Reverse Engineering and
Malware Analysis Fundamentals

Portable Executable Format

Adam Podlosky
@apodlosky

MacEwan Technology Resource Policy

- Use of these resources for disruptive, discriminatory, illegal, harassing or malicious purposes is strictly prohibited.
- Users will comply with all copyright and licensing agreements.
- Users shall not attempt to circumvent data protection schemes or uncover security vulnerabilities.
- **NO protection circumvention, NO piracy, NO malware**
- https://www.macewan.ca/contribute/groups/public/documents/policy/use_itresources_standard.pdf

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 an understanding of pointers will help
- I do not expect knowledge of x86 assembly but it will come up

Reverse Engineering

- Old practice in most other fields, e.g. copyrights exist
- Hardware reverse engineering–PCB design, emulators
- Software reverse engineering
  - Examining closed-source software, firmware images
  - Device drivers on open-source operating systems
  - Interoperability with proprietary file formats or network protocols

Careers in Reverse Engineering

- Anti-cheat development for games
- Software protection (anti-piracy) development
- Information security
  - Digital Forensics and Incident Response (DFIR)
  - Malware analysis and security research
  - Vulnerability (exploit) development
- Working for Huawei in “R&D”...

Legality of Reverse Engineering

- Reverse engineering proprietary technology can violate Canadian intellectual property laws
- End User License Agreements (EULAs) for commercial products or closed-source software often forbid reverse engineering their technology
- The Copyright Modernization Act of Canada introduced similar protections as the US Digital Millennium Copyright Act (DMCA)
- Consult a lawyer, I am most certainly not one

PE

R.C.E. begins with the binary format...

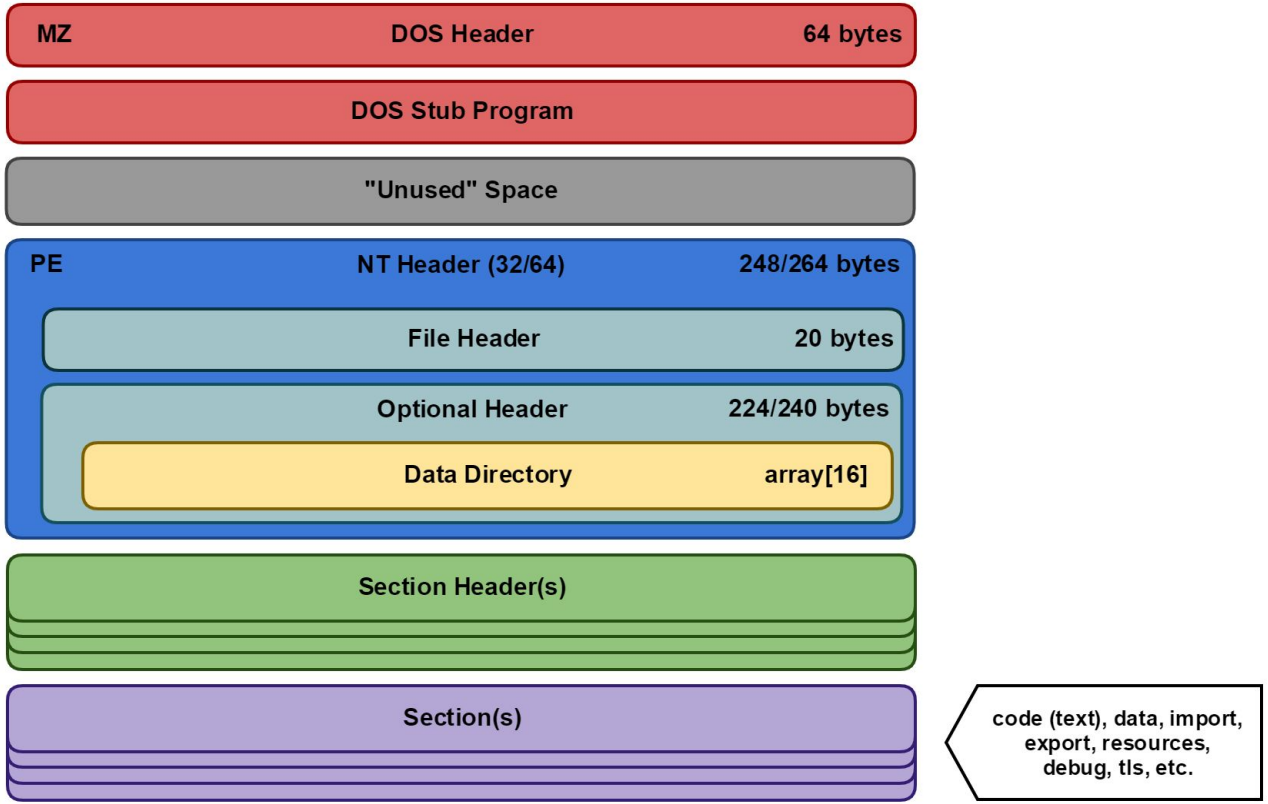
What is a Portable Executable?

