CS-501

Introduction to Malware, Threat Hunting
& Offensive Capabilities Development

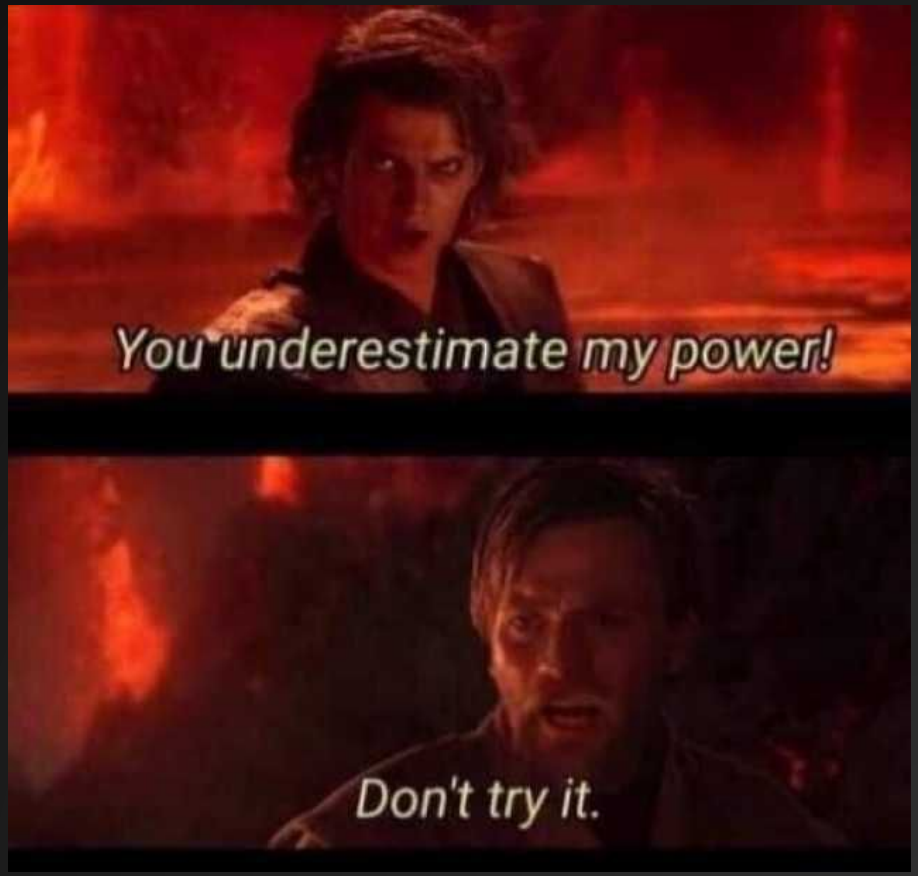
Lecture 6:
Intro to PE File Format

HW-2 is up!

- It is Due one week from today. You should get started on this ASAP. (Thursday, the 17th)
- In this assignment you will get exposed to CrackMes: puzzles used to practice your reverse engineering skills!
- The goal is simple: Find the input that makes the binary print “Cracked!”
- In this assignment you will get comfortable staring at some basic assembly, and finding the main function when a C-runtime is used.
- You don’t have to ever run the binary, but you can to verify your answer!

Get started soon!

- If you don’t have previous reverse engineering experience, you probably can’t do this homework the night before it is due!
- Get started ASAP: it is difficult to predict where you will get stuck!
- None of the CrackMes are designed to trick you per say, but they might not be straight forward!



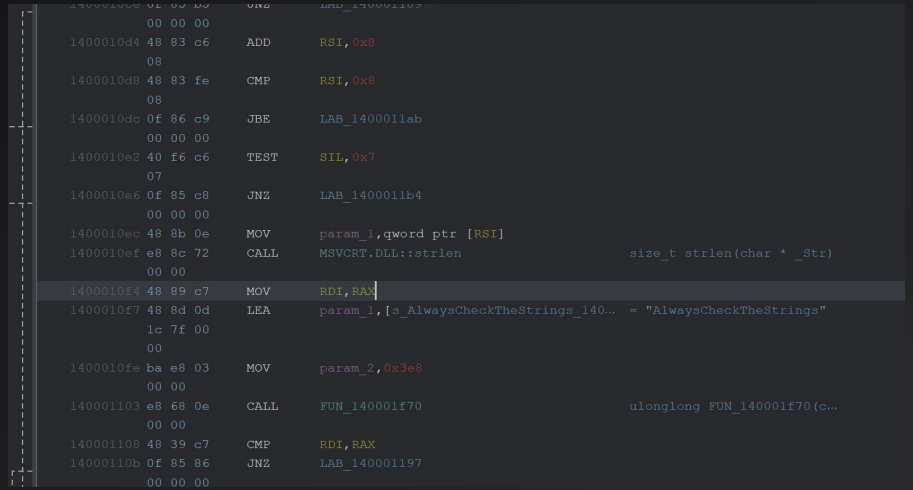
CrackMes 2021: A Savage Hypocrisy

Let’s look at the CrackMe’s I released for the first iteration of the class.

We will only cover the first 3, but you are welcome to check out the binaries/solutions on the previous course github

Crackme 0

- Intended difficulty: trivial (so long as you know how to find the main entry point!)
- Run strings on the binary.
- Find main
- See that it is directly comparing your input to a static string
- GG

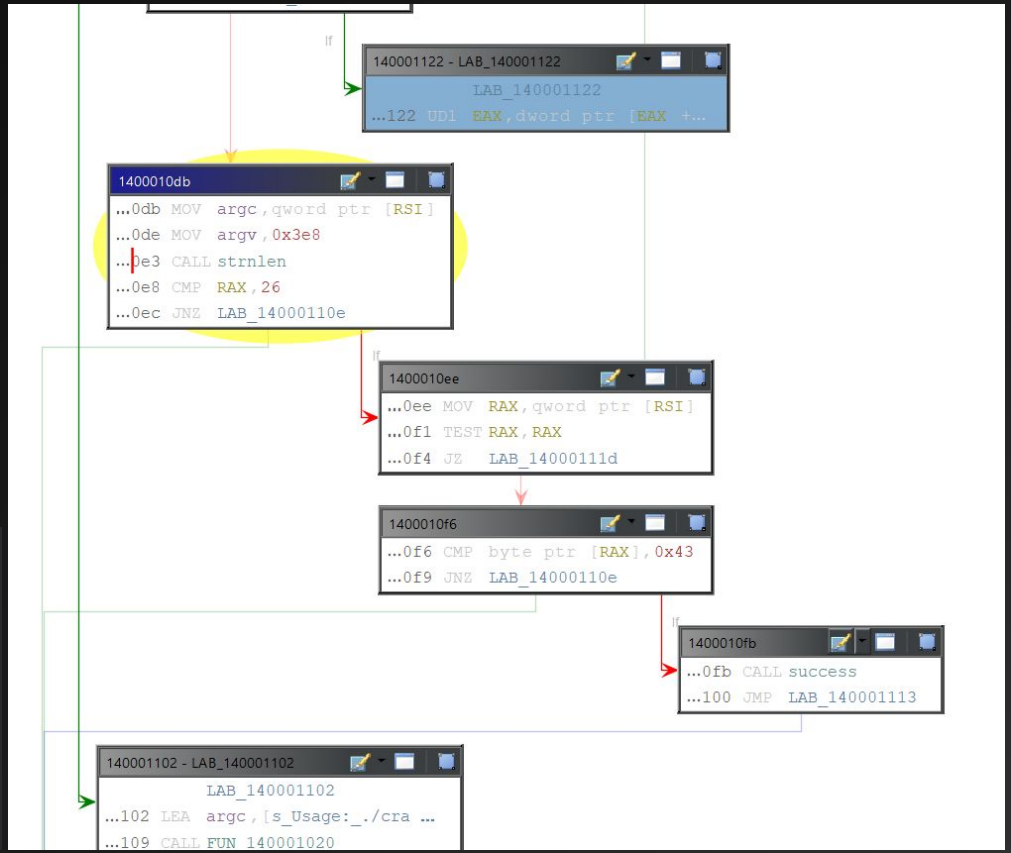


Crackme 1:

