





# Lecture 0x07: DLLs, Linking Continued

# Crackmes: Fall 2021

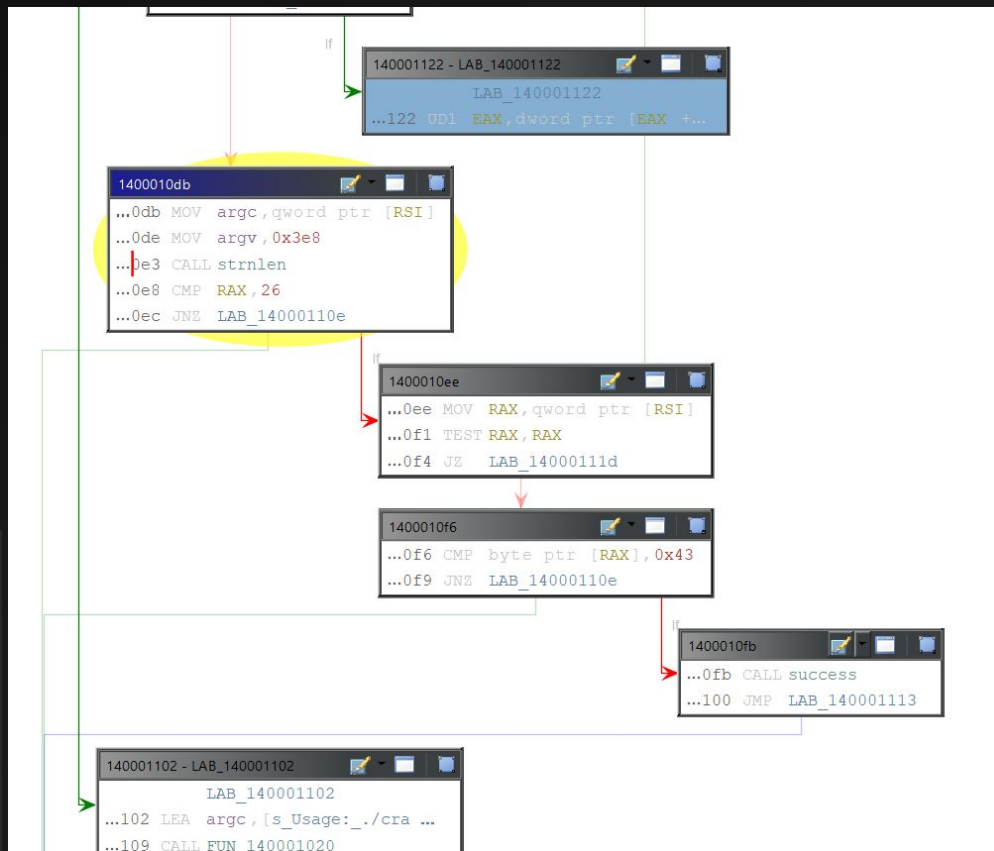


# Crackme 1:

Intended Difficulty: Easy

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

```
uVar1 = strlen(*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?

```
Decompile: main - (crackme2.exe)
18  uStack52 = CONCAT13(uStack49,0x7a747e);
19  local_38 = 0x70687a6b;
20  if (argc == 2) {
21      /* get the contents of argv[1] ( our argument!) */
22      user_input = argv + 1;
23      if (user_input < (char **)0x9) {
24 LAB_1400011ad:
25          do {
26              /* Check if the length of argument 2 is equal to */
27              invalidInstructionException();
28          } while( true );
29      }
30      if (((ulonglong)user_input & 7) != 0) goto LAB_1400011b7;
31      uVar2 = Strnlen(*user_input,0xb);
32      if (uVar2 != 7) goto LAB_140001185;
33      pcVar1 = *user_input;
34      uVar2 = 0;
35      do {
36          if (((pcVar1 + uVar2 < pcVar1) || (pcVar1 == (char *)0x0)) || (pcVar1 + uVar2 == (char *
37              ) || (CARRY8((ulonglong)&local_38,uVar2))) goto LAB_1400011ad;
38          if ((byte)pcVar1[uVar2] + 5 != (uint)*(byte *)((longlong)&local_38 + uVar2)) {
39              if ((longlong)&local_38 + uVar2 == 0) goto LAB_1400011ad;
40              if (pcVar1[uVar2] + 5 != (int)(char)*(byte *)((longlong)&local_38 + uVar2))
41                  goto LAB_140001185;
42          }
43          uVar2 = uVar2 + 1;
44      } while (uVar2 != 7);
45      Success();
```

## 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 (...0ops)

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>

```
In [38]: x = 0x70687a6b.to_bytes(4, "little")  
In [39]: y = 0x7a747e.to_bytes(3, "little")  
In [40]: z = x + y  
In [41]: zz = bytearray()  
In [42]: for i in z:  
...:     zz.append(i - 5)  
...:  
In [43]: zz  
Out[43]: bytearray(b'\x0f\x0d\x0c')
```

# Review: 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*

# Calc.exe

File Settings View Compare Info

calc.exe

- DOS Header
- DOS stub
- NT Headers
  - Signature
  - File Header
  - Optional Headers
- Section Headers
  - .text → EP = C70
  - .rdata
  - .data
  - .pdata
  - .rsrc
  - .reloc

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
C70	48	83	EC	28	E8	2B	FA	FF	FF	48	83	C4	28	E9	7E	FD
C80	FF	FF	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC
C90	48	83	EC	28	E8	0B	01	81	38	63	73	6D	E0	75	23	83
CA0	78	18	04	75	1D	8B	48	20	8D	81	E0	FA	6C	E6	83	F8
CB0	02	76	08	81	F9	00	40	99	01	75	07	FF	15	5F	09	00
CC0	00	CC	33	C0	48	83	C4	28	C3	CC	CC	CC	CC	CC	CC	CC
CD0	48	83	EC	28	E8	0D	B5	FF	FF	FF	FF	15	9F	08	00	
CE0	00	33	C0	48	83	C4	28	C3	CC	CC	CC	CC	CC	CC	FF	25
CF0	4C	09	00	00	CC	CC	CC	CC	CC	CC	CC	CC	48	83	EC	18
D00	33	D2	48	8D	41	FF	48	83	F8	FD	77	3C	B8	4D	5A	00

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
H	.	i	(	+	ú	y	y	H	.	À	(	~	ý			
y	y	i	i	i	i	i	i	i	i	i	i	i	i	i	i	i
H	.	i	(	H	.	.	.	8	c	s	m	à	u	#	.	
x	.	u	.	.	H	.	.	.	à	ú	1	æ	.	ø		
.	v	.	ù	.	ø	.	u	.	y	.	.	.	.	.		
.	z	3	À	H	.	À	(	À	i	i	i	i	i	i	i	i
H	.	i	(	H	.	.	.	u	y	y	y	.	.	.	.	
.	3	À	H	.	À	(	À	i	i	i	i	i	i	y	%	
L	.	.	.	.	i	i	i	i	i	i	i	i	H	.	i	.
3	Ò	H	.	À	y	H	.	ø	y	w	<	.	M	Z	.	

