Reverse Engineering and
Malware Analysis Fundamentals

Portable Executable Format

Adam Podlosky
@apodlosky

Objectives

- Understand the Portable Executable (PE) format
- Observe how malware often misuses and/or abuses PEs
- Demonstrate select reverse engineering tools and techniques
  - Knowledge of C and x86 assembly is not required…
  - ...but it may help

What is a Portable Executable?

- PE is the file format for executables on Microsoft Windows
- Like the ELF (Executable and Linkable Format) is to *nix
- PE files denoted by .EXE, .DLL, .SYS extensions (and others)
- What is really inside a PE?
  - **Headers** – structures
    - Contains offsets – “**RVAs**”, bitmasks, word values etc.
  - **Sections** – data
    - Sections contain code, strings, images, etc.

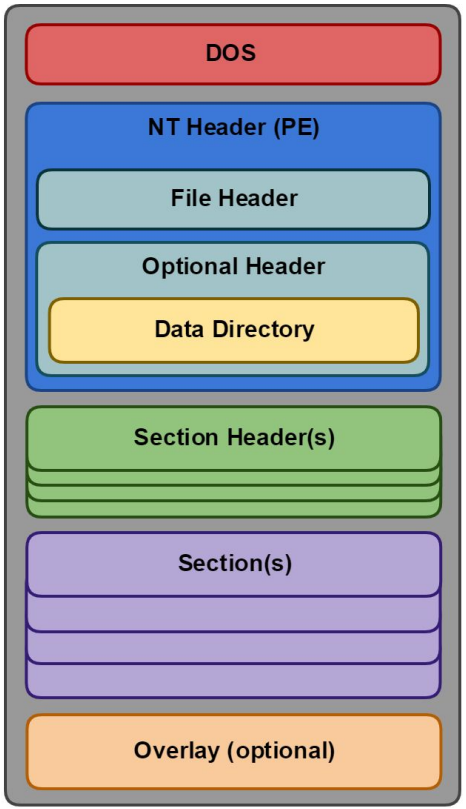
History of the Portable Executable

- **MZ** format (16-bit) in MS-DOS (~1980? Before my time…)
  - MZ executables used the .EXE file extension
  - Execution compatibility removed in Windows x64
- **PE32** (32-bit) introduced with Windows NT 3.1 in 1993
- **PE32+** (64-bit) originally for DEC Alpha CPUs, never released
  - First x86-64 (AMD64/EM64T) “x64” version in 2003
  - Intel *Itanic*...Itanium...version came somewhere in between

Terminology

- **Compiler**
  - Converts source code into machine code, intermediate files
- **Linker**
  - “Glues” intermediate files together to make a binary (PE)
- **RVA – “offsets”**
  - Relative Virtual Address, relative to a **Base** address
  - Explained later...

Headers, Stubs, Directories, and Sections



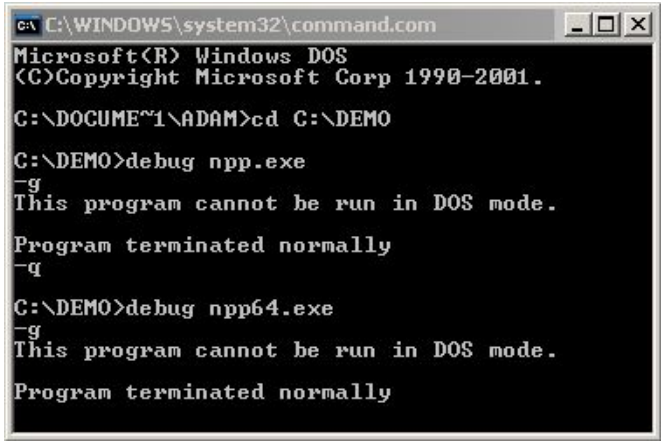
- Legacy **DOS header** and **DOS stub**
- Modern **NT header** (PE header)
  - Includes **File** and **Optional** Headers
- **Section headers** table (ToC of sections)
- **Sections** with code, data, resources, etc.
- **Overlay** is appended data, NOT in a section

DOS Header

- Every PE file begins with the DOS header
  - 64 bytes in length
  - First two bytes **magic** value “MZ”
  - Last four bytes are the **offset** to the NT header
- Everything between is ignored
  - Unless executing the PE in MS-DOS…

DOS Stub

- All PEs have a 16-bit MS-DOS 2.0 program to print “**This program cannot be run in DOS mode**.” and exit



- However, a 64-bit PE considered an invalid application on 32-bit versions of Windows

NT Header – the “PE Header”

- Located by DOS header
- Begins with 4-byte **Signature** “PE00”
- Contains **File Header**
  - Important values for other headers
- Contains **Optional Header**
  - Not optional, required on Windows
  - Different on 32bit vs 64bit

File Header

- **Machine** architecture (x86-32, x86-64, ARM, MIPS, etc.)
- Build **Timestamp** used for bound imports
- **Number of Sections** in the section header table
- **Size of Optional Header** can vary (includes padding)
  - Used to locate the first section header
- **Characteristics** of PE: executable, DLL, no relocations, etc.

(Not) Optional Header

- **Address Of Entry Point** is the RVA to begin execution at
- **Image Base** (virtual address) to map PE at
- **DLL Characteristics** flags that influence the Loader
  - Mostly enabling security features
- Windows **Subsystem** to exec.: CLI, GUI, driver, (& others)
- **File** and **Section Alignment** for on disk and in-memory
  - Relationship between these values is important

(Not) Optional Header Cont.

- Fields for header sizes, adjusting stack/heap paging
- Mostly ignored or not used
  - Checksum (only verified for drivers)
  - Base of: Code, Data
  - Size of: Code, Init. Data, Uninit. Data
  - Versions for Image, Linker, Subsystem (>3), and OS
  - Loader Flags, Win32Version not used

Data Directories (within Optional Header)

- **NumberOfRvaAndSizes** is the number of data directories
  - Few PEs have a full directory
- Each **Data Directory** includes
  - An **RVA** to a specialized directory header
  - The **Size** of the header
- 16 different possible data directories, index specific
  - Export, Imports (x4), Resource, Relocation, TLS, and others