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

History of the Portable Executable

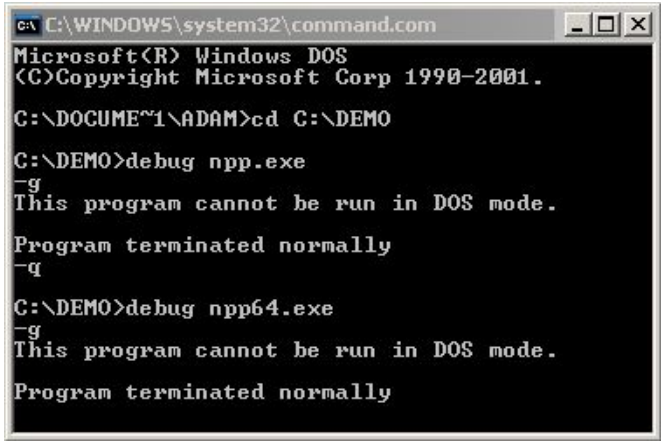
- **MZ** format (16-bit) in MS-DOS (~1980? Before my time…)
  - MZ executables used the .EXE file extension
  - Execution compatibility removed in Windows x64
- **PE32** (32-bit) introduced with Windows NT 3.1 in 1993
- **PE32+** (64-bit) originally for DEC Alpha CPUs, never released
  - First x86-64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

Headers, Stubs, Directories and Sections



DOS Stub

- All PEs contain 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

PE Format Defined w/C Structures

- TODO: diagram, but we're all programmers here…

- Also, padding/packing, bit fields, endianness, unnamed unions, zero-length (flexible) arrays

DOS Header

```
#define IMAGE_DOS_SIGNATURE  0x5A4D  // MZ

typedef struct {
    UINT16  e_magic;    // Magic number
    UINT16  e_cblp;
    UINT16  e_cp;
    UINT16  e_crlc;
    // ...  (14 fields omitted)
    INT32   e_lfanew;   // Offset of NEW header
} IMAGE_DOS_HEADER;
```

- `e_magic` contains the ‘MZ’ value
- `e_lfanew` is the offset to the NT header

NT Header (32/64)

```
#define IMAGE_NT_SIGNATURE   0x00004550  // PE00

typedef struct {
    UINT32                  Signature;
    IMAGE_FILE_HEADER       FileHeader;      // Headers in a header
    IMAGE_OPTIONAL_HEADER32 OptionalHeader;  // ^
} IMAGE_NT_HEADERS32;

typedef struct {
    UINT32                  Signature;
    IMAGE_FILE_HEADER       FileHeader;
    IMAGE_OPTIONAL_HEADER64 OptionalHeader;
} IMAGE_NT_HEADERS64;
```

File Header

```
typedef struct {
    UINT16  Machine;                // The target architecture
    UINT16  NumberOfSections;       // Number of Section Header structures
    UINT32  TimeDateStamp;
    UINT32  PointerToSymbolTable;
    UINT32  NumberOfSymbols;
    UINT16  SizeOfOptionalHeader; // Indicates if Optional Header is present
    UINT16  Characteristics;        // Flags that can influence the loader
} IMAGE_FILE_HEADER;
```

- File Header contains several values that are required to interpret other header structures

Optional Header (32/64)

```
typedef struct {
    UINT16   Magic;                  // 0x10B for PE32, 0x20B for PE32+
    // ...   (versions omitted)
    UINT32   SizeOfCode;             // Size of code (.text) section
    // ...   (sizes omitted)
    UINT32   AddressOfEntryPoint;    // RVA to begin execution at once loaded
    UINT32   BaseOfCode;             // RVA to code section
    UINTPTR  ImageBase;              // Preferred base virtual address to map at (32/64)
    UINT32   SectionAlignment;       // Alignment of sections in memory, page-size (4K usually)
    UINT32   FileAlignment;          // Alignment of PE on disk, usually 512
    // ...   (versions omitted)
    UINT32   SizeOfImage;
    UINT32   SizeOfHeaders;          // Size of all headers, including section header table
    UINT32   CheckSum;               // File checksum, only verified for drivers
    UINT16   Subsystem;              // Subsystem responsible to run this executable
    // ...   (sizes and flags omitted)
    UINT32   NumberOfRvaAndSizes;    // Determine size of DataDirectory[] (not always 16)
    IMAGE_DATA_DIRECTORY DataDirectory[16]; // Data directories used to locate sections
} IMAGE_OPTIONAL_HEADERxx; // 32/64 dependent
```

Array of Data Directory Structures

```
typedef struct {
    UINT32  VirtualAddress;  // RVA of index-specific directory structure
    UINT32  Size;
} IMAGE_DATA_DIRECTORY;
```

- Each specific entry locates a specialized directory structure
  - Exports, Imports (Bound, IAT, Delay), Resources
  - Exceptions, Security, Relocations, Debug Info
  - TLS, Load Config, COM Runtime, Global Ptr (Itanium-only)
- Most PEs do not include every directory type