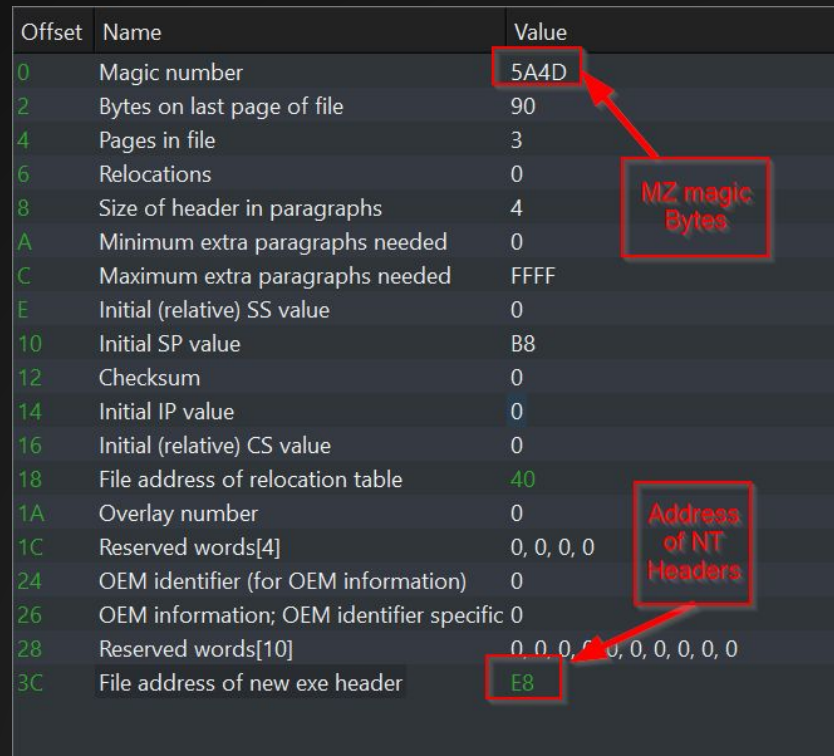
Disasm: .text | General | DOS Hdr | Rich Hdr | File Hdr | Optional Hdr | Section Hdrs | Imports | Resources | Exception | BaseReloc | Debug | LoadConfig | Hint

	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	81F909409901	CMP ECX, 0x1994000	
18B9	7507	JNE SHORT 0x1400018C2	
18BB	FF155F090000	CALL QWORD PTR [RIP + 0x95F]	[msvcrt.dll].?terminate@YAXXZ
18C1	CC	INT3	

# 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

Offset	Name	Value
0	Magic number	5A4D
2	Bytes on last page of file	90
4	Pages in file	3
6	Relocations	0
8	Size of header in paragraphs	4
A	Minimum extra paragraphs needed	0
C	Maximum extra paragraphs needed	FFFF
E	Initial (relative) SS value	0
10	Initial SP value	B8
12	Checksum	0
14	Initial IP value	0
16	Initial (relative) CS value	0
18	File address of relocation table	40
1A	Overlay number	0
1C	Reserved words[4]	0, 0, 0, 0
24	OEM identifier (for OEM information)	0
26	OEM information; OEM identifier specific	0
28	Reserved words[10]	0, 0, 0, 0, 0, 0, 0, 0, 0, 0
3C	File address of new exe header	E8



# NT Headers

- Signatures
- File Header
- Optional Header

calc.exe

- DOS Header
- DOS stub
- NT Headers
  - Signature
  - File Header
  - Optional Header

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
E8	50	45	00	00	64	86	06	00	10	C4	40	03	00	00	00	00
F8	00	00	00	00	F0	00	22	00	0B	02	0E	14	00	0C	00	00
108	00	62	00	00	00	00	00	00	70	18	00	00	00	10	00	00
118	00	00	00	40	01	00	00	00	00	10	00	00	00	02	00	00
128	0A	00	00	00	0A	00	00	00	0A	00	00	00	00	00	00	00

# Signature

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

A screenshot of a hex editor window. The interface includes a toolbar at the top with icons for navigation and editing. The main area is a table with two columns: the left column shows memory addresses (E8, F8, 108, 110) and the right column shows the corresponding hex values. The row for address E8 is highlighted in yellow, showing the hex values 50 45 00 00. To the right of the hex values, the ASCII representation of the data is shown, which is 'PE..'. The row for address F8 shows hex values 00 00 00 00 F0 00 22 00 0B 02 0E 1, and the row for address 108 shows hex values 00 62 00 00 00 00 00 00 70 18 00 0.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
E8	50	45	00	00	64	86	06	00	10	C4	40	00	00	00	00	00
F8	00	00	00	00	F0	00	22	00	0B	02	0E	10	00	00	00	00
108	00	62	00	00	00	00	00	00	70	18	00	00	00	00	00	00
110	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00



# 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

Disasm	General	DOS Hdr	Rich Hdr	File Hdr	Optional Hdr	Section Hdrs	Imports
Offset	Name	Value	Meaning				
EC	Machine	8664	AMD64 (K8)				
EE	Sections Count	6	6				
F0	Time Date Stamp	340c410	Friday, 24.09.1971 16:02:24 UTC				
F4	Ptr to Symbol Table	0	0				
F8	Num. of Symbols	0	0				
FC	Size of OptionalHeader f0	240					
FE	Characteristics	22					

# 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	8000	
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: (preferred) 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	8000	
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	

Size of the Image in bytes, as well as the headers

Subsystem (are we a console application or GUI?)

