Advanced Topics in Malware Analysis

Static Malware Analysis Tools and Techniques

Brendan Saltaformaggio, PhD

Assistant Professor

School of Electrical and Computer Engineering

Introduction to Malware Analysis Tools

Georgia Tech

1

Learning Objectives

- Discuss how to analyze executable files
- Evaluate sections, entry point, and dependencies of executable files.
- Describe the static executable inspection tools necessary for analysis and debugging.
- Employ GHIDRA to analyze and debug executable files.

Georgia Tech

2

Static Malware Analysis

Study code of malware sample without executing

Advantages

- No danger of executing payload
- Discovery of otherwise hidden behaviors

Disadvantages

- Encryption / packing
- Complexity
- Compilation obfuscates high level language structures

Georgia Tech

3

Tools for the Trade

- EXE manipulation tools (e.g., dumpbin/PEView/PE Explorer)
  - Analyze structure of EXE
  - Dump code, data, import tables, export tables, resources, …
- Disassemblers (e.g., IDA, Ghidra)
  - Analyze code (generally in assembler)
  - Annotate code and data to increase understanding
- Decompilers (e.g., Hex-Rays decompiler plugin for IDA)
  - Attempt to translate into high level languages/pseudocode

Georgia Tech

4

Analyzing Executable File Formats

The goal of analyzing executable files is to understand:

- Sections
- Entry point
- Dependencies
  - Imported functions
  - Exported functions
- Symbol table
- Code
- Initialized data
- Relocation information
- …

- e.g., dumpbin (Windows), PEView (Windows), readelf (Linux), otool (Mac OS X)
- Tools are NOT a replacement for a deep understanding of the executable formats
  - Read, study, Google, learn everything you can about the format of the file you're analyzing

5

Must I?

- Short answer: Yes
- For many old DOS viruses, firmware, etc. and some Windows malware, it may be possible to do reverse engineering without caring about executable file formats
- In general, nearly all modern malware heavily abuse executable file formats to infect and propagate
- During reverse engineering, code may be opaque (to you) unless you understand executable file formats in detail
- Reverse engineering is often all about the nasty details…sorry!
  - Google, Google, Google! E.g., "What is offset 0x65 in PE Header?"

6

Advanced Topics in Malware Analysis

Static Malware Analysis Tools and Techniques

Brendan Saltaformaggio, PhD

Assistant Professor

School of Electrical and Computer Engineering

Executable Formats

Georgia Tech

7

Executable Formats

- Executable format allows a loader to **instantiate** a new program execution
  - Loader may be a program or part of the operating system
- Compiler may even **insert** some “set up” code before your **main**(…) function
  - Helps handle transition from the loader to the program
  - Initializes data, calls initialization functions in libraries
  - Read about: **__tmainCRTStartup**
- Programs then execute machine code directly and interface with the runtime system (libraries & OS)

Georgia Tech

8

Executable Formats (Cont.)

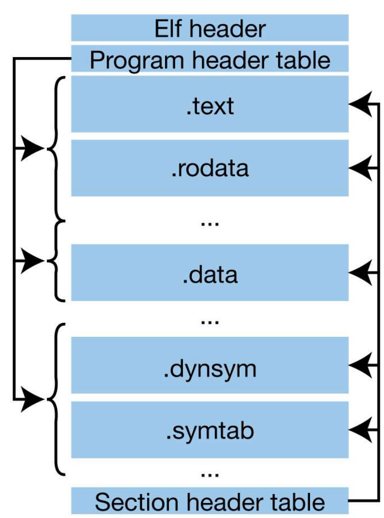
- Executable formats have evolved, many different formats exist
- DOS/Windows executables evolved from COM files that were restricted to 64KB to EXE files executing in 16-bit mode to 32-bit and 64-bit Windows executables
- On Unix, ELF (Executable and Linkable Format) is most common, by far

9

ELF Format (LINUX)

ELF allows two interpretations of each file:

- **Segments** contain permissions and mapped regions
- **Sections** enable linking and relocation
- The **loader**:
  1) Reads the ELF header
  2) Maps segments into a new virtual address space
  3) Resolves relocations
  4) Starts executing from the entry point
- If **.interp** section is present, the interpreter loads the interpreter executable (and resