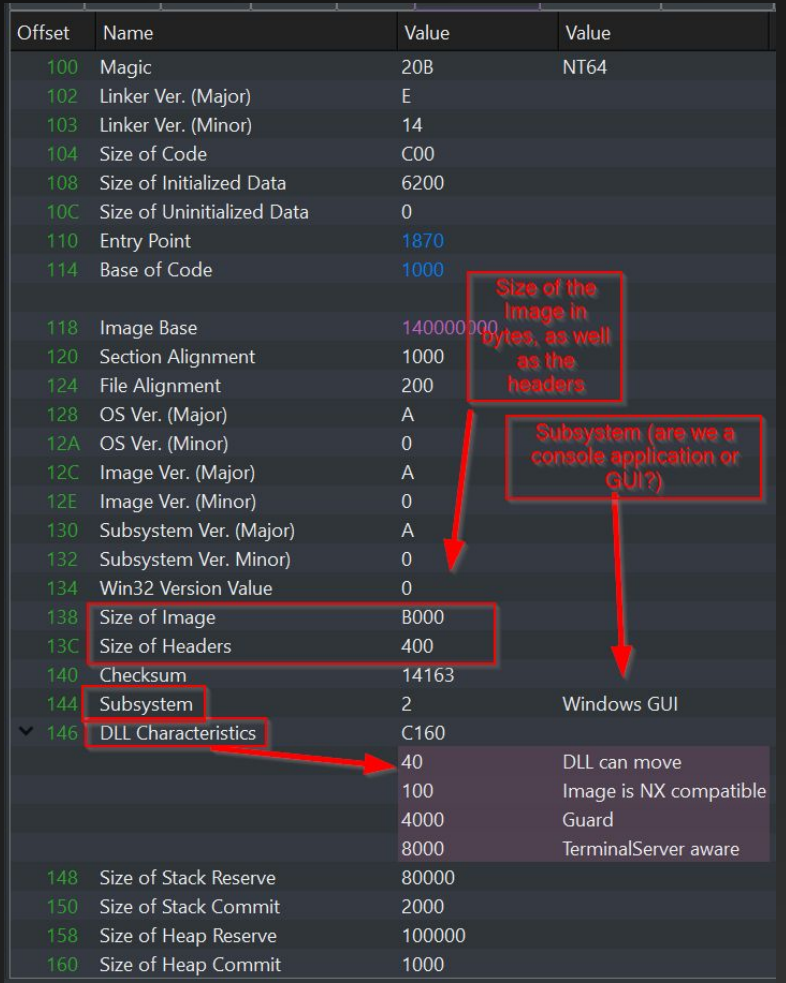
- Magic: Architecture of image
- Entry Point: Relative virtual address (RVA) from the Base Address
- Image Base: (prefered) Base address: Where in memory the PE “prefers” to be loaded. If the location is unavailable, the Image needs to be relocated



Offset	Name	Value	Value
100	Magic	20B	NT64
102	Linker Ver. (Major)	E	
103	Linker Ver. (Minor)	14	
104	Size of Code	C00	
108	Size of Initialized Data	6200	
10C	Size of Uninitialized Data	0	
110	Entry Point	1870	
114	Base of Code	1000	
118	Image Base	140000000	
120	Section Alignment	1000	
124	File Alignment	200	
128	OS Ver. (Major)	A	
12A	OS Ver. (Minor)	0	
12C	Image Ver. (Major)	A	
12E	Image Ver. (Minor)	0	
130	Subsystem Ver. (Major)	A	
132	Subsystem Ver. Minor)	0	
134	Win32 Version Value	0	
138	Size of Image	B000	
13C	Size of Headers	400	
140	Checksum	14163	
144	Subsystem	2	Windows GUI
146	DLL Characteristics	C160	
		40	DLL can move
		100	Image is NX compatible
		4000	Guard
		8000	TerminalServer aware
148	Size of Stack Reserve	80000	
150	Size of Stack Commit	2000	
158	Size of Heap Reserve	100000	
160	Size of Heap Commit	1000	

Optional Headers (pt 2)

- SizeOfImage: the virtual size of the image
- SizeOfHeaders: the size of the headers
- DLLCharacteristics: flags including knowledge of hardening features such as ASLR/ CFG...etc. Not super important for us other than assuming knowledge of ASLR.



Sections

.text Executable code (machine instructions)

.data: global variables

.rdata: read only global variables

Dynamically Linked Libraries:

- Refresh: What is a DLL?

DLL

A PE with with DLL characteristic field set.

Usually it has exported functions which can be referenced by code outside of the DLL

The Windows API is implemented in a handful of DLLs that export specific functions

Win32 API

We will mostly leverage documented functions from the Windows API

The function definitions are well documented

Reading that documentation however, is a skill that must be learned

Sometimes, we need more control over what we are trying to accomplish, and will leverage undocumented functions stored in NTDLL.dll

NtDLL.dll

Implements the Windows Native API. This is the lowest layer of code that is still Userland code.

It is used to communicate with the kernel for system call invocation.

NtDLL also implements the Heap Manager, the (executable) Image loader and some of userland thread pools. Every process loads this DLL in the same location in memory!

Kernel32.dll

Contains (more or less) the same functionality as NtDLL!

It exposes basic operations such as memory management, input/output (I/O) operations, process and thread creation, and synchronization functions

It can be thought of as a compatibility layer, as it almost always calls directly into NTDLL.dll

This is to maintain backwards compatibility– where the Win32 API rarely changes, but the Native API changes from release to release.

32bit vs 64bit

This class will focus on 64bit executables (Intel x86 64)

When developing code that needs to run on either a 32bit or 64bit systems, you need to take care when assuming the size of various types. For example, type sizes vary across 32bit and 64bit architectures and you need to take care when defining them.

Processes

Processes

Nothing in userland is executed outside of the context of a process.

You don’t “run a processes”

You run threads

Processes are containers, and there is no such thing (to my knowledge) as code running outside of a process

Threads

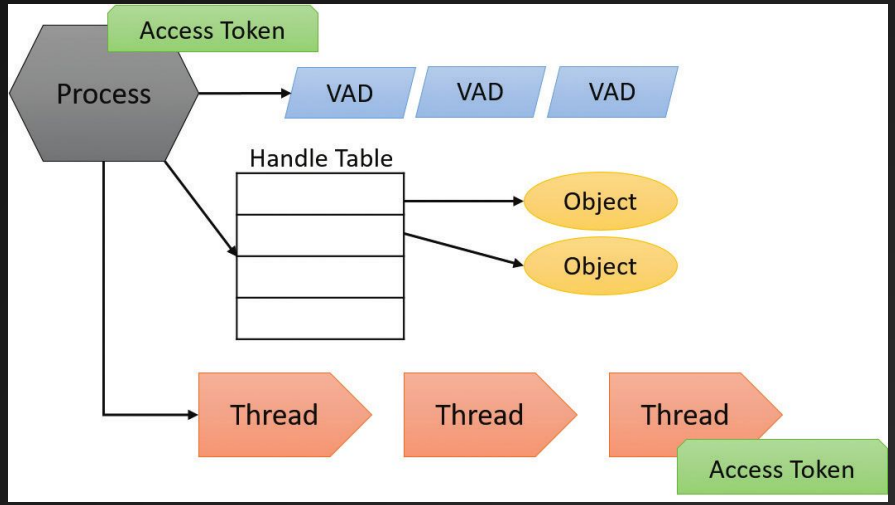
Unit of execution contained within a process

- Actual entity that executes code

More on threads in later. For now, we will only deal with processes that have a single executing thread.

Process Resources

- Uniquely identified by a Process ID
- Contains a security context in the form of a Process Access Token
- Maintains a table of handles to objects
- Has a *private* virtual address space
- >=1 Thread (possibly with its own token)



Virtual Memory

An abstraction layer around physical memory

Each process gets its own private virtual memory

2^32 bytes worth on 32 bit (possibly more with some extensions)