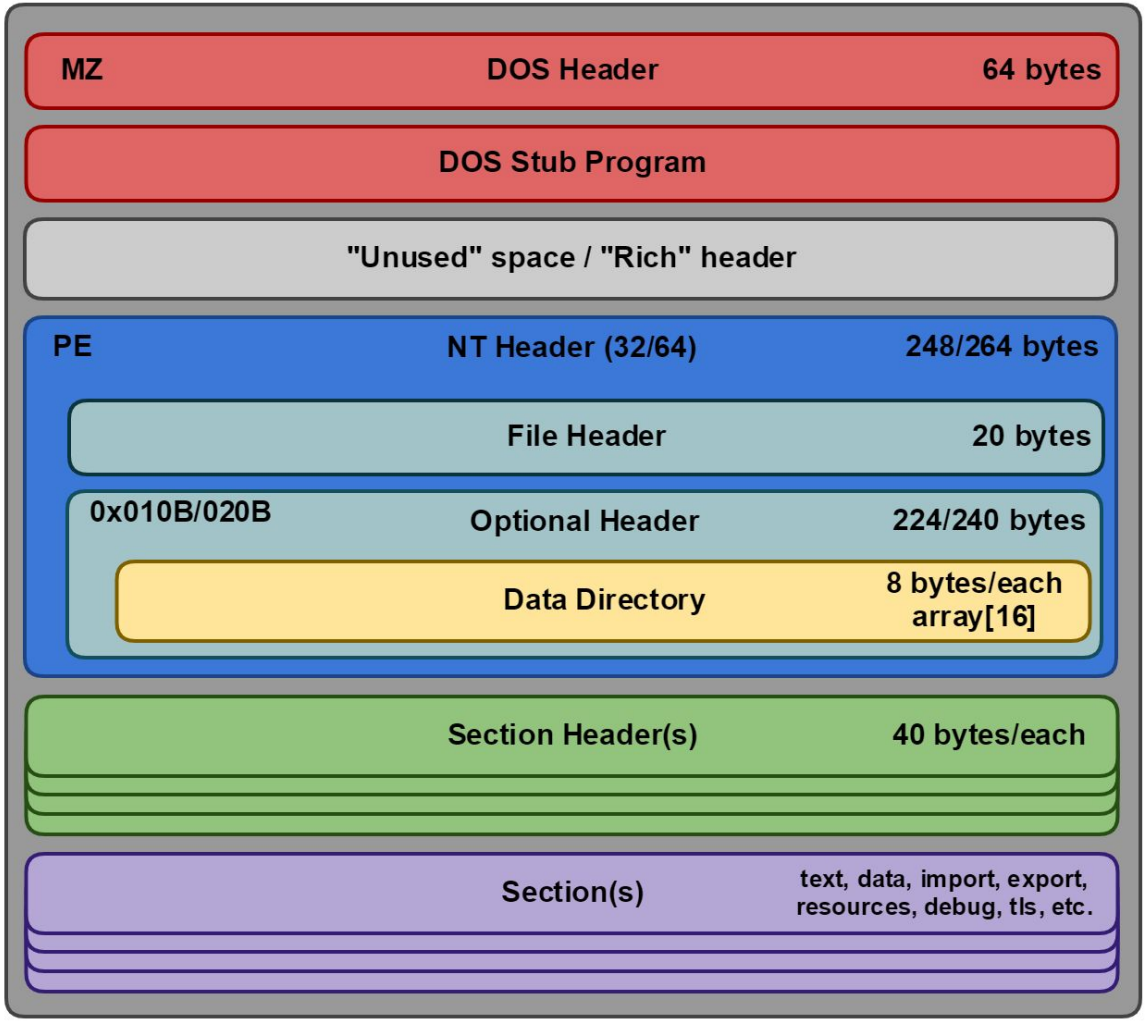
Section Header

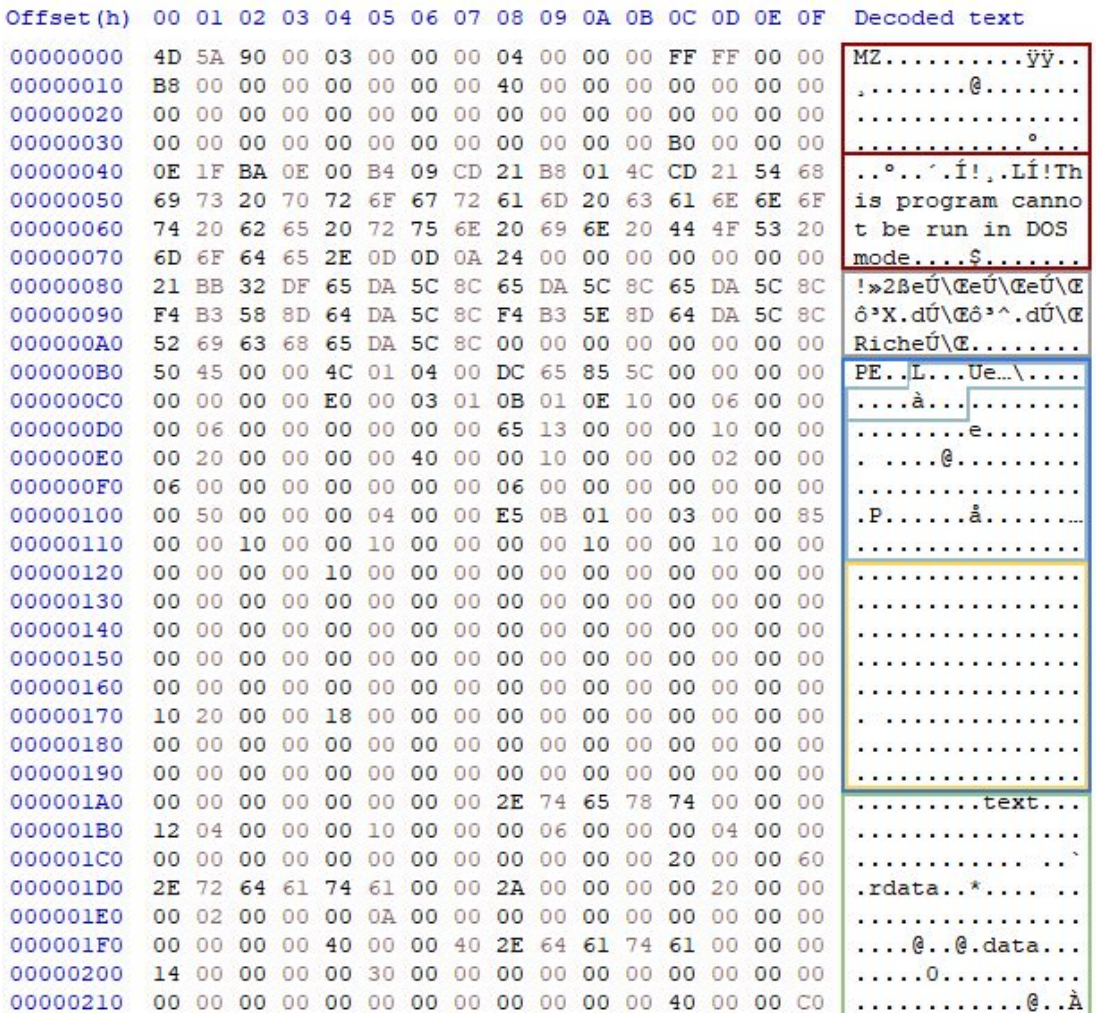
- Header for every section
  - **Pointer To Raw Data** location on disk (file offset)
  - **Size Of Raw Data** on disk
- Informs the loader where to map section into memory
  - (Relative) **Virtual Address** to map at
  - **Virtual Size** when mapped into memory
- **Characteristics**: data, code, permissions (R/W/X), etc.