Section Header

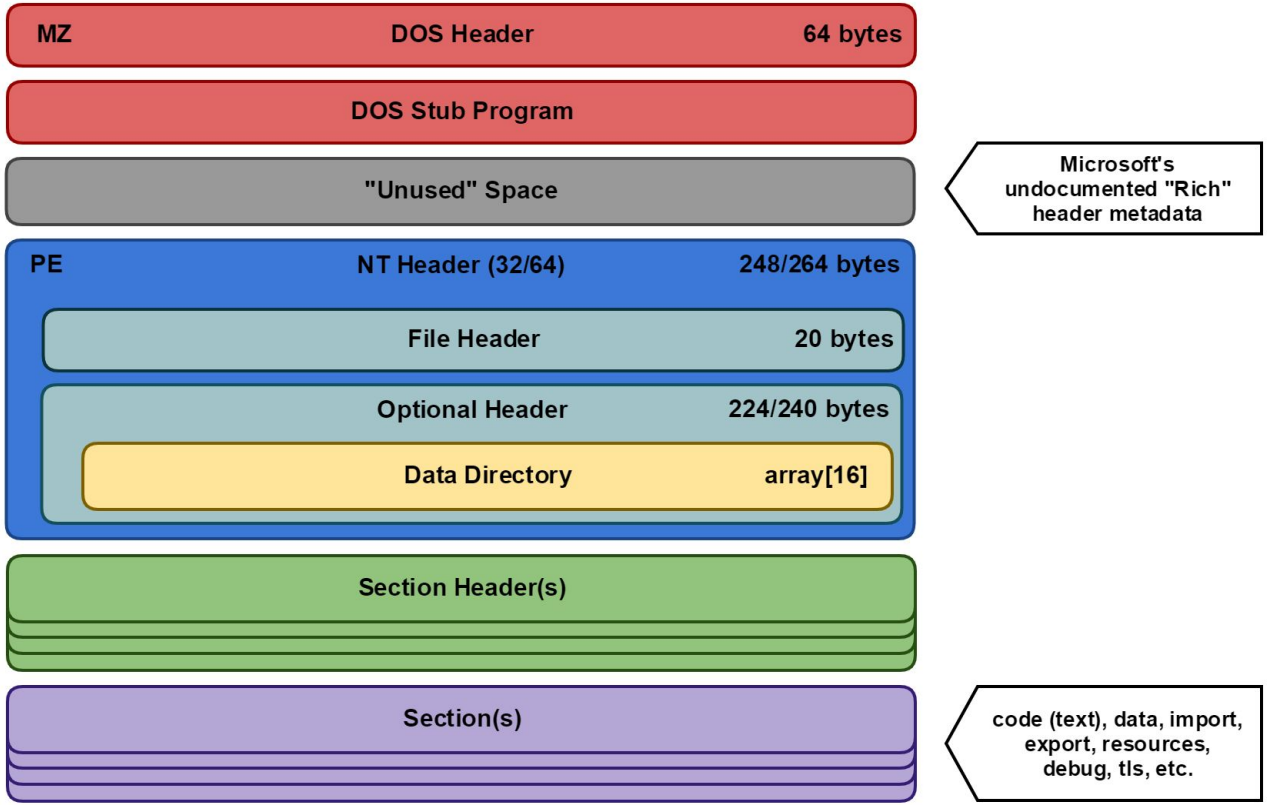
```
typedef struct {
    CHAR    Name[8];                // Short name describing the section
    union {
       UINT32  PhysicalAddress;
       UINT32  VirtualSize;         // Size of section in memory
    } Misc;
    UINT32  VirtualAddress;         // RVA of section when mapped into memory
    UINT32  SizeOfRawData;          // Size of section on disk (aligned)
    UINT32  PointerToRawData;       // File offset of section
    UINT32  PointerToRelocations;
    UINT32  PointerToLinenumbers;
    UINT16  NumberOfRelocations;
    UINT16  NumberOfLinenumbers;
    UINT32  Characteristics;        // Flags and memory page permissions
} IMAGE_SECTION_HEADER;
```

- Informs the loader where to map sections into memory

Enough with the Structures!!!

- Still 70+ structures and 200+ macros to cover…
  - Roughly 2,400 lines worth
  - See “Image Format” in the ***winnt.h*** header file
  - Included with the Microsoft Windows Platform SDK
- Microsoft PE and COFF Specification
- Corkami’s PE Posters: 101 (PNG) and 102 (PDF)
- And Ero Carrera's PE File Format Graphs