Intended Difficulty: Easy

The binary checks if the length of the input is 26, and if it starts with a C

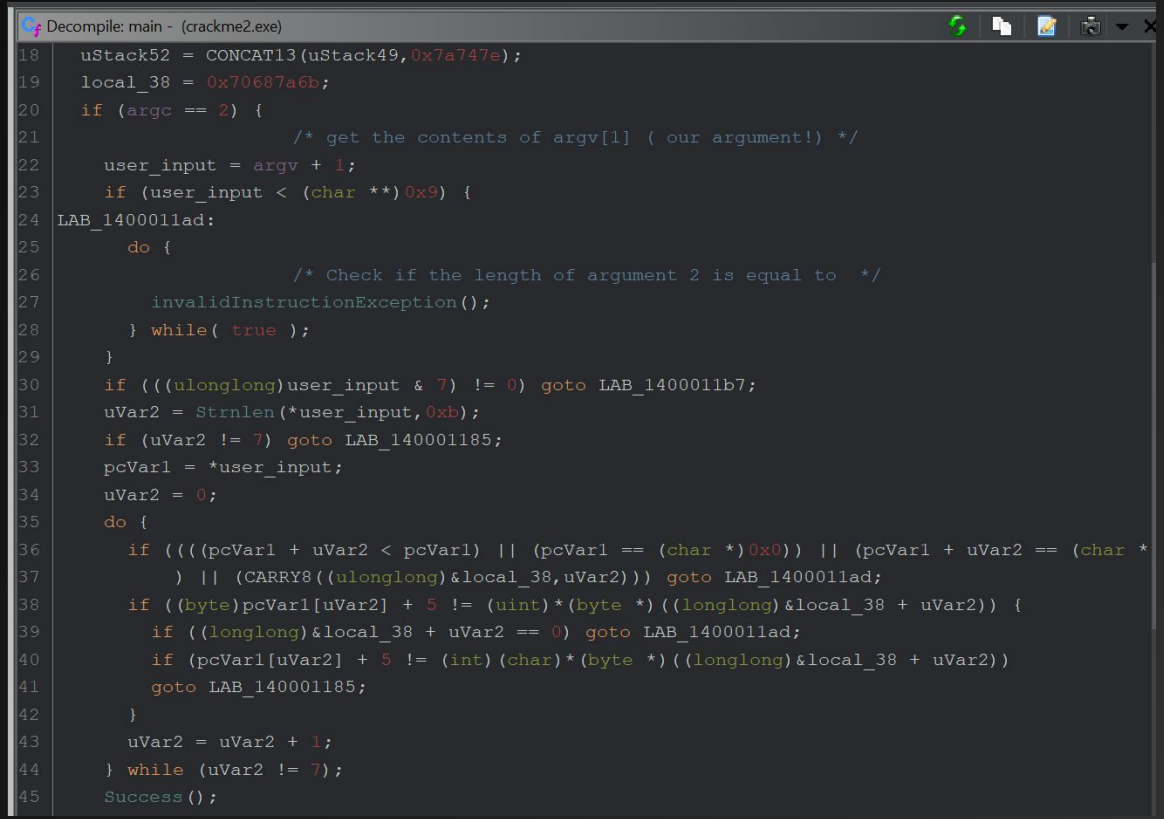
```
  uVar1 = strnlen(*ppcVar2,1000);
  if (uVar1 == 26) {
    if (*ppcVar2 == (char *)0x0) goto LAB_14000111d;
    if (**ppcVar2 == 'C') {
      success();
      return 0;
    }
  }
}
else {
```



Crackme 2 (Learn from my mistakes.)

Intended Difficulty:
Moderate

Question to answer:
WTF is happening inside of that do-while loop?



Crackme 2 (Again, learn from my mistakes!)

- A stack variable is declared, with values inside of the ASCII range (HINT)
- We iterate through values in argv[1]
- Copy that character into a buffer
- Copy the char from the stack var into a buffer
- Checks to see if the the input char + 5 is the same as the stack variable
- If not, Fail. If yes, continue
- Once we are done checking all the characters in the stack string with no errors, Success

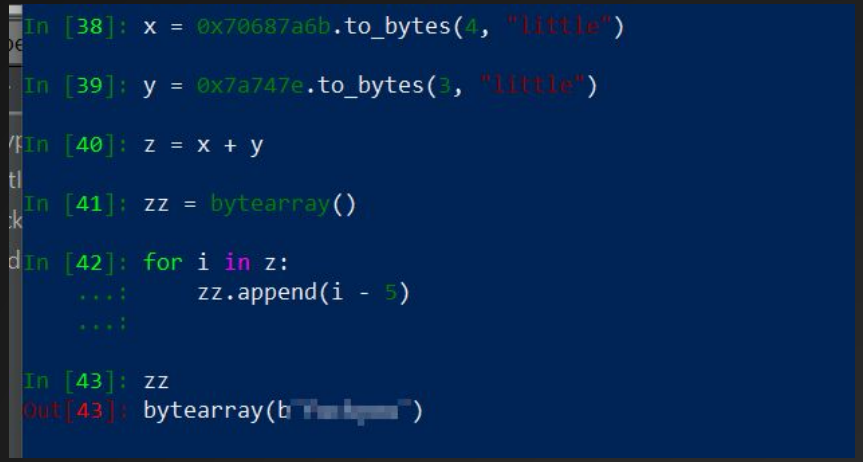
Crackme 2 (...Oops)

Moral of the story: Don’t leave your work until the night before it is due.

You will make careless mistakes, and if you’re like me and don’t have tenure it could be a bad look!

Also…maybe don’t use aggressive Debug statements.