# 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 programmer 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

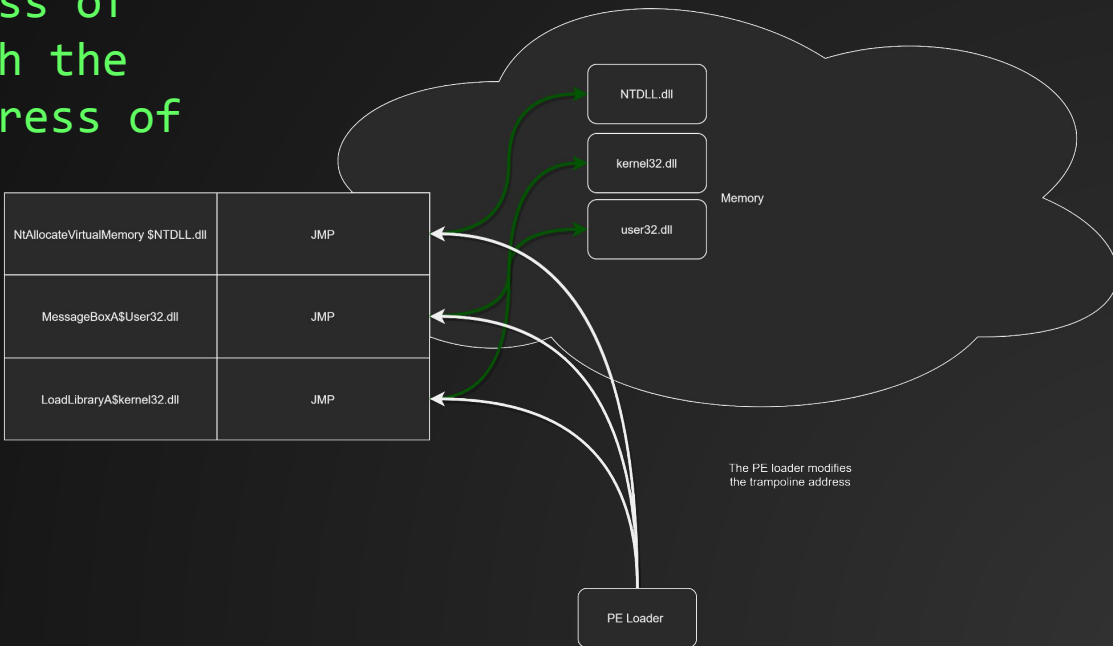
The IAT is effectively a collection of key value pairs, where the key is an identifier of an imported function, and the value is the address of a small function that jumps to the location of the real function.