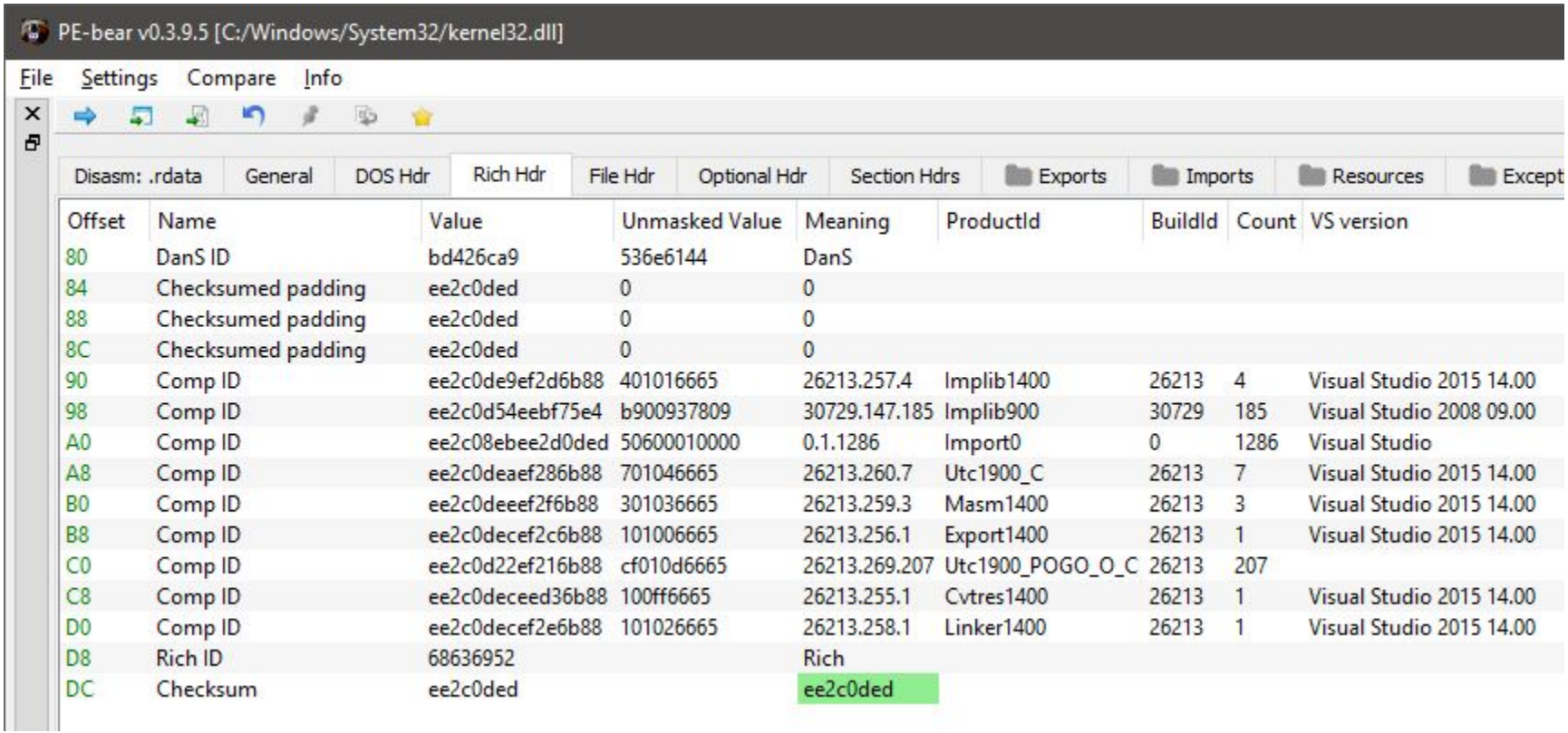
Review of Locating Headers



“Rich” Header - MSVC Metadata

- *Unused* section from the PE-COFF spec. holds metadata
- This undocumented header contains a record of the compilation environment: tool versions, object counts, etc.
- Linker gathers *@comp.id* symbols from intermediate files
- Simple XOR encoding; easily removed, modified, or ***faked***
- http://bytepointer.com/articles/the_microsoft_rich_header.htm

“Rich” Header Viewed from PE-bear



Offset	Name	Value	Unmasked Value	Meaning	ProductId	BuildId	Count	VS version
80	DanS ID	bd426ca9	536e6144	DanS				
84	Checksumed padding	ee2c0ded	0	0				
88	Checksumed padding	ee2c0ded	0	0				
8C	Checksumed padding	ee2c0ded	0	0				
90	Comp ID	ee2c0de9ef2d6b88	401016665	26213.257.4	Implib1400	26213	4	Visual Studio 2015 14.00
98	Comp ID	ee2c0d54eebf75e4	b900937809	30729.147.185	Implib900	30729	185	Visual Studio 2008 09.00
A0	Comp ID	ee2c08ebee2d0ded	50600010000	0.1.1286	Import0	0	1286	Visual Studio
A8	Comp ID	ee2c0deaef286b88	701046665	26213.260.7	Utc1900_C	26213	7	Visual Studio 2015 14.00
B0	Comp ID	ee2c0deeef2f6b88	301036665	26213.259.3	Masm1400	26213	3	Visual Studio 2015 14.00
B8	Comp ID	ee2c0decef2c6b88	101006665	26213.256.1	Export1400	26213	1	Visual Studio 2015 14.00
C0	Comp ID	ee2c0d22ef216b88	cf010d6665	26213.269.207	Utc1900_POGO_O_C	26213	207	
C8	Comp ID	ee2c0deceed36b88	100ff6665	26213.255.1	Cvtres1400	26213	1	Visual Studio 2015 14.00
D0	Comp ID	ee2c0decef2e6b88	101026665	26213.258.1	Linker1400	26213	1	Visual Studio 2015 14.00
D8	Rich ID	68636952		Rich				
DC	Checksum	ee2c0ded		ee2c0ded				

Common Names for Sections

- Defined by Data Directory and Section Header, not by name

.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.)
.pdata, .xdata	Exception handling information, read-only	Other section names for .NET CLR code and many more for Itanium support	

Process Creation and Image Loader (Simplified)

- Application calls a CreateProcess API, path to a PE image
- Kernel creates structures and objects for process and thread
- User-mode execution begins at *NTDLL!RtlUserThreadStart*
- More user-mode initialization (e.g. PEB, TEB structures)
- Image Loader (Ldr* functions) parses and maps the PE file and its dependencies into the process address space
- Breakpoint if debugger attached, execute PE entry point