OR git commit messages: https://www.youtube.com/watch?v=KjYBh7rq0-Y



PE File Format

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*

Tools

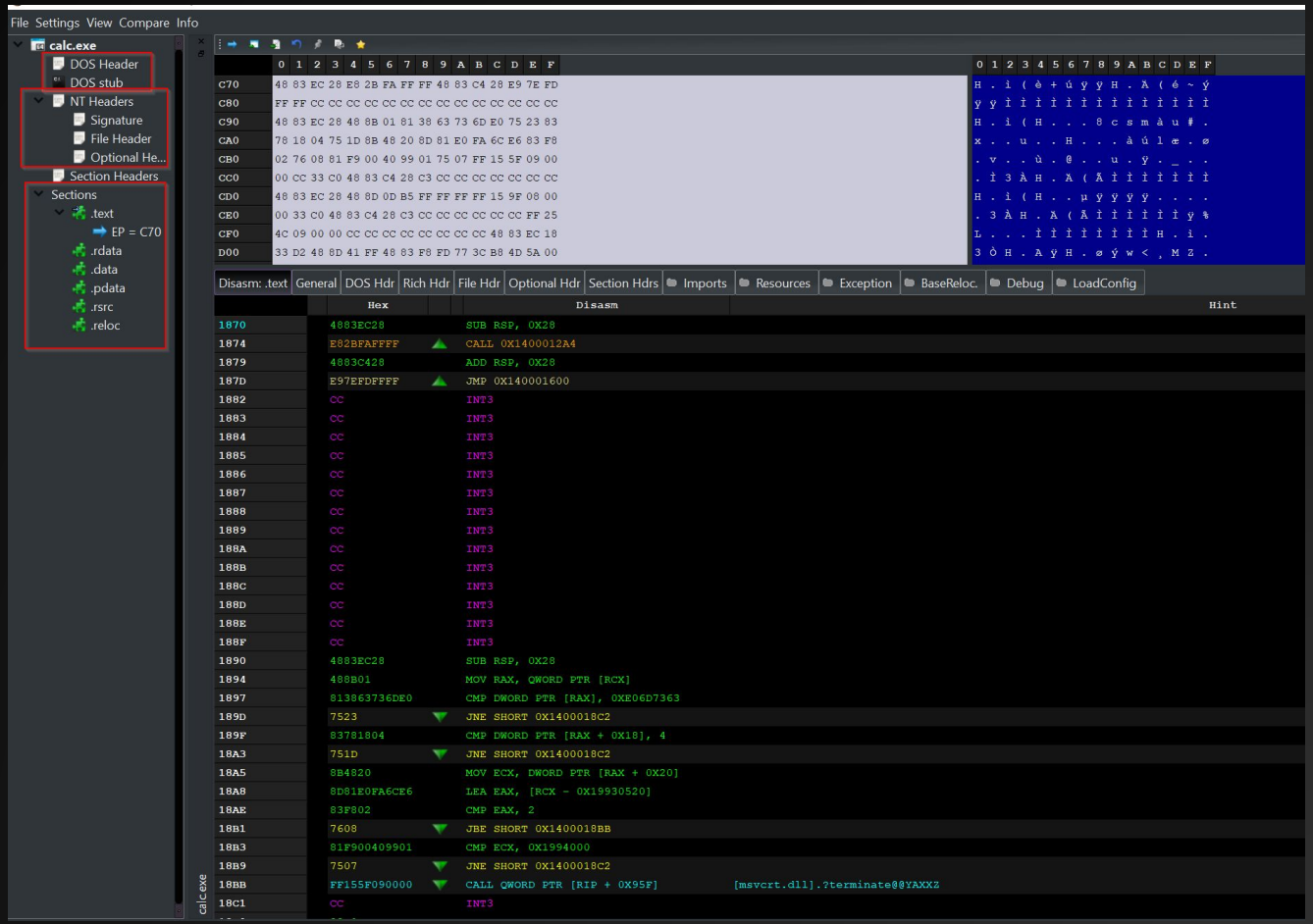
In this lecture, we will use x64dbg, and PE-Bear to explore the PE file format.

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

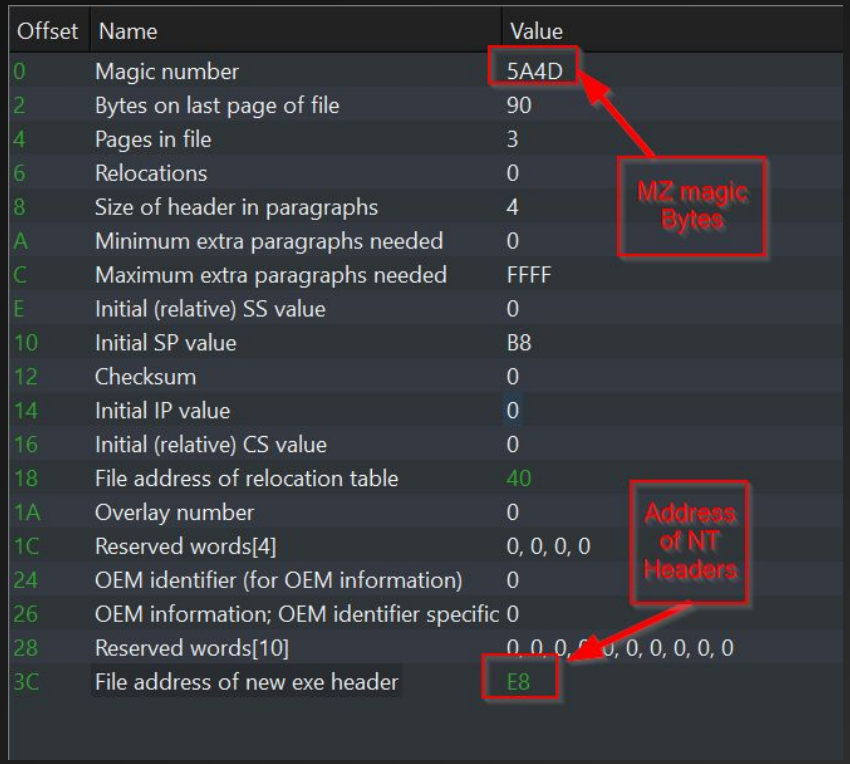
Run PE-Bear.exe $path.Source (in powershell)

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

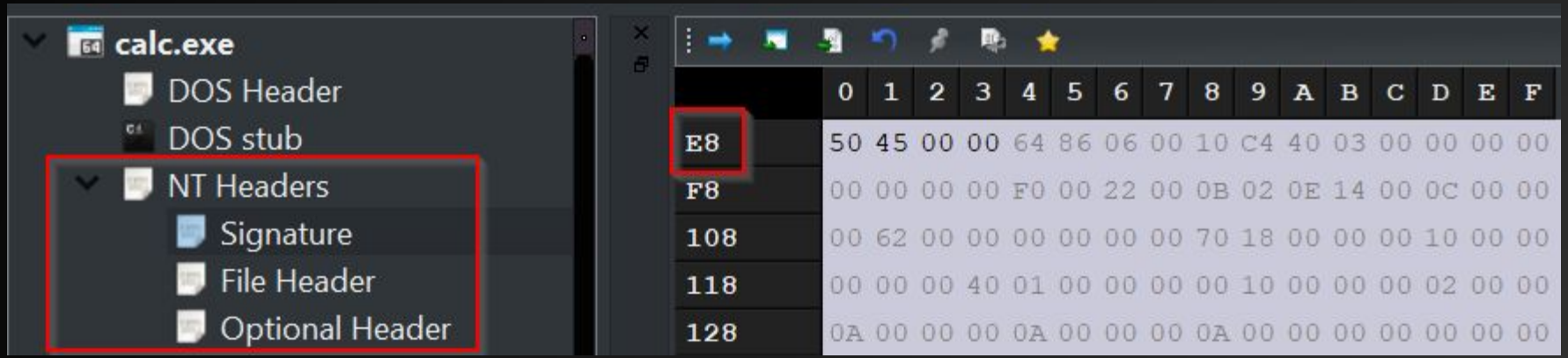
DOS Header

- 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



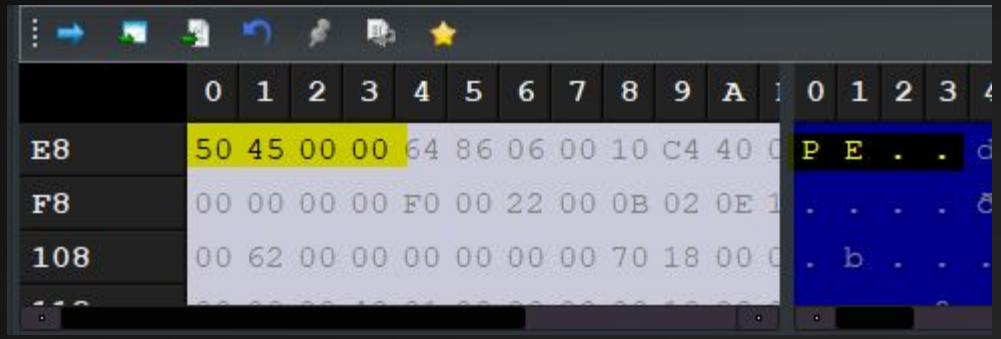
NT Headers

- Signatures
- File Header
- Optional Header



Signature

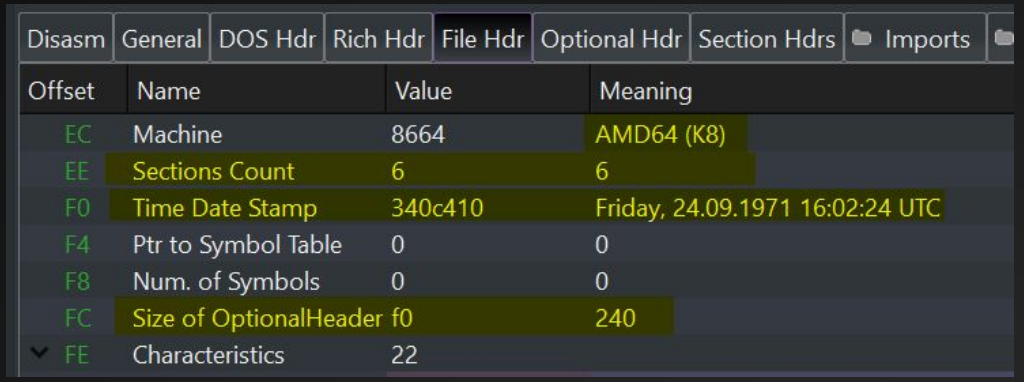
- Usually 4 bytes containing
- “PE\0\0”
- For our purposes, it is only used to verify the file format.



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

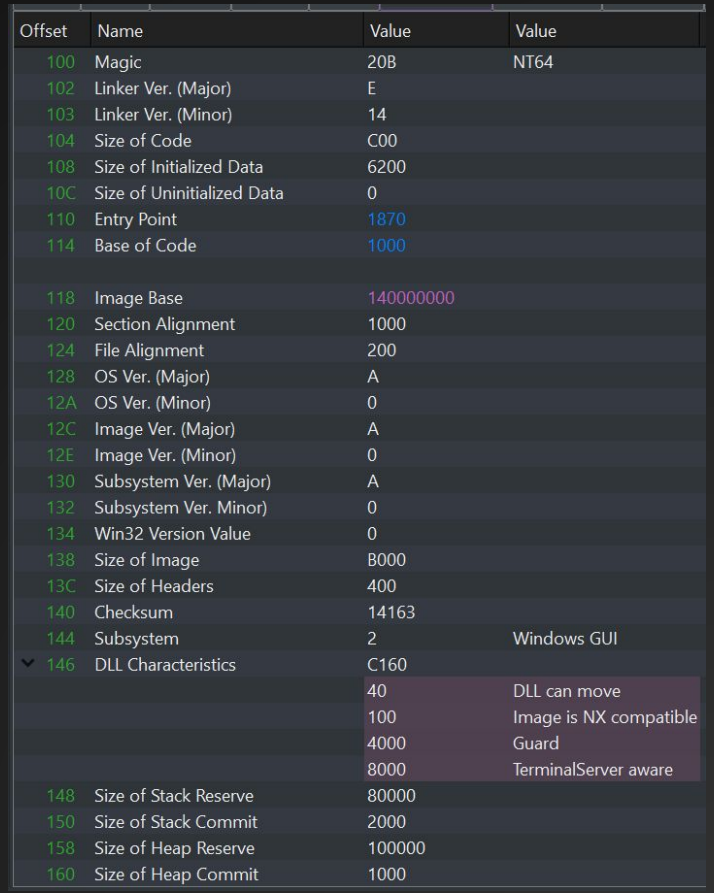


Optional Headers

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

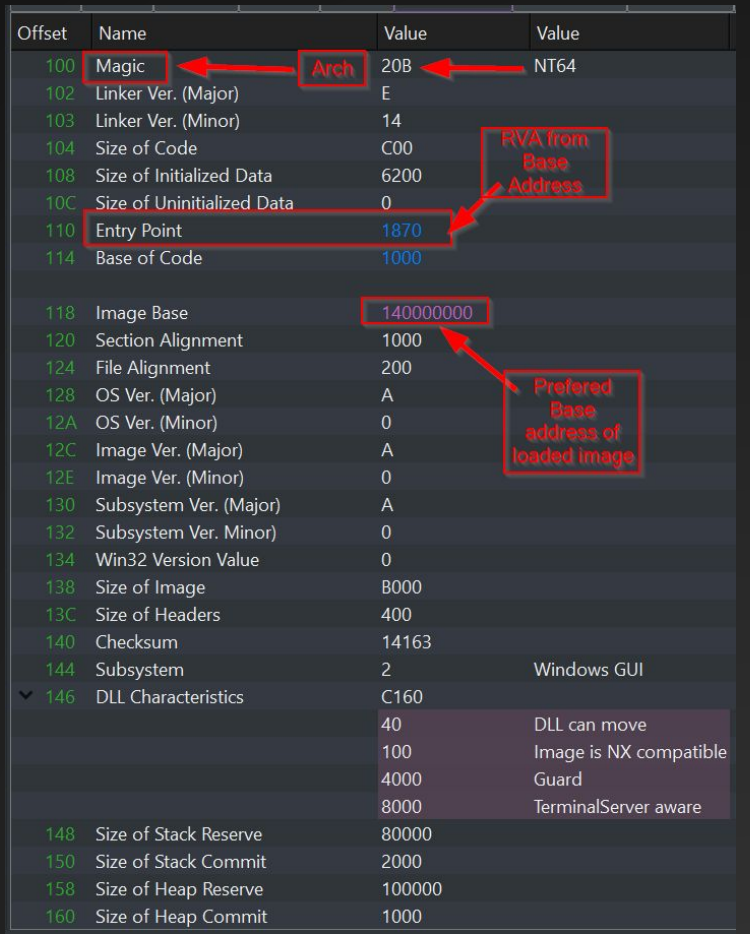
Specifically, values found here are used to build the *Import Address Table*, and perform *Base Relocations*



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)