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



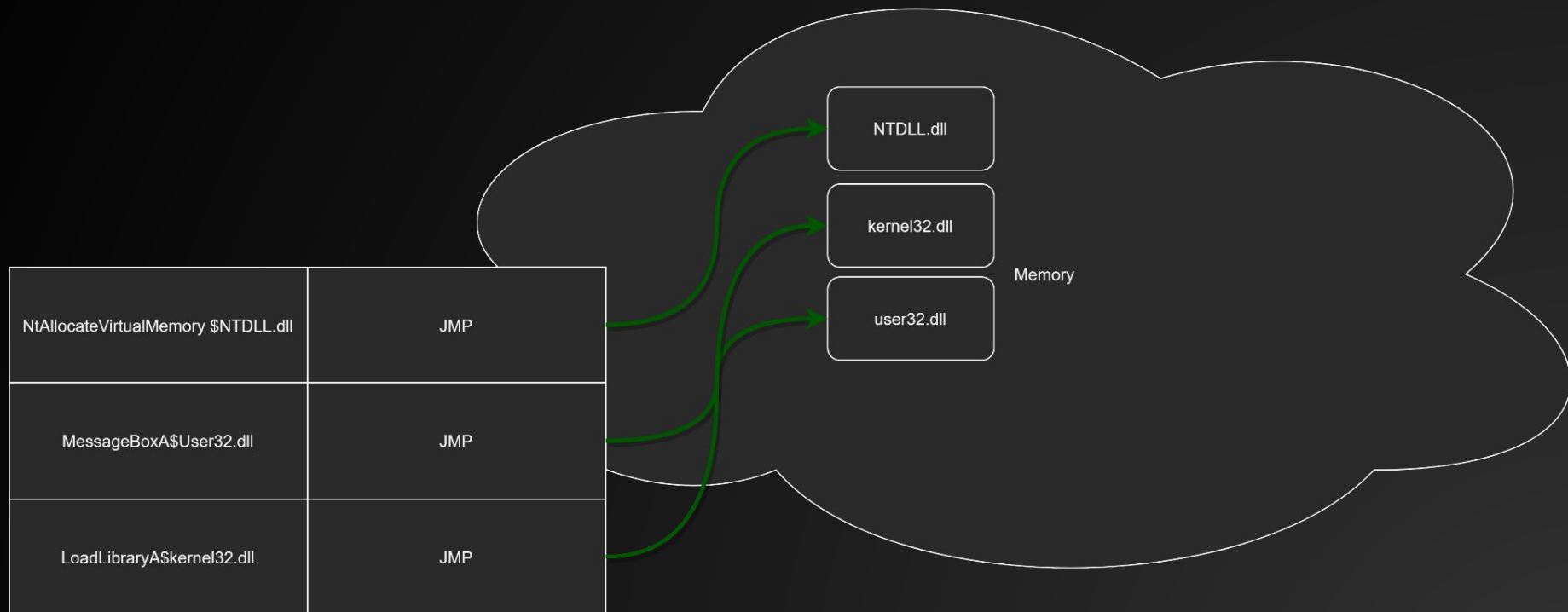


# Resolving Imports

- 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 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.

000000000040103E	90	nop	
000000000040103F	90	nop	
0000000000401040	FF25 F23F0000	jmp qword ptr ds:[<&MessageBoxA>]	JMP.&MessageBoxA
0000000000401046	90	nop	
0000000000401047	90	nop	
0000000000401048	0F1F8400 00000000	nop dword ptr ds:[rax+rax],eax	
0000000000401050	FF	???	
0000000000401051	FF	???	
0000000000401052	FF	???	

# Visual: IAT

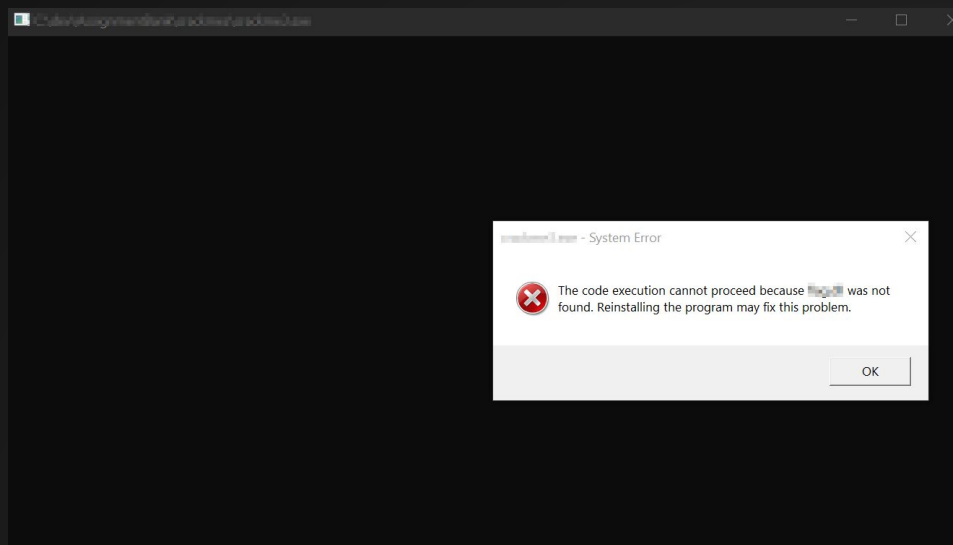


## Example of a crash:

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!