Loader - Mapping a PE into Memory

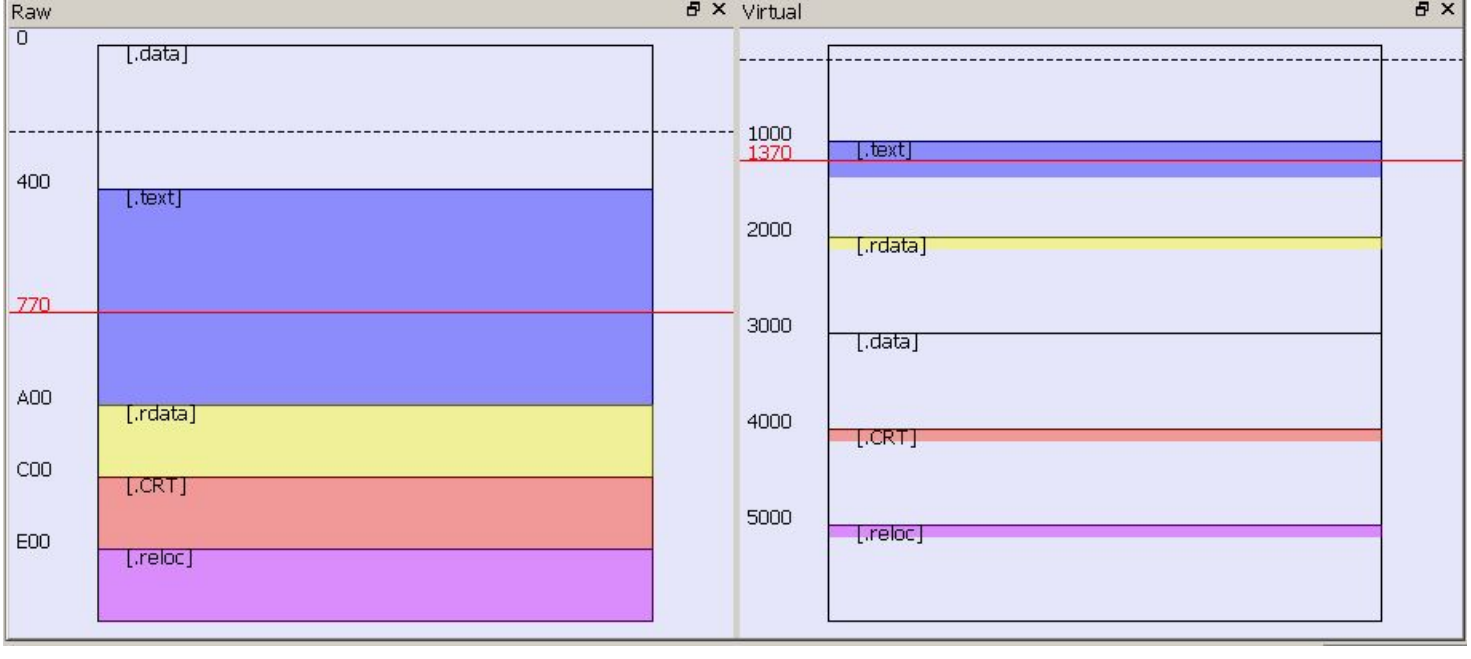
1. ASLR randomizes base address (ex. system DLLs set at boot)
2. Loader walks section headers, maps sections at the virtual address specified by the RVA, sets page permissions
    - Base relocations performed if needed
3. Loader parses import table to determine dependencies
    - For each new dependency, proceed to step 1
4. Resolves function imports and overwrites addresses in IAT

Relative Virtual Addresses (RVAs)

- `VirtualAddress = BaseVA + RelativeVA`
- Where *BaseVA* is the base virtual address the PE is mapped at
- Optional Header contains a preferred *ImageBase* but the *Image Loader* will map the PE to a different address
  - Address Space Layout Randomization added in Vista
  - Exceptions: no relocation section, no dynamic base flag

Loader - Mapped Visualization in PE-bear

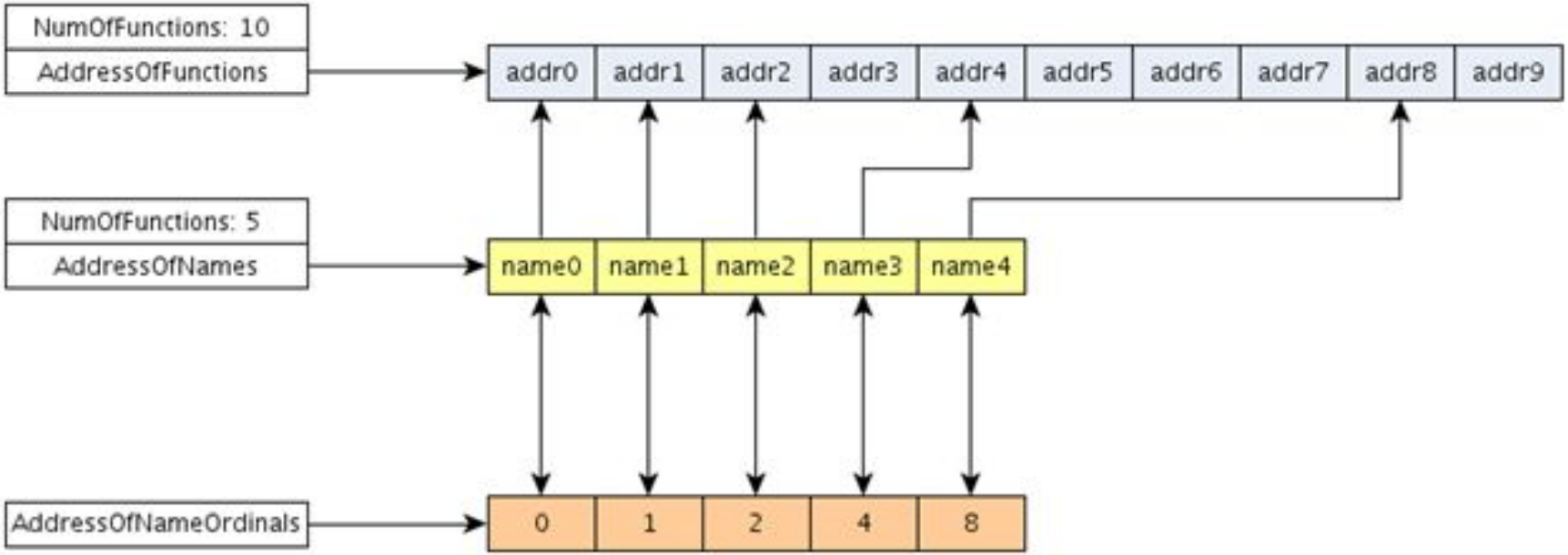
Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of Reloc.	Num. of Linenum.
.text	400	600	1000	412	60000020	0	0	0
.rdata	A00	200	2000	2A	40000040	0	0	0
.data	0	0	3000	14	C0000040	0	0	0
.CRT	C00	200	4000	C	40000040	0	0	0
.reloc	E00	200	5000	40	42000040	0	0	0