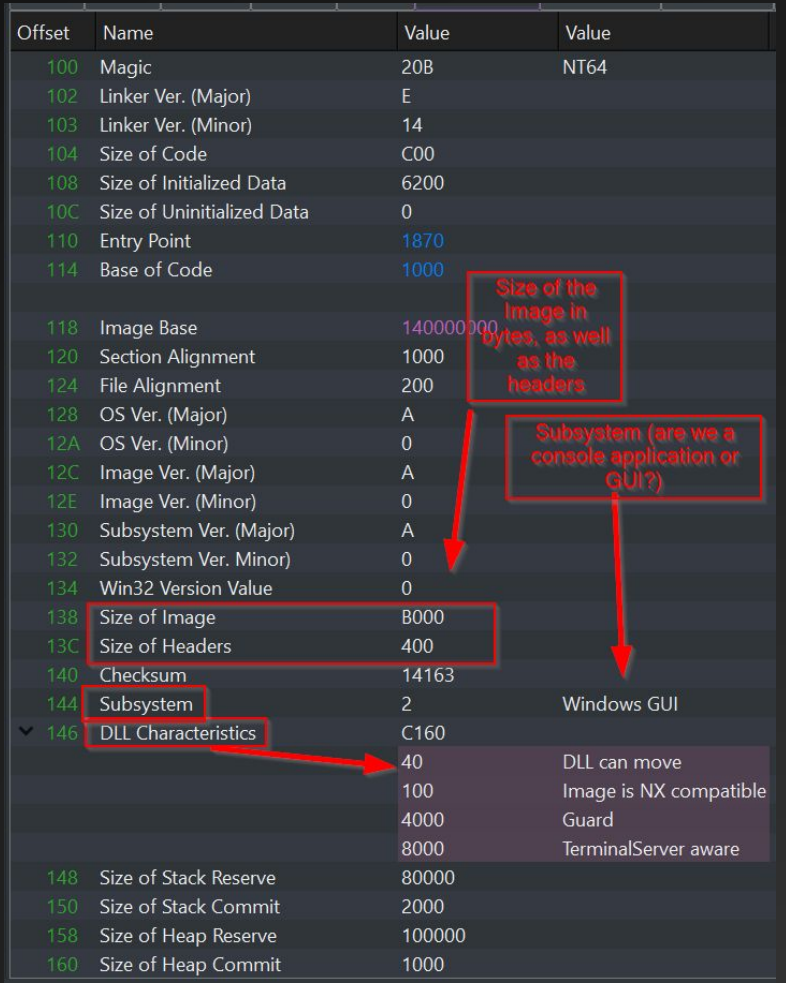
- Magic: Architecture of image
- Entry Point: Relative virtual address (RVA) from the Base Address
- Image Base: (prefered) Base address: Where in memory the PE “prefers” to be loaded. If the location is unavailable, the Image needs to be relocated



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.



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	

Resolving Imports

When a DLL is loaded by a process:

- It is not guaranteed to be placed in the same *absolute* location in memory
- Nor is it guaranteed to be placed in the same *relative* location

Resolving Imports

To handle variability in load location, the programer can can declare all imports in the *Import Address Table* (IAT)

This shifts the work to the PE loader to resolve all the imports away from the programmer!

When a dependency is declared in the IAT, the PE loader will attempt to resolve the dependency. If it fails, the program crashes.

Recall this is a subtype of dynamic linking called *Implicit Linking*

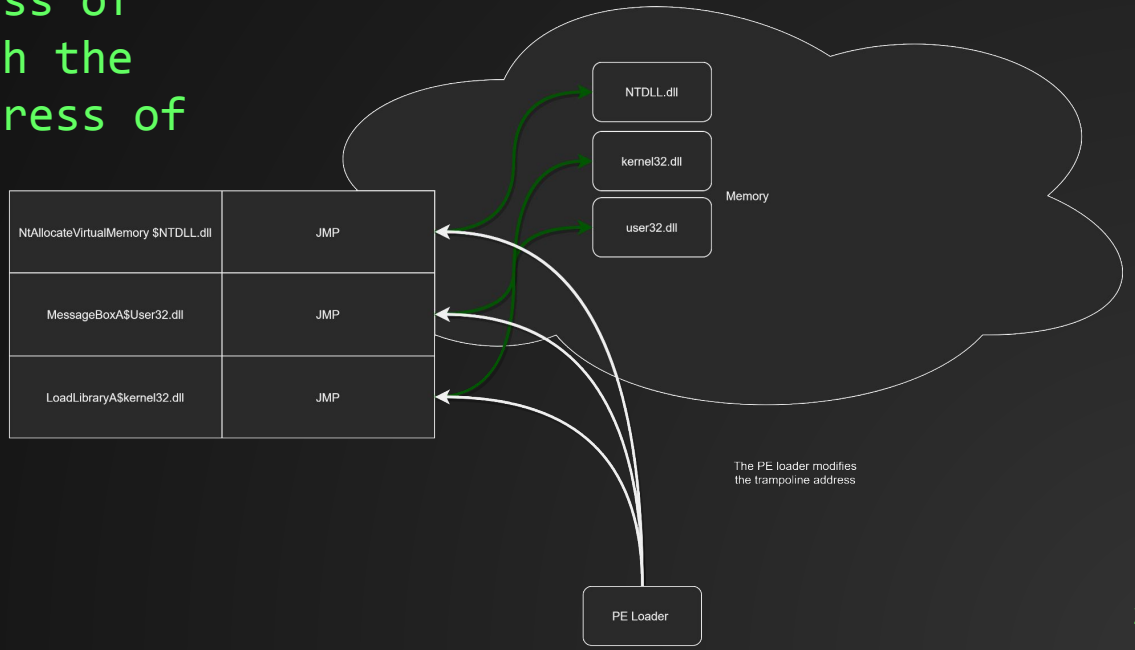
Resolving Imports

The IAT is effectively a collection of key value pairs, where the key is an identifier of an imported function, and the value is the address of a small function that jumps to the location of the real function.

This is reflected in x64dbg when we set a breakpoint at a call to a remote library: the breakpoint is set at the Virtual Addresses of the remote function, but in our code we reference it via the IAT

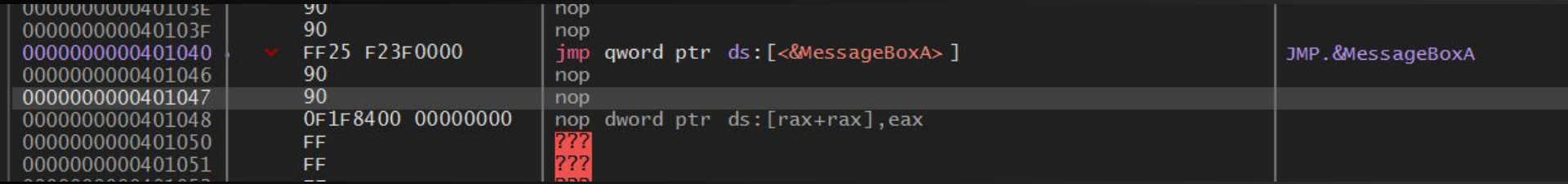
Resolving Imports

When resolving imports, the PE loader will load all required DLLs, identify the address of the loaded DLL, and patch the IAT with the correct address of the requested function