Common Sections Names

.text	Executable code (often referred to as the “code” section), read/execute permission		
.bss	Uninitialized data, read/write	**.data**	Initialized data, read/write
.rdata	Initialized data, read-only	**.debug***	Debugging information, not mapped
.edata	Exported function tables, read-only	**.idata**	Imported function tables, read/write
.reloc	Relocation tables, read-only	**.rsrc**	Resource data (bitmaps, icons, etc.)
.tls	Thread Local Storage directory, read-only	**.pdata, .xdata**	Exception handling information, read-only

PE ‘sammich


MZ
DOS Header
64 bytes
DOS Stub Program
"Unused" space / "Rich" header
PE
NT Header (32/64)
248/264 bytes
File Header
20 bytes
0x010B/020B
Optional Header
224/240 bytes
Data Directory
8 bytes/each
array[16]
Section Header(s)
40 bytes/each
Section(s)
text, data, import, export,
resources, debug, tls, etc.



Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	4D	5A	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00	MZ.....ÿÿ..
00000010	B8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00	¸.......@.....
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000030	00	00	00	00	00	00	00	00	00	00	00	00	B0	00	00	00	.....°
00000040	0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68	..º..´.Í!¸.LÍ!Th
00000050	69	73	20	70	72	6F	67	72	61	6D	20	63	61	6E	6E	6F	is program canno
00000060	74	20	62	65	20	72	75	6E	20	69	6E	20	44	4F	53	20	t be run in DOS
00000070	6D	6F	64	65	2E	0D	0D	0A	24	00	00	00	00	00	00	00	mode....$.....
00000080	21	BB	32	DF	65	DA	5C	8C	65	DA	5C	8C	65	DA	5C	8C	!»2ßeÚ\ŒeÚ\ŒeÚ\Œ
00000090	F4	B3	58	8D	64	DA	5C	8C	F4	B3	5E	8D	64	DA	5C	8C	ô³X.dÚ\Œô³^.dÚ\Œ
000000A0	52	69	63	68	65	DA	5C	8C	00	00	00	00	00	00	00	00	RicheÚ\Œ.....
000000B0	50	45	00	00	4C	01	04	00	DC	65	85	5C	00	00	00	00	PE..L...Üe…\....
000000C0	00	00	00	00	E0	00	03	01	0B	01	0E	10	00	06	00	00	....à.....
000000D0	00	06	00	00	00	00	00	00	65	13	00	00	00	10	00	00	........e.....
000000E0	00	20	00	00	00	00	40	00	00	10	00	00	00	02	00	00	. ....@.....
000000F0	06	00	00	00	00	00	00	00	06	00	00	00	00	00	00	00	.....
00000100	00	50	00	00	00	04	00	00	E5	0B	01	00	03	00	00	85	.P.....å.....…
00000110	00	00	10	00	00	10	00	00	00	00	10	00	00	10	00	00	.....
00000120	00	00	00	00	10	00	00	00	00	00	00	00	00	00	00	00	.....
00000130	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000140	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000150	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000160	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000170	10	20	00	00	18	00	00	00	00	00	00	00	00	00	00	00	. .....
00000180	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000190	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000001A0	00	00	00	00	00	00	00	00	2E	74	65	78	74	00	00	00	.....text...
000001B0	12	04	00	00	00	10	00	00	00	06	00	00	00	04	00	00	.....
000001C0	00	00	00	00	00	00	00	00	00	00	00	00	20	00	00	60	....`
000001D0	2E	72	64	61	74	61	00	00	2A	00	00	00	00	20	00	00	.rdata..*.... ..
000001E0	00	02	00	00	00	0A	00	00	00	00	00	00	00	00	00	00	.....
000001F0	00	00	00	00	40	00	00	40	2E	64	61	74	61	00	00	00	....@..@.data...
00000200	14	00	00	00	00	30	00	00	00	00	00	00	00	00	00	00	.....0.....
00000210	00	00	00	00	00	00	00	00	00	00	00	00	40	00	00	C0	............@..À

DOS header

DOS stub program

“Rich” header

NT header (PE)

- File Header
- Optional Header
- Data Directory table

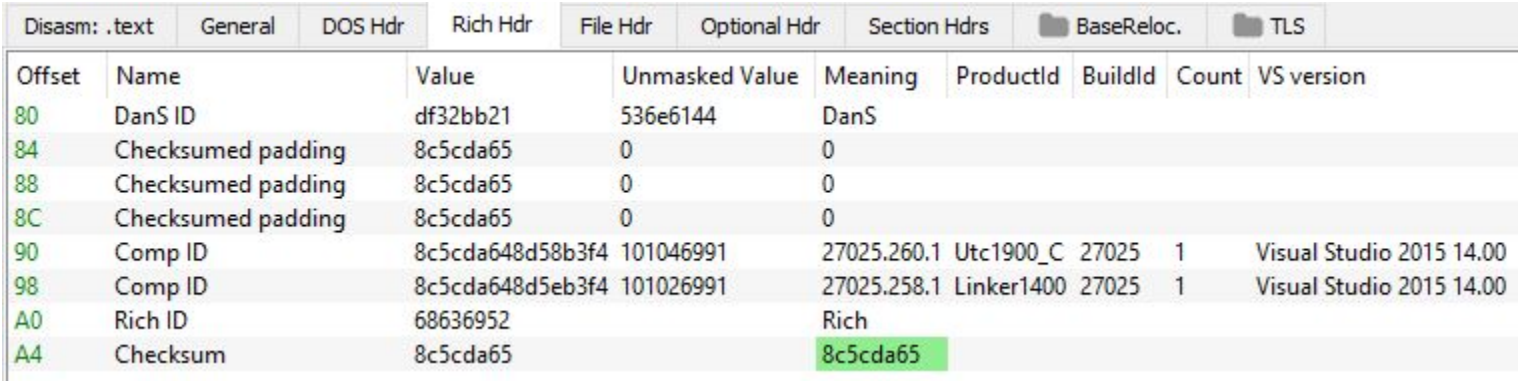
Section Header table

“Rich” Header

- This *Unused* section (per PE-COFF spec.) holds metadata
  - Microsoft Visual C++ (MSVC) compilation environment:
    - Tool versions: assembler, compiler, linker, etc.
    - Object counts (intermediate files fed to linker)
  - Created by the Linker
- XOR encoding, simple checksum
  - Easily removed, modified, or ***faked***