Export Section (.edata)

- *IMAGE_EXPORT_DIRECTORY* defines RVAs to arrays of function name RVAs, ordinals, and function address RVAs
- Functions can be exported by name and/or an ordinal number
- The same function can be exported with multiple names but only a single ordinal number
- Exported functions can be forwarded to another library

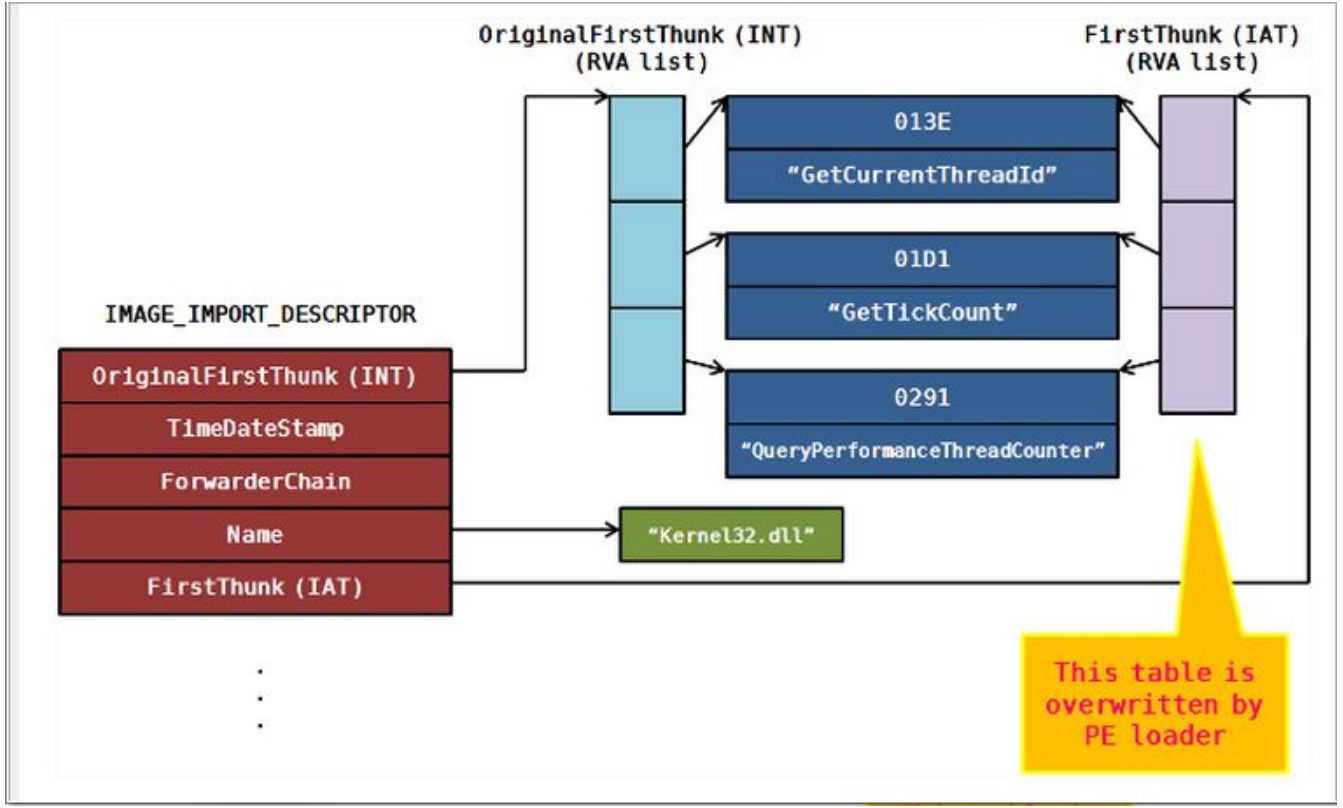
Tables in the Export Directory



Import Section (.idata)

- Array of *IMAGE_IMPORT_DESCRIPTOR* structures
  - One per DLL, terminated by a zeroed out structure
- Each descriptor defines an RVA to the DLL name and RVAs to two arrays of *IMAGE_THUNK_DATA* structures
  - Import Address Table (IAT) and Import Name/Lookup Table
  - **IAT function addresses are overwritten by the Loader**
- Also, Bound Imports, Delay Imports, and Forwarder Chains (old)

Import Descriptor and Thunks



Other Directories and Sections

- Resources section contains dialogs, media files, string tables, and occasionally used to store additional PE files
- TLS directory defines callbacks to setup and cleanup per-thread resources, doesn’t work well for libraries
- Relocation tables, basically a list of “fix-ups” for the Image Loader if the PE is loaded an address other than ImageBase
- Exception tables, debug sections, and more

Demo/Tutorial

“Tell me and I forget,

teach me and I may remember,

involve me and I learn.”