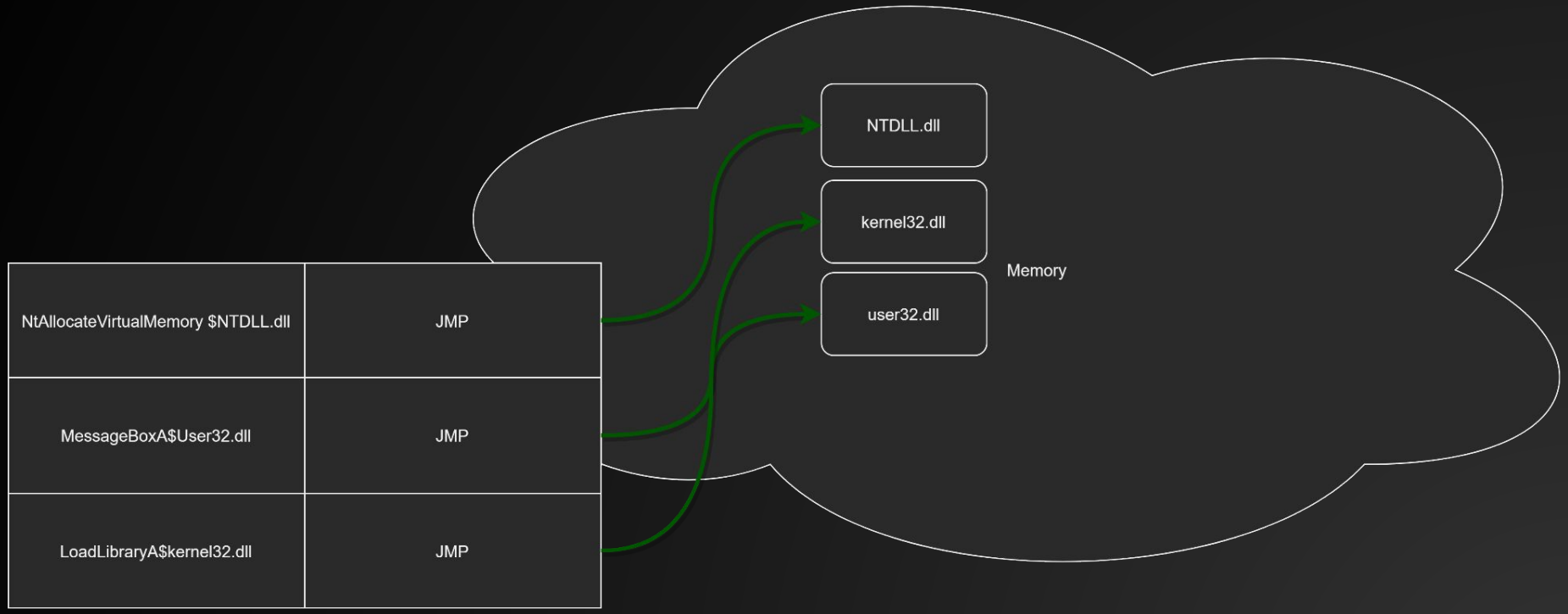


Resolving Imports

- The application code simply calls the function referenced in the IAT, which is itself a *trampoline* to the real code.
- I.e., it is a static location user code can can reference, which is simply an unconditional jump to the real virtual address of the required function.
- WHen a PE lives on disk, this *jump table* is null, and is set at runtime by the PE loader.

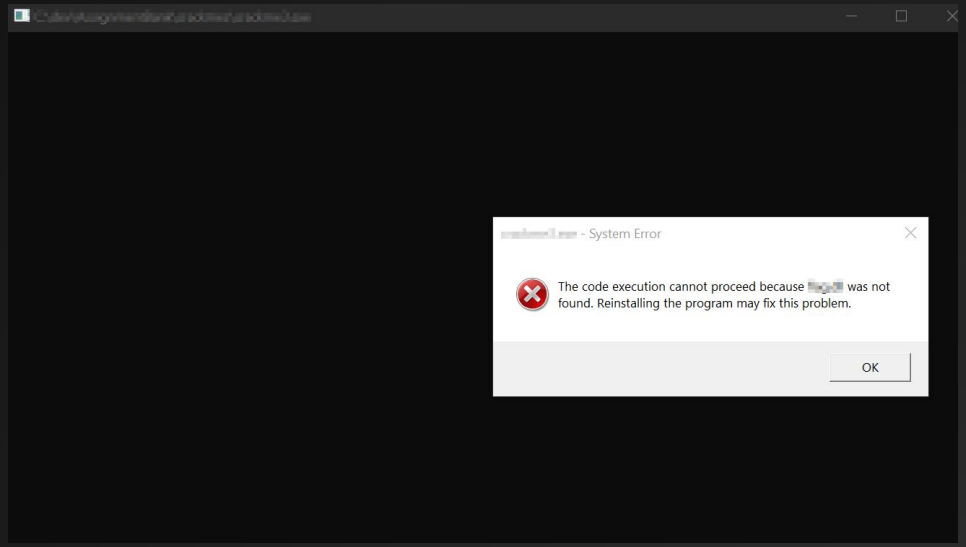


Visual: IAT


NTDLL.dll
kernel32.dll
user32.dll
Memory
NtAllocateVirtualMemory $NTDLL.dll
JMP
MessageBoxA$User32.dll
JMP
LoadLibraryA$kernel32.dll
JMP

Example of a crash:

Double click an EXE with a missing dependency, and you will likely see a message like this:



...Hint Hint Nudge Nudge...

You might need a bit of programming for 1 of the crackmes!


HINT HINT
NUDGE NUDGE

Dynamically Linked Libraries:

- Refresh: What is a DLL?
- How do we build a DLL?
- How do we execute a DLL?
- How do we export functions in a DLL?
- How do we call functions from 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