# 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

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

Disasm: .text    General    DOS Hdr    File Hdr    Optional Hdr    Section Hdrs    Exports

Offset	Name	Value	Meaning
2600	Characteristics	0	
2604	TimeStamp	62055CE9	Thursday, 10.02.2022 18:43:53 UTC
2608	MajorVersion	0	
260A	MinorVersion	0	
260C	Name	8032	test.dll
2610	Base	1	
2614	NumberOfFunc...	1	
2618	NumberOfNames	1	
261C	AddressOfFunc...	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

PE-bear v0.5.4 [C:/dev/CS-501-malware-course/LectureCode/lecture\_6/bin/testimp.exe]

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

**testimp.exe**

- DOS Header
- DOS stub
- NT Headers
  - Signature
  - File Header
  - Optional H...

Disasm: .text General DOS Hdr File Hdr Optional Hdr Section

Offset	Name	Func. Count	Bound?	OriginalFirstT
3000	test.dll	1	FALSE	8050
3014	KERNEL32.dll	11	FALSE	8060
3028	msvcrt.dll	26	FALSE	80C0

test.dll [ 1 entry ]

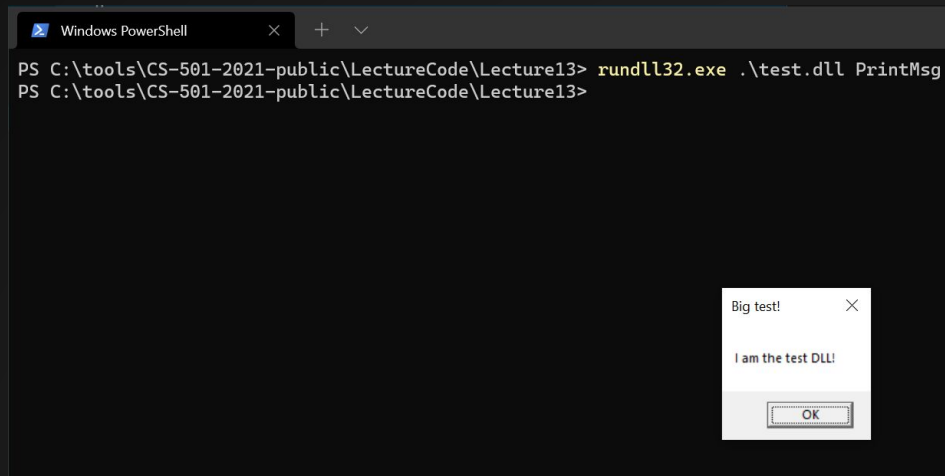
Call via	Name	Ordinal	Original Thunk	Th
8198	IAmAGoodNoodle	-	82E0	82

# 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

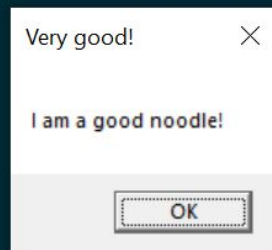
If you give it the name of an unexported function, it will call ProcessAttach then error out



# Testing our DLL: rundll32

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

Windows PowerShell

```
PS C:\dev\CS-501-malware-course\LectureCode\lecture_6\bin> .\testimp.exe
```

Very good!

I am a good noodle!

OK

## Dynamic Linking→Explicit Linking

- The programmer explicitly calls LoadLibrary followed by GetProcAddress, and has the proper function prototypes inside of their code. Both 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 function!



# 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

# Reverse Engineering Implications

- If a dll is explicitly linked, you cannot set a breakpoint for the library in the entry point (why?)
- The DLL doesn't show up in the IAT

How can we hide the strings for LoadLibrary/GetProcAddress?

Discussion:  
Defense Evasion using different  
types of linking!