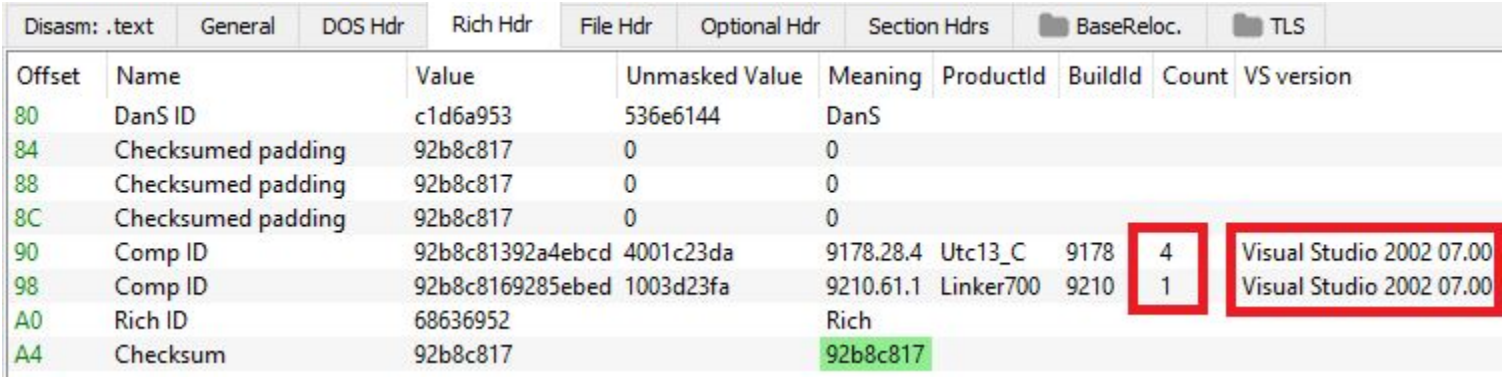
Decoded “Rich” Header (PE-bear)

Original



Offset	Name	Value	Unmasked Value	Meaning	ProductId	BuildId	Count	VS version
80	DanS ID	df32bb21	536e6144	DanS				
84	Checksumed padding	8c5cda65	0	0				
88	Checksumed padding	8c5cda65	0	0				
8C	Checksumed padding	8c5cda65	0	0				
90	Comp ID	8c5cda648d58b3f4	101046991	27025.260.1	Utc1900_C	27025	1	Visual Studio 2015 14.00
98	Comp ID	8c5cda648d5eb3f4	101026991	27025.258.1	Linker1400	27025	1	Visual Studio 2015 14.00
A0	Rich ID	68636952		Rich				
A4	Checksum	8c5cda65		8c5cda65				

“Faked”



Offset	Name	Value	Unmasked Value	Meaning	ProductId	BuildId	Count	VS version
80	DanS ID	c1d6a953	536e6144	DanS				
84	Checksumed padding	92b8c817	0	0				
88	Checksumed padding	92b8c817	0	0				
8C	Checksumed padding	92b8c817	0	0				
90	Comp ID	92b8c81392a4ebcd	4001c23da	9178.28.4	Utc13_C	9178	4	Visual Studio 2002 07.00
98	Comp ID	92b8c8169285ebed	1003d23fa	9210.61.1	Linker700	9210	1	Visual Studio 2002 07.00
A0	Rich ID	68636952		Rich				
A4	Checksum	92b8c817		92b8c817				

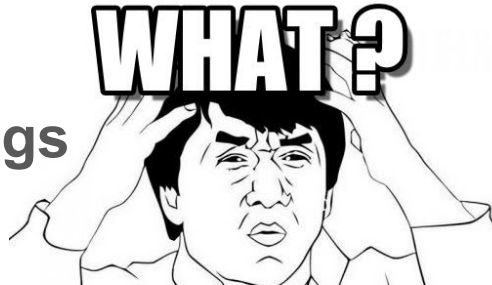
PE Data Directories

Exporting a Function from a Library

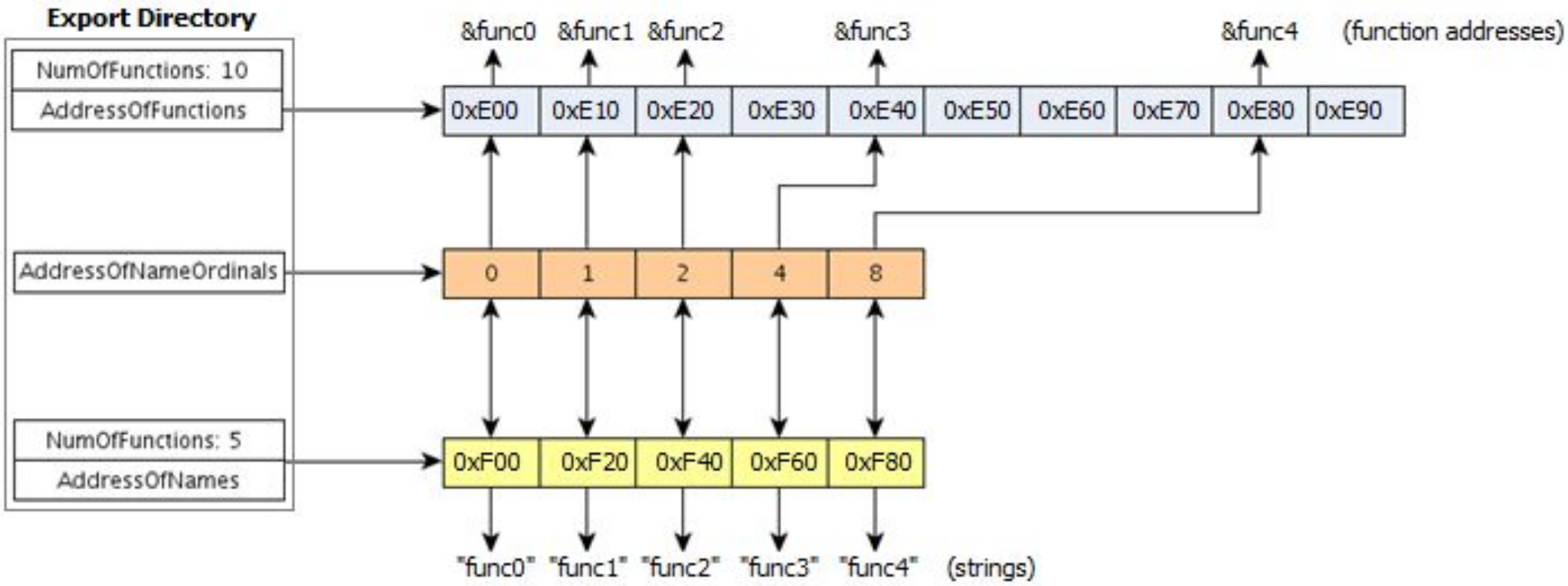
- A library *mathstuff.dll* exports functions
- `__declspec(dllexport) bool DoMathStuff(int a, int b);`
  - Alternatively, definition file (.def) with linker
- Result
  - **Export Tables** include *DoMathStuff* function
  - DLL contains a **Export Directory** referencing these tables

Export Directory and .edata Section

- Use DataDirectory[0] to locate the Export Directory
- Export Directory contains a **name** of the DLL and...
- RVAs to three other tables and their table sizes
  - **Address Table** - array of RVAs to function addresses
  - **Name Table** - array of RVAs to function names (strings)
  - **Ordinal Table** - array of 16bit ordinal values
- **RVA to exp. dir., RVA to array of RVAs to things**