Always Use a Virtualized OS

- Do NOT analyze executables on your host operating system
  - Setup and secure a virtual machine (see resources)
  - Also, **REMnux** (“Kali” for malware): https://remnux.org
- The provided binaries were written and compiled by myself
  - They are **NOT** malicious
  - Endpoint/antivirus software may display false-positive alerts

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
- ResourceHacker - View and edit PE resources
- SysinternalsSuite - Windows troubleshooting tools
- x32dbg/x64dbg - Great assembly-level debugger for Windows

empty.exe - Part 1, Example 1

- Source File: part1_intro/empty.c
- Objectives:
  - **View source**: Valid C program? Can it compile? Can it link?
  - **PE-bear**: entry point? sections?
  - **IDA**: entry function?
  - **x32dbg**: loaded/mapped modules?

hello.exe - Part 1, Example 2

- Source File: part1_intro/hello.c
- Objectives:
  - **PE-bear**: find entry point
  - **IDA**:
    - So many functions from a single-line program?
    - Find entry point and compare with written code

hello_msgbox.exe - Part 1, Example 3

- Source File: part1_intro/hello_msgbox.c
- Objectives:
  - **View source**: *WinMain()* instead of *main()*?
  - **PE-bear**: compare subsystem value with hello.exe
  - Review subsystems: Console, Windows, (older, POSIX)

hello_winapi_nocrt.exe - Part 1, Example 4

- Source File: part1_intro/hello_winapi_nocrt.c
- Objectives:
  - View source:
    - WinAPI functions instead of standard C functions?
    - *EntryPoint()* instead of *main()* function?
  - **PE-bear**: find entry point
  - **IDA**: a lot fewer functions without C runtime library

greeting.dll - Part 2, Example 1

- Source File: part2_intro/greeting.{c,h,def}
- Objectives:
  - View source: *DllMain()* instead of *main()* function?
  - **PE-bear**: exported functions

nullpad.exe - Part 2, Example 2

- Source File: part2_intro/nullpad.{c,h,rc}
- Objectives:
  - View source: dynamically resolves functions from greeting.dll using *LoadLibrary* and *GetProcAddress*
  - **PE-bear**: imported functions, resources
  - **ResourceHacker**: view resources
  - Vulnerable to DLL hijacking?

annoying.dll - Part 2, Example 3

- Source File: part2_intro/hijack.{c,h}
- Objectives:
  - View source: greeting.c compiled with *-DANNOYING*
  - Review *Dynamic-Link Library Search Order*
  - How could this DLL be used in a hijacking attempt against the Nullpad application

hello_getproc.exe - Part 3, Example 1

- Source File: part3_intro/hello_getproc.c
- Objectives:
  - **PE-bear**: imports
    - *GetModuleHandle*, *LoadLibrary*, and *GetProcAddress*?
  - **IDA**: identify functions, cross-reference strings

hello_modenum.exe - Part 3, Example 2

- Source File: part3_intro/hello_modenum.c
- Objectives:
  - **PE-bear**: imports
  - **IDA**: identify functions, cross-reference strings helps
  - **x32dbg**: set breakpoint on *GetProcAddress*
    - Debugging can greatly speed up reversing

hello_stealth.exe - Part 3, Example 3a

- Source File: part3_intro/{hello_stealth.c, nt_internal.h}
- Objectives:
  - **PE-bear**: entry point, imports, section names
  - **PEiD KANAL**: any signatures?
  - **IDA**: several functions, no imports nor strings for clues
  - **x32dbg**: debugging can save time, e.g. return values
  - Bonus: zero out TLS directory, still executes - how?

hello_stealth_faked.exe - Part 3, Example 3b

- Several PE headers have been altered with a hex editor
- Objectives:
  - **PE-bear**:
    - Versions of tools used to compile (“Rich” header)
    - Build timestamp of PE
    - Linker and Windows versions
  - Compare with *hello_stealth_original.exe*

hello_stealth_upx.exe - Part 3, Example 3c

- File has been packed with a PE packer
- Objectives:
  - **PE-bear**: entry point, imports, section names
  - **PEiD**: detected packer signatures
  - **IDA**: most of the code has been obscured
  - Unpacking a packed PE? It varies, but *upx -d* is easy