Building DLLs

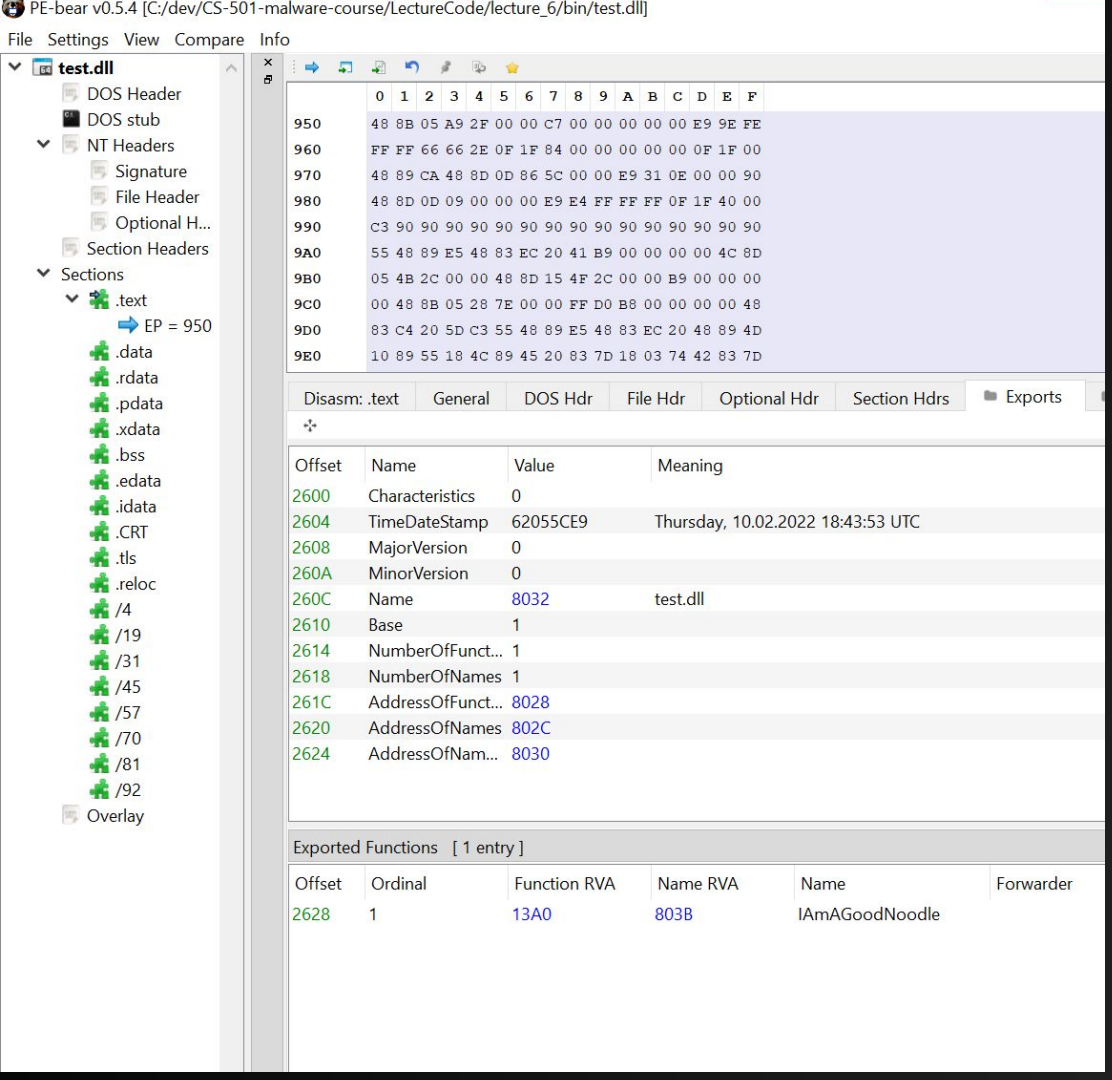
```
BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved)
{
    switch (fdwReason)
    {
    case DLL_PROCESS_ATTACH:
        printEntry();
        OutputDebugStringW(L"DLL_PROCESS_ATTACH");
        break;

    case DLL_THREAD_ATTACH:
        OutputDebugStringW(L"DLL_THREAD_ATTACH");
        break;

    case DLL_THREAD_DETACH:
        OutputDebugStringW(L"DLL_THREAD_DETACH");
        break;

    case DLL_PROCESS_DETACH:
        OutputDebugStringW(L"DLL_PROCESS_DETACH");
        break;
    }

    return TRUE;
}
PS C:\> g++ .\testdll.cpp -shared -o test.dll
```

Exporting Code

```
 __declspec(dllexport) int IAmAGoodNoodle()
{
    MessageBoxA(NULL, "I am a good noodle!", "Very good!", MB_OK );
  return 0;
}
```

Note for Cpp, you will need an extern “C” directive

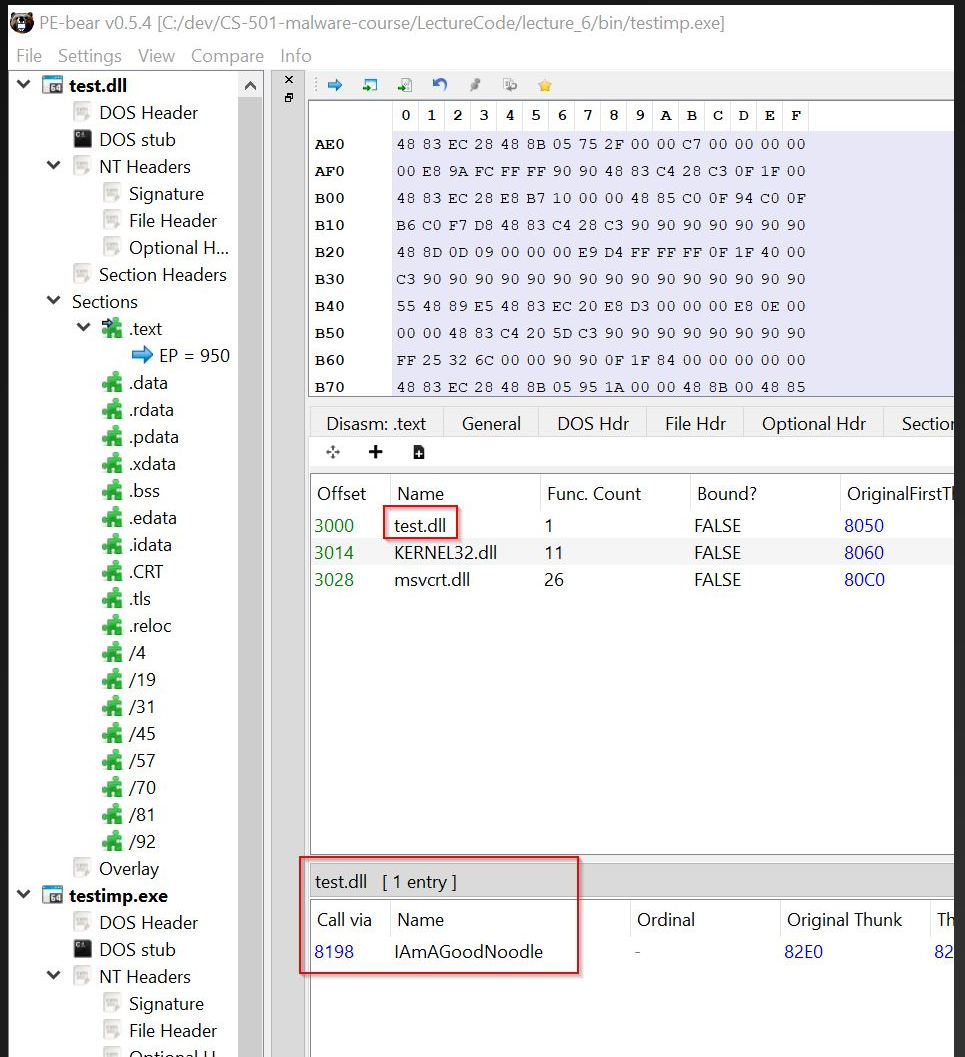
Demo
Building a DLL you can link against