Export Tables: Names, Ordinals, and Addresses



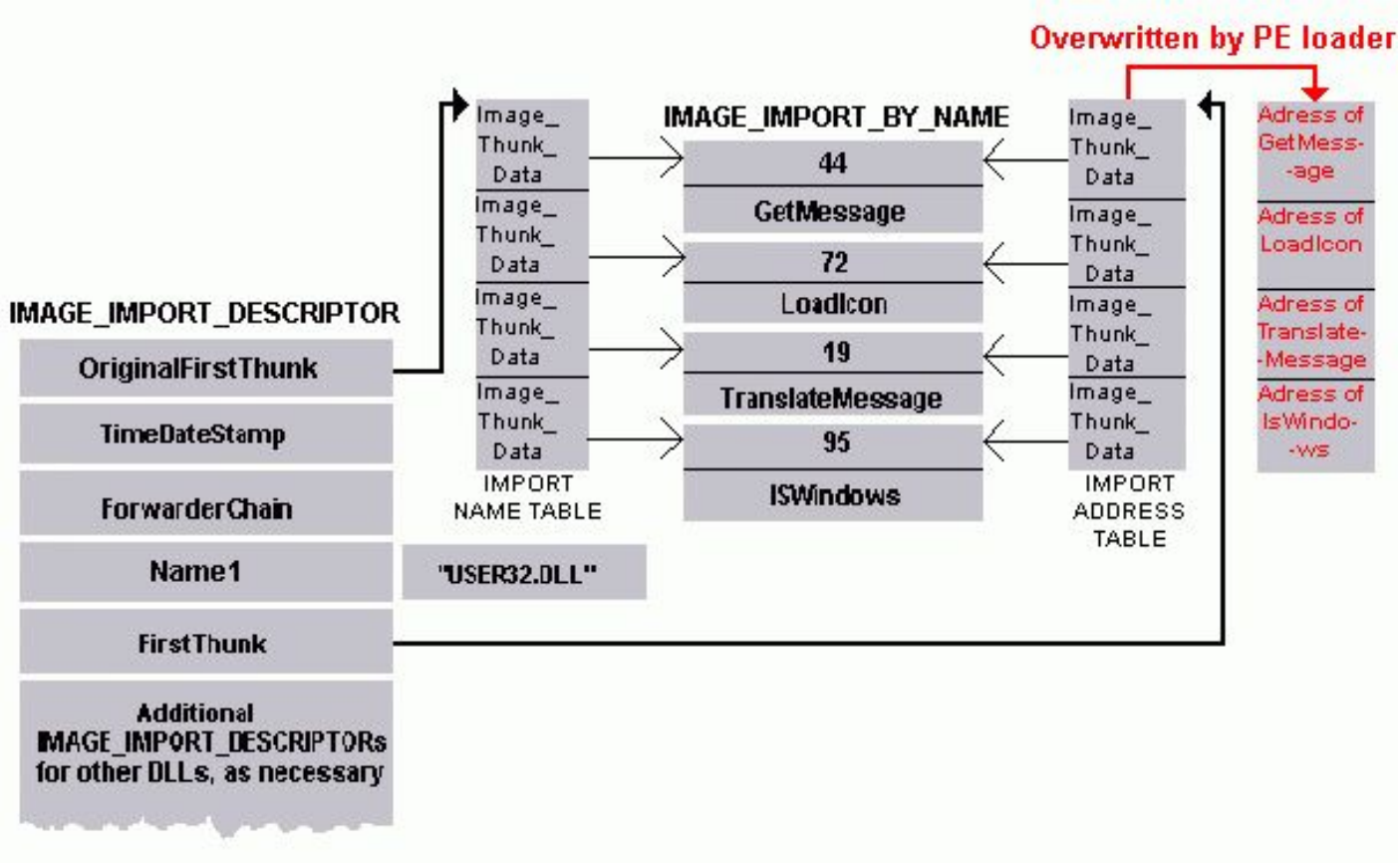
Importing a Function from a Library

- An application imports *mathstuff.dll* library
  - Imports the *DoMathStuff* function
- `__declspec(dllimport) bool DoMathStuff(int a, int b);`
  - Linker also requires an import library (.lib)
- **Import-By-Name** for *DoMathStuff* function
- Two **Import Thunks** reference this Import-By-Name
- **Import Descriptor** for *mathstuff.dll* references both Thunk lists

Import Descriptor and .idata Section

- Use DataDirectory[**1**] to locate Import Descriptor Table
- Array of Import Descriptors (one per DLL), last one is zeroed
- Descriptor includes a DLL **name**, RVAs to two thunk lists:
  - **Original First Trunk -** Import Name Table (INT)
  - **First Thunk -** Import Address Table (IAT)
    - Function addresses are overwritten by the loader

Import Descriptor and Thunks



Other Types of Imports

- **Bound Imports** - faster loading
  - DLL timestamps and function offsets written in thunks
- **Delay Imports** - compatibility
  - Imported functions are be resolved only once called
- **Forwarder Chains** - compatibility
  - Move function to another library, forward import to it
- **Ordinals** - export/import by a number instead of name

Manually Resolving Functions

- Windows API functions: https://docs.microsoft.com/
- **LoadLibraryA(fileName)**
  - Loads a DLL into the process' address space
- **GetProcAddress(library, functionName)**
  - Retrieves the address of an exported function/symbol
- Also relevant: **GetModuleHandleA()**, **FreeLibrary()**

Thread Local Storage (TLS)

- A mechanism for creating per-thread storage
- `__declspec(thread) int tls_value = 0;`
- Threads can simultaneously read/write to this variable
- Compiler and linker magic…
  - **TLS callback** functions allocate and free memory for this
  - Referenced in **TLS directory** (located by DataDirectory[9])

More Data Directories

- **Resource**
  - Dialogs, media files, string tables, etc.
- **Security**
  - Authenticode signatures (signed binary)
- **Relocation**
  - “Fix-up” tables if loaded at different image base

More Data Directories Cont.

- **Exception**
  - SEH handlers and unwinding information
- **Debug**
  - Symbols or location of symbol database
- **Load Config**
  - Hot patching, Control Flow Guard (CFG), Shadow Stack?
- And more...

Enough with the Structures!!!

- Still 60+ structures and 200+ macros to cover…
  - See “Image Format” in the ***winnt.h*** header file
  - Included with the Microsoft Windows Platform SDK
- Microsoft PE and COFF Specification (73 pages)
- PE format visualizations:
  - Corkami’s PE Posters: 101 (PNG) and 102 (PDF)
  - Ero Carrera's PE File Format Graphs

Parsing PE Files

- Avoid writing your own PE parser, it's hard to get right:
  - CVE-2016-10402, CVE-2016-5308, CVE-2016-2208, CVE-2013-3900, CVE-2012-2273, CVE-2010-1640, CVE-2007-0125, CVE-2006-1614, CVE-2005-0249, ...
- Use Microsoft's DbgHelp Image APIs instead
- Even better, use Ero Carrera's Python module, ***pefile***: https://github.com/erocarrera/pefile

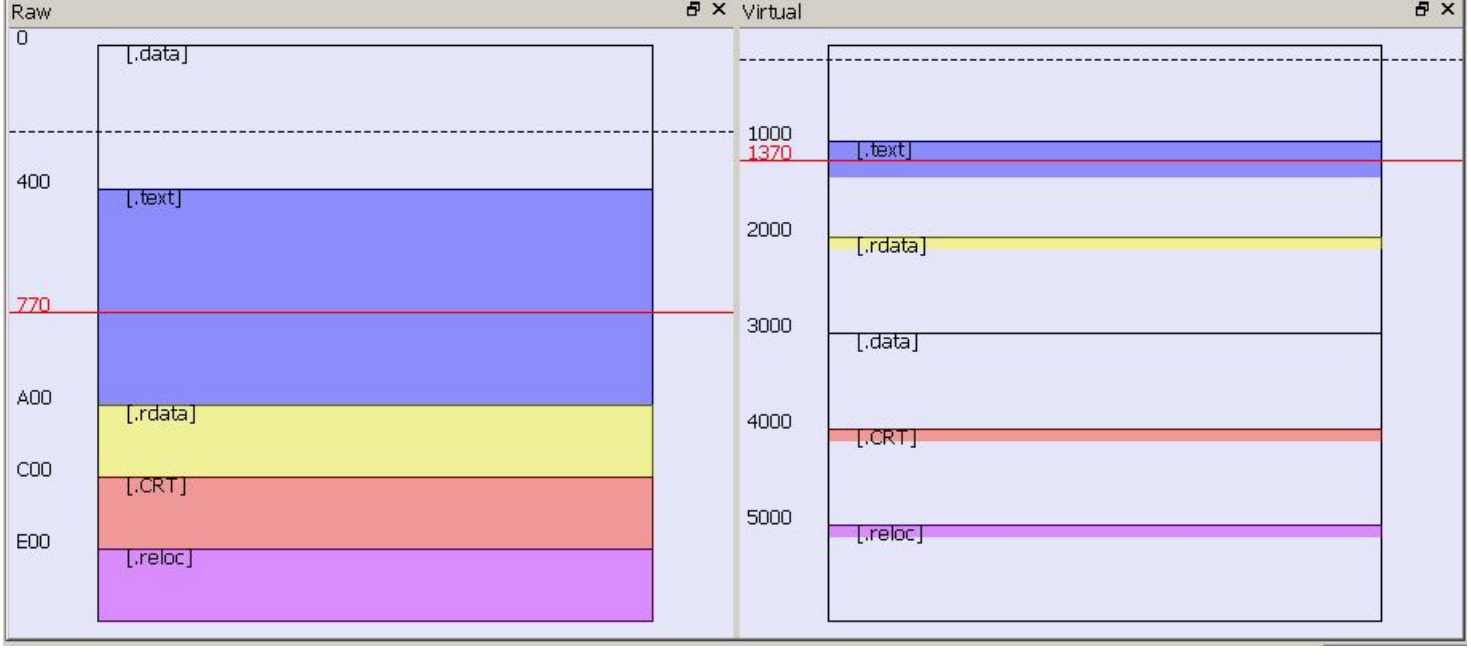
PE Execution

Executing/Loading a PE

- Image Loader lives in *NTDLL.DLL* (Ldr functions)
  a. Loader maps PE headers at determined **Image Base**
  b. PE headers are parsed
  c. Sections mapped into process address space
  d. Import directories and export tables used to resolve functions
     - If a new dependency is needed, load and proceed to (a)
- Execute PE entry point

Section Mapping Visualization (PE-bear)

Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of Reloc.	Num. of Linenum.
.text	400	600	1000	412	60000020	0	0	0
.rdata	A00	200	2000	2A	40000040	0	0	0
.data	0	0	3000	14	C0000040	0	0	0
.CRT	C00	200	4000	C	40000040	0	0	0
.reloc	E00	200	5000	40	42000040	0	0	0



Relative Virtual Addressing (RVA)

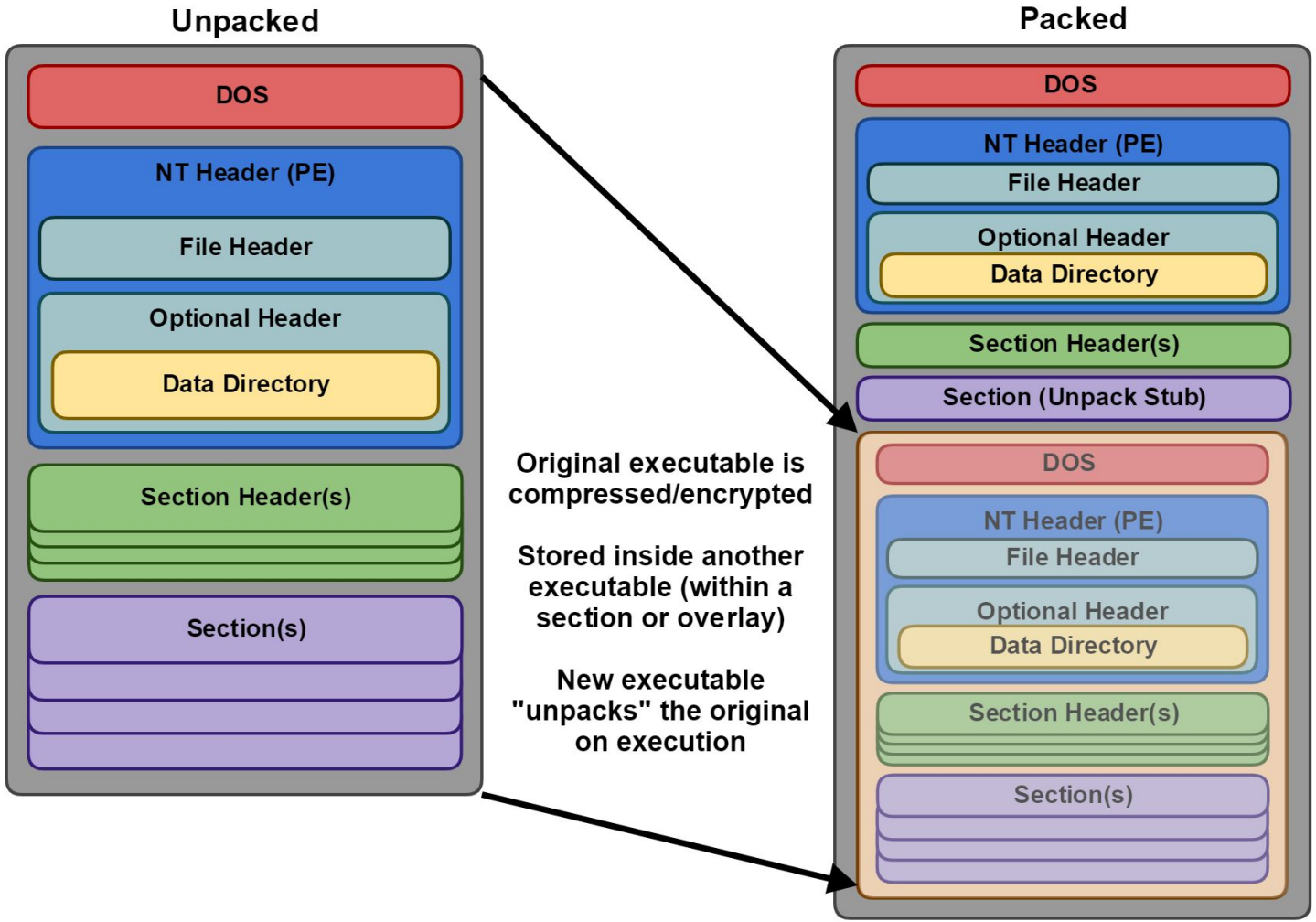
- Addresses are relative to the base address
- `VirtualAddress = BaseVA + RelativeVA`
  - **BaseVA** is the base virtual address the PE is mapped at
  - **RelativeVA** is the offset (the RVA)
  - **VirtualAddress** is the location in the process' address space

Determining the Image Base

- Optional Header contains a preferred **Image Base**
- Loader will map PE to a different base address
  - Address Space Layout Randomization, added in Vista
  - Collision with an already mapped image
- Exceptions to relocating
  - No dynamic base flag (“ASLR” flag) in DLL Characteristics
  - No relocation tables

00400000	00001000	hello_stealth_original.exe		IMG	-R---	ERWC-
00401000	00001000	".text"	Executable code	IMG	ER---	ERWC-
00402000	00001000	".rdata"	Read-only initialized data	IMG	-R---	ERWC-
00403000	00001000	".data"	Initialized data	IMG	-RW--	ERWC-
00404000	00001000	".tlsc"	Thread-local storage	IMG	-R---	ERWC-
00410000	000C5000	\Device\HarddiskVolume4\Windows\S		MAP	-R---	-R---
00590000	00007000			PRV	-RW--	-RW--
00597000	00009000	Reserved (00590000)		PRV		-RW--
00700000	00005000			PRV	-RW--	-RW--
00705000	000FB000	Reserved (00700000)		PRV		-RW--
766F0000	00001000	kernelbase.dll		IMG	-R---	ERWC-
766F1000	001C3000	".text"	Executable code	IMG	ER---	ERWC-
768B4000	00004000	".data"	Initialized data	IMG	-RW--	ERWC-
768B8000	00006000	".idata"	Import tables	IMG	-R---	ERWC-
768BE000	00001000	".didat"		IMG	-R---	ERWC-
768BF000	00001000	".rsrc"	Resources	IMG	-R---	ERWC-
768C0000	0002A000	".reloc"	Base relocations	IMG	-R---	ERWC-
77710000	00001000	kernel32.dll		IMG	-R---	ERWC-
77711000	0000F000	Reserved (77710000)		IMG		ERWC-
77720000	00064000	".text"	Executable code	IMG	ER---	ERWC-
77784000	0000C000	Reserved (77710000)		IMG		ERWC-
77790000	0002F000	".rdata"	Read-only initialized data	IMG	-R---	ERWC-
777BF000	00001000	Reserved (77710000)		IMG		ERWC-
777C0000	00001000	".data"	Initialized data	IMG	-RW--	ERWC-
777C1000	0000F000	Reserved (77710000)		IMG		ERWC-
777D0000	00001000	".rsrc"	Resources	IMG	-R---	ERWC-
777D1000	0000F000	Reserved (77710000)		IMG		ERWC-
777E0000	00005000	".reloc"	Base relocations	IMG	-R---	ERWC-
777E5000	0000B000	Reserved (77710000)		IMG		ERWC-
77920000	00009000			IMG	-R---	ERWC-
77930000	00001000	ntdll.dll		IMG	-R---	ERWC-
77931000	0011C000	".text"	Executable code	IMG	ER---	ERWC-
77A4D000	00001000	"RT"		IMG	ER---	ERWC-
77A4E000	00006000	".data"	Initialized data	IMG	-RW--	ERWC-
77A54000	00003000	".mrdata"		IMG	-R---	ERWC-
77A57000	00001000	".00cfg"		IMG	-R---	ERWC-
77A58000	0006F000	".rsrc"	Resources	IMG	-R---	ERWC-
77AC7000	00005000	".reloc"	Base relocations	IMG	-R---	ERWC-
7FE50000	00005000			MAP	-R---	-R---

Common PE Obfuscation Techniques


Unpacked
DOS
NT Header (PE)
File Header
Optional Header
Data Directory
Section Header(s)
Section(s)
Original executable is compressed/encrypted
Stored inside another executable (within a section or overlay)
New executable "unpacks" the original on execution
Packed
DOS
NT Header (PE)
File Header
Optional Header
Data Directory
Section Header(s)
Section (Unpack Stub)
DOS
NT Header (PE)
File Header
Optional Header
Data Directory
Section Header(s)
Section(s)

Purpose of PE Packers

- Hides “on-disk” code, unpacks at runtime
  - Antivirus/sandbox may unpack to analyze
- Commercial packers often include anti-piracy measures
  - Protection systems for games or software wrap executables
- Malware is often packed with a custom packer
  - Usually includes various countermeasures

Abusing the PE Format

- Many obfuscation techniques, some common ones:
- Dynamically resolving functions at runtime
  - Hides imports
- Encoding or encrypting strings
  - Easily readable otherwise
- Thread local storage (TLS) callbacks
  - Alternative entry point

Obfuscation Techniques

- These techniques attempt to mask the PE’s functionality
- Reduce signatures that security products use for detection
  - Often creating additional signatures
- Increase analysis complexity
  - **Wastes a reverse engineer’s time**

Demo

“Tell me and I forget,

teach me and I may remember,

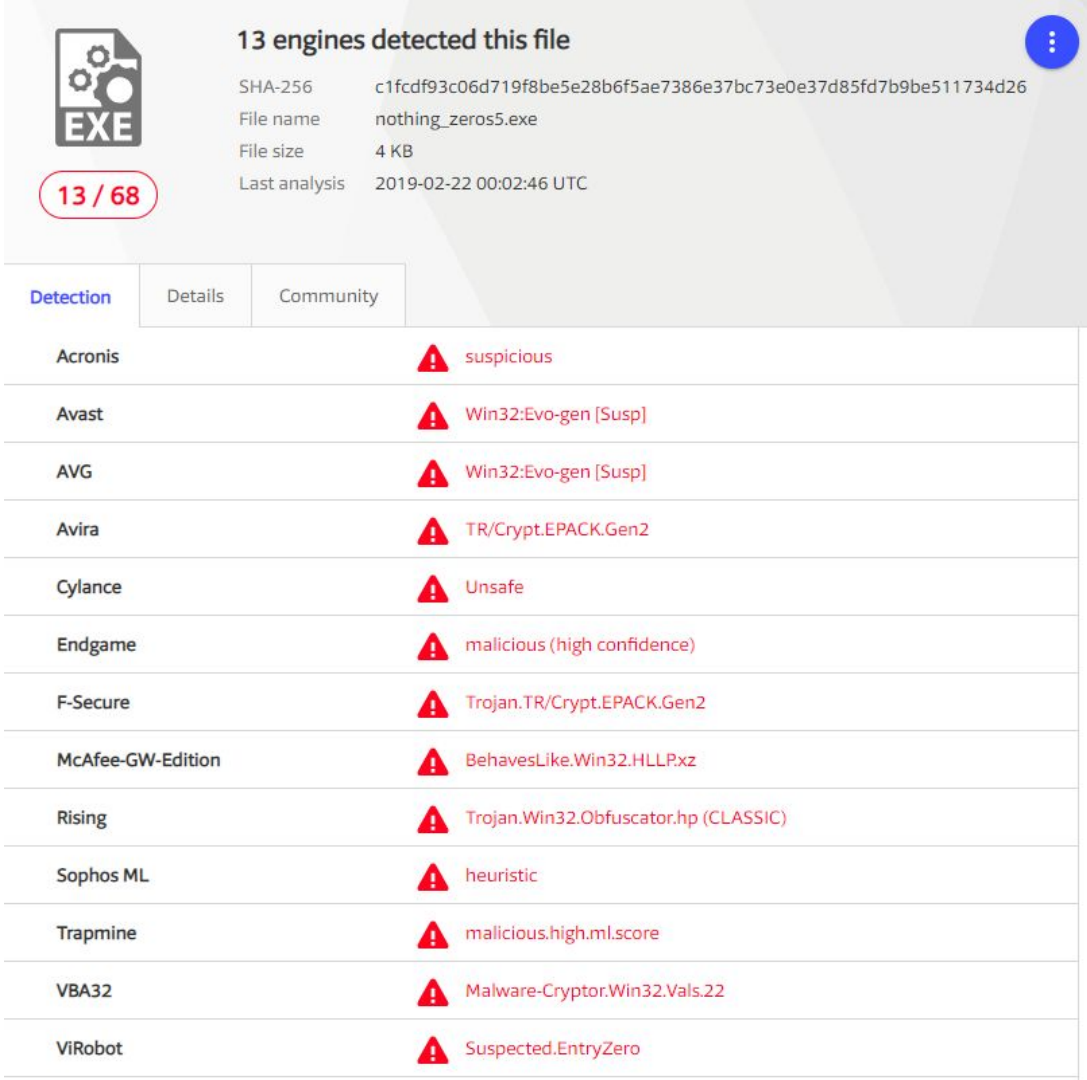
involve me and I learn.”

Simple Obfuscation Demo

- Write the simplest application
  - No malicious activity (no touching files, registry, etc.)
  - No executable packer
  - Just prints “hello, world” to standard output
- Goal:
  - Have it detected as malicious (false-positive)

VirusTotal Results

- Submitted *hello_zeros.exe*
- 13 antivirus software products suspected this simple “hello world” application is malicious
- No packer was used



hello_zeros.exe

- Source File: part3_obfus/hello_stealth.c
- Objectives:
  - Identify entry point, imports, sections
  - Identify any strings or cryptographic signatures
  - Fully reverse engineer the executable in IDA
    - How are imported functions resolved?
    - Debugging can save reversing time
  - Bonus: zero out TLS directory, still executes - how?

infector.exe

- Source File: part4_infect/infector.c
- Infection process:
  - Locate a code cavity in target executable
  - Write target’s OEP into stub code
    - After stub executes, returns to OEP
  - Write stub into code cavity, adjust section headers if needed
  - Set PE’s new entry point to the inserted stub

Tools Used In Demo

- PE-bear - PE file format viewer/editor (by @hasherezade)
- IDA - Industry standard disassembler ($$$, freeware version)
  - Alternatives: Cutter/Radare2, BinaryNinja ($), Hopper ($)
- PEiD - PE and packer identification
- SysinternalsSuite - Windows troubleshooting tools
- x32dbg/x64dbg - Assembly-level debugger for Windows

Wrapping Up

Where Do I Start?

- Learn low-level programming languages
  - e.g. Assembly (arch. dependent) and C
- Hardware architecture and operating system internals
  - Will make reverse engineering easier
- **The most important thing is to just start**
  - Your knowledge will progress as you read information to understand specific APIs, instructions, and techniques

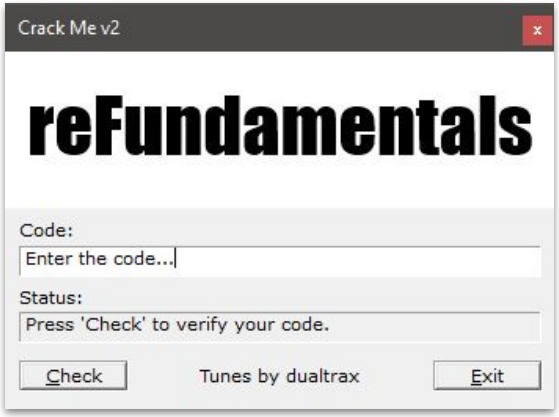
Practicing Reverse Engineering

- Crack-mes - crack them (patching, crypto challenges, etc.)
- Unpack-mes - unpack them
- Malware samples - all of the above
  - Exercise caution...

crackme1.exe

- Console-based
- Hints:
  - Code validation is base on the entered name
  - Note *GetProcAddress* imported from kernel32.dll
  - Locate validation function by debugging or following references to strings

crackme2.exe



- GUI-based, packed
- Bonus, plays chiptunes while reversing
- Hints:
  - Code validation is based on values from certain WinAPI calls
  - Identify cryptographic signatures (e.g. *findcrypt*, *PEiD KANAL*)
  - *User32!GetDlgItemTextA* retrieves text from an edit control

Solve with Gov’t Tools

- Ghidra - NSA’s disassembler
  - https://ghidra-sre.org/
- CyberChef - GCHQ’s web-app for encryption, encoding, etc.
  - https://gchq.github.io/CyberChef/.

More Resources

- RCE Labs https://www.begin.re/ by @OphirHarpaz
- Malware Labs http://malwareunicorn.org/ by @malwareunicorn
- ARM Labs https://azeria-labs.com/ by @Fox0x01 (Azeria)
- OpenAnalysis https://oalabs.openanalysis.net/
  - OALabs Live Youtube channel
- https://github.com/apodlosky/reFundamentals/RESOURCES.md
  - List of articles, books, software, etc. I find useful

Fin.

Questions?

Slides, demos, and source code are available at:

https://github.com/apodlosky/reFundamentals/