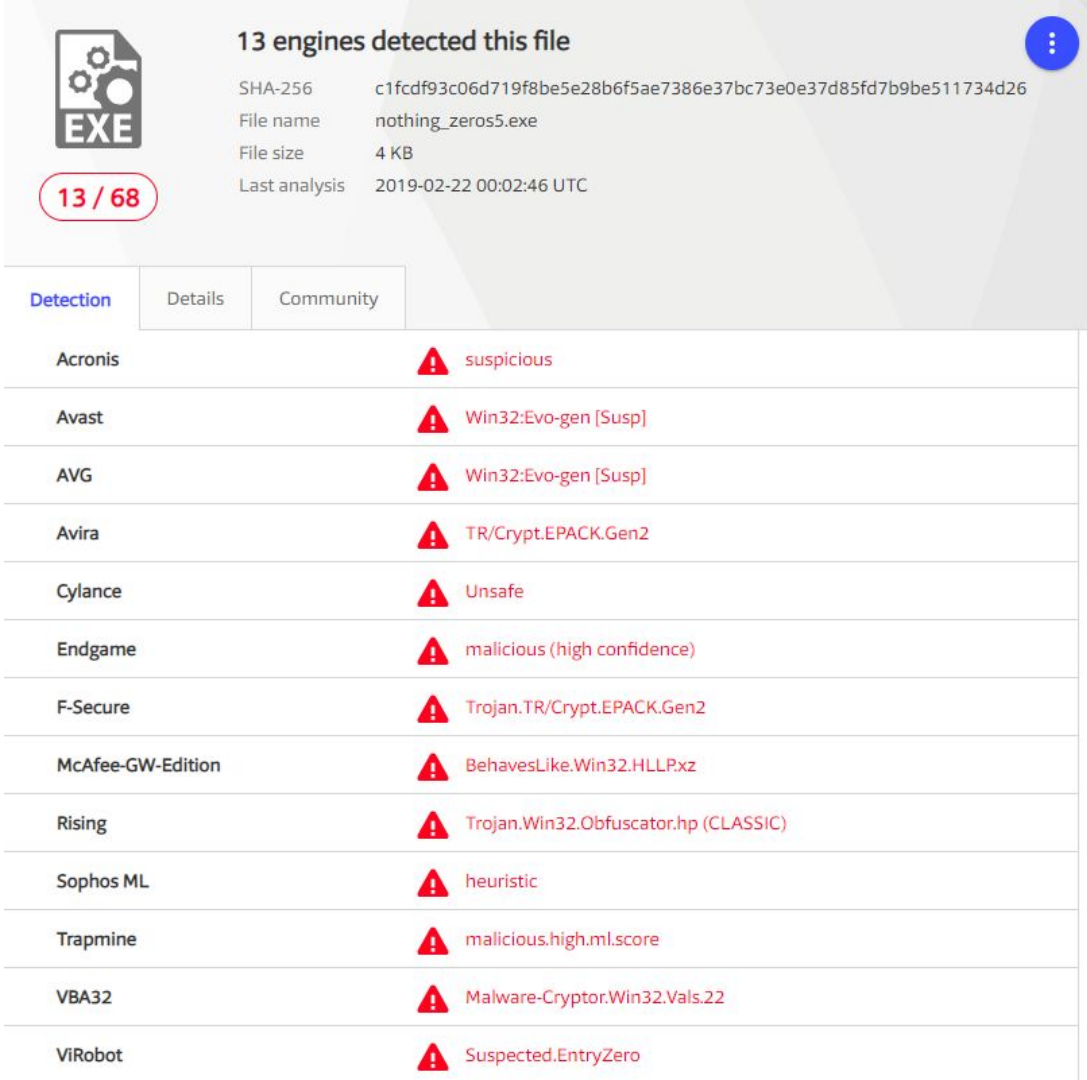
crackme1.exe - Take Home Assignment!

- Objective: reverse engineer the program and determine the algorithm required to generate the secret code
- Console-based crackme, run from command prompt
- Hints:
  - Code validation is base on the entered name
  - Locate validation function by debugging or following references to strings

Malware

VirusTotal

- Submit suspicious files, hashes, and URLs for scanning
- Submitted *hello_stealth.exe*
- 13 antivirus software products suspected this “hello world” application is malicious



Yara Rule for Example 3a

```
import pe
rule malware_probably_ep0 {
    meta:
        desc = "Questionable PE file, EP==0 JMP at offset 3"
    condition:
        pe.entry_point == 0x0 and
        int8(0x3) == 0xE9 and
        pe.sections[0].name == "" and
        pe.sections[1].name == "" and
        pe.sections[2].name == ""
}
```

Demonstrated PE Obfuscation Techniques

- Dynamically resolving functions at runtime
  - Hides imports for unusual or suspicious functions
- Encoding or encrypting strings that would otherwise be easily readable in *.data* or *.rdata* sections
- Modifying PE headers to perform unusual behaviors
- TLS callbacks used as an alternative entry point
- Falsifying headers or planting strings to complicate attribution

Demonstrated PE Obfuscation Techniques Cont.

- Executable packers to hide code “on-disk”, unpacks at runtime
  - Antiviruses (or sandboxes) may unpack to analyze
- Using undocumented or unofficial operating system functions and data structures
- These techniques attempt to mask the PE’s functionality
  - Reduce signatures that security products use for detection
  - Increase complexity for analysts and reverse engineers

Malware Analysis

- Most malware is packed with custom packers, first step is typically unpacking or dumping and reconstructing it
- Identify traits: persistence, purpose, communication (C2 or ?), encryption keys, evasion techniques (debug, sandbox, VM)
- Fully reverse engineering a binary is not common, often only when doing a write-up on a new malware strain
- Binary diff tools useful to compare with previous versions

Finding Live Malware Samples

- Most commercial malware databases with current malware require subscriptions (VirusTotal, VirusBay, MalDatabase, etc.)
- theZoo: https://github.com/ytisf/theZoo
- Follow other malware analysts on twitter, they often post URLs to current samples they have analyzed
- It should go without saying, but...exercise caution
    - 0days are real and don’t always require execution

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 API's instead
- Even better, use Ero Carrera's Python module, ***pefile***: https://github.com/erocarrera/pefile

Wrapping Up

Where Do I Start?

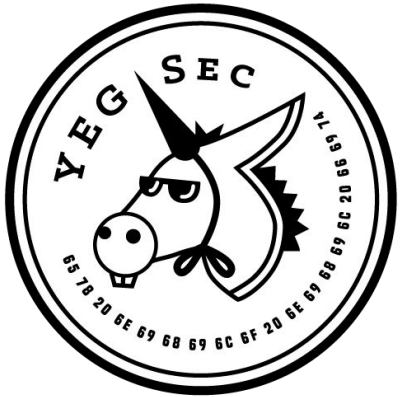
- Learn assembly, C programming, hardware architecture, and operating system internals; will make R.C.E. a lot easier
  - MacEwan's CMPT 280, 380, 480, 229, 360, 361, 464
- Crackme's and Unpackme's are great to practice with
- **The most important thing is to just start**
  - Your knowledge will progress as you read information to understand specific APIs, instructions, and techniques

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

Security Is Neat

- Join yegsec: https://www.yegsec.ca/
  - Monthly meet-ups at Startup Edmonton
  - Security-focussed talks
  - InfoSec professionals to bounce ideas off of
  - Aspiring students are welcome



Fin.

Questions?

Slides, demos, and source code are available at:

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