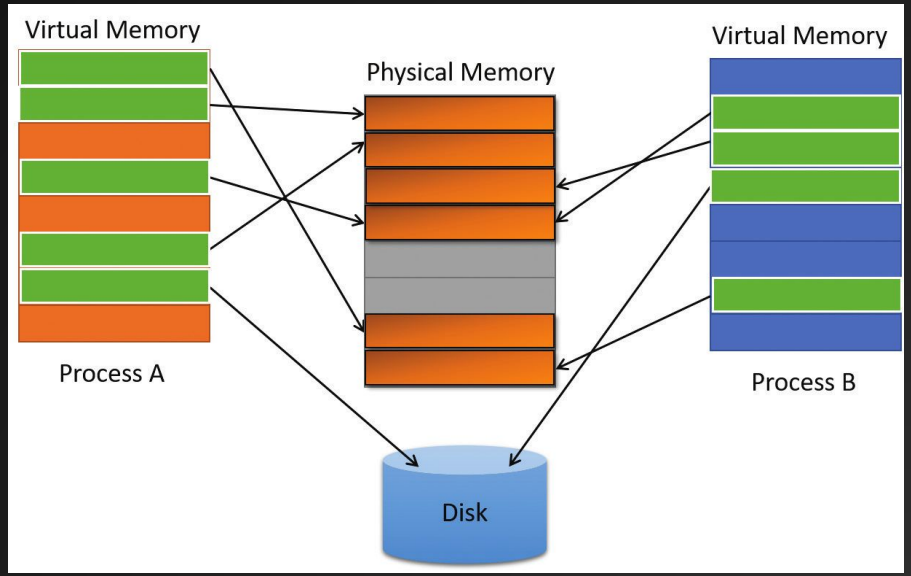
2^64 bytes worth on 64 bit

The kernel lies about how much space there really is

Think of a virtual address space as a giant, contiguous array of bytes

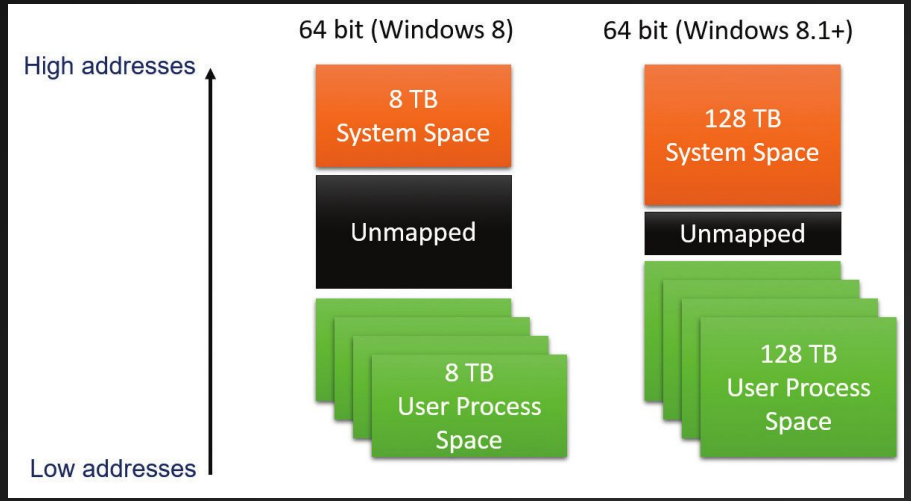
Process Virtual Memory

- Windows implements a linear Virtual Address space
- Provides a logical interface for a process to interact with indirectly with physical memory



Memory Layout

- Each processes gets its own *Virtual Address Space*
- The Windows OS divides Virtual Address space into two portions: kernel and Userland*
- Memory here grows “downward” from higher address spaces to lower address spaces
- On x64, the upper half is reserved for kernel space and the bottom half is reserved for processes



Memory

Virtual: A Block of raw memory

Stack: technically can be managed by the programmer but is usually managed by the processes and the compiler

Heap Memory: managed by the programer with a special data structure called a Heap (Basically a priority queue)

Stack: what is it good for?

- Local variables
- Passing arguments to a function
- Returning values from a function
- Function invocation/ABI

Stack Memory

- Data structure built on top of our Virtual Address Space that allows us to Push, and pop values from a stack
- Whenever we invoke a function with *call* a new *stack frame* is created. We call this the function call stack
- This is a contiguous chunk of memory that acts as a working space for a function’s duration

Stack Memory

- Stack memory is temporary: once the function returns the memory is reclaimed
- Still error prone but is typically safer and faster than Heap allocations
- A programer usually does not need to worry about managing stack memory
- We need to know how much is needed at compile time!

Heap Allocation

Allows for dynamic allocation sizes.

Recall we don’t need to know how much memory is needed at compile time

Heap memory is managed by a heap allocator.

The programmer allocates with a call to malloc and frees the memory with a call to free

If the programer forgets to call free, that memory is now unusable until the process terminates and we have introduced a *memory leak*

Heap Memory

- Memory that can be allocated/deallocated at runtime
- Used for data whose size is not known at compile time
- Slower than stack

Process Creation

Somewhat complicated. We will simplify it for this class, than dive into it next class after we talk about handles.

- Kernel opens the image (executable file) and verifies it is the correct format
- The kernel creates a new process kernel object and a thread kernel object
- The kernel maps the image to an address space, as well as ntdll.dll
  - Note this gets mapped to just about every type of process
- The creator process notifies Windows subsystem process (Csrss.exe) that a new process and thread have been created
- From the kernel's perspective, the process is created at this point
- Some magic happens, imports are resolved and after all the required LLs are declared, we reach the entry point and the program starts

Basic Information of a Process

- Name: Usually the executable name. This is NOT a unique identifier
- Process ID (PID: Unique ID of a process. PIDS are reused after a process terminates
- Status: Running, Suspended, Not Responding
- Username: the user who is running the process. It also includes the primary token that holds the security context for the user
- Session ID: Session number under which the process executes. Session 0 is for system processes and services. Session 1 and higher are used for interactive logins.