PE-bear v0.5.4 [C:/dev/CS-501-malware-course/LectureCode/lecture_6/bin/test.dll]
File Settings View Compare Info
test.dll
DOS Header
DOS stub
NT Headers
Signature
File Header
Optional H...
Section Headers
Sections
.text
EP = 950
.data
.rdata
.pdata
.xdata
.bss
.edata
.idata
.CRT
.tls
.reloc
/4
/19
/31
/45
/57
/70
/81
/92
Overlay
0 1 2 3 4 5 6 7 8 9 A B C D E F
950 48 8B 05 A9 2F 00 00 C7 00 00 00 00 00 E9 9E FE
960 FF FF 66 66 2E 0F 1F 84 00 00 00 00 00 0F 1F 00
970 48 89 CA 48 8D 0D 86 5C 00 00 E9 31 0E 00 00 90
980 48 8D 0D 09 00 00 00 E9 E4 FF FF FF 0F 1F 40 00
990 C3 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
9A0 55 48 89 E5 48 83 EC 20 41 B9 00 00 00 00 4C 8D
9B0 05 4B 2C 00 00 48 8D 15 4F 2C 00 00 B9 00 00 00
9C0 00 48 8B 05 28 7E 00 00 FF D0 B8 00 00 00 00 48
9D0 83 C4 20 5D C3 55 48 89 E5 48 83 EC 20 48 89 4D
9E0 10 89 55 18 4C 89 45 20 83 7D 18 03 74 42 83 7D
Disasm: .text
General
DOS Hdr
File Hdr
Optional Hdr
Section Hdrs
Exports
Offset Name Value Meaning
2600 Characteristics 0
2604 TimeDateStamp 62055CE9 Thursday, 10.02.2022 18:43:53 UTC
2608 MajorVersion 0
260A MinorVersion 0
260C Name 8032 test.dll
2610 Base 1
2614 NumberOfFunct... 1
2618 NumberOfNames 1
261C AddressOfFunct... 8028
2620 AddressOfNames 802C
2624 AddressOfNam... 8030
Exported Functions [1 entry]
Offset Ordinal Function RVA Name RVA Name Forwarder
2628 1 13A0 803B IAmAGoodNoodle

DLL Search

The PE loader will check multiple well known paths, finally checking the current directory for the required DLL

If you aren’t careful with where you set your dependency, you can open up your code to DLL hijacking



Running DLLs

Rundl32.exe (there is a 32-bit version and a 64-bit version)

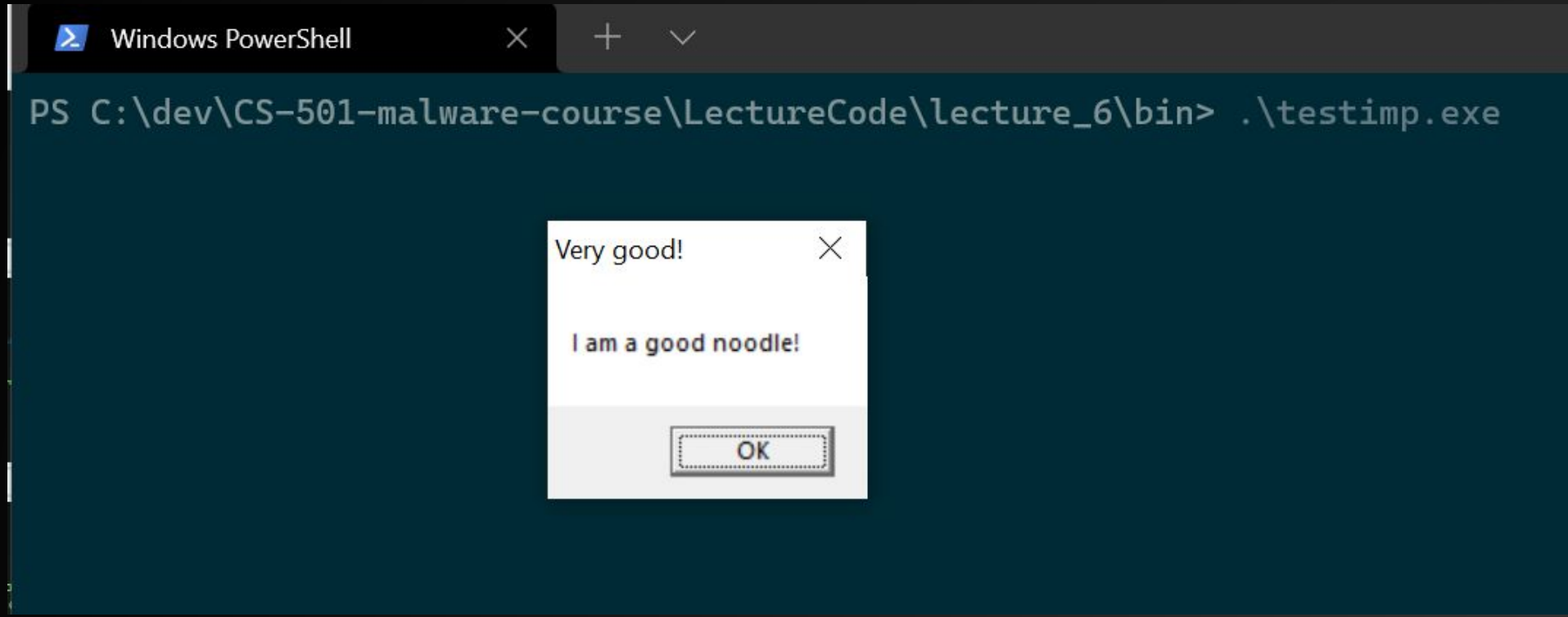
Rundl32.exe can execute specific functions, or simply the DLLMain

If you give it the name of an unexported function, it will call ProcessAttach then error out

Testing our DLL: rundl32

```
    Directory: C:\dev\CS-501-malware-course\LectureCode\lecture_6


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----         2/10/2022   1:43 PM                bin
-a----         2/10/2022  12:48 PM          89056 a.exe
-a----         2/10/2022   1:43 PM            337 Makefile
-a----         2/10/2022  12:48 PM            618 mbox.c
-a----         2/10/2022   1:39 PM            788 testdll.c
-a----         2/10/2022   1:42 PM             94 testimp.c
-a----         2/10/2022   1:43 PM          57551 testimp.exe


PS C:\dev\CS-501-malware-course\LectureCode\lecture_6> cd .\bin\
PS C:\dev\CS-501-malware-course\LectureCode\lecture_6\bin> rundll32.exe
PS C:\dev\CS-501-malware-course\LectureCode\lecture_6\bin> rundll32.exe .\test.dll
PS C:\dev\CS-501-malware-course\LectureCode\lecture_6\bin> rundll32.exe .\test.dll IAmAGoodNoodle
PS C:\dev\CS-501-malware-course\LectureCode\lecture_6\bin>
```

Testing our DLL: Implicit Linking



Dynamic Linking→Explicit Linking

- The programer explicitly calls LoadLibrary followed by GetProcAddress, and has the proper function prototypes inside of their code. Bot of these functions live in Kernel32.dll
- The legitimate reason to do this, is to allow the execution of the program to continue even if a DLL does not exist.
- If an import for an implicitly loaded DLL is missing, the program stops.
- The application using an explicitly loaded DLL can choose how to handle a missing DLL.
- This is a common in malware trying to hide its imports

How does this work?

- LoadLibraryA: gets a handle to a library in memory.
- If the library isn’t already loaded into memory, it will load it! If it can’t, it returns a NULL pointer
  - The handle it returns is the Virtual Address of the memory mapped library for the current processes Private Address space!
    - In particular, it is the *base address* of the library.
- GetProcAddress: given a handle to a library, and an identifier for a function such as a name, it returns the Virtual Address of the funcion!

Using Function Pointers

- Functions aren’t first class objects in C/C++
- However, functions themselves are just memory addresses!
  - with arguments that give the compiler hints on how to call it!
- To invoke a function by a memory address, we first need the address of the function, and information about how to call it
- We can either handle this using raw assembly, or we can declare a function pointer type with the arguments and calling conventions specified
- Then we just cast the raw address to a function pointer type and call it!

Demo:
Explicitly Linked Mbox

Discussion:
Defense Evasion using different types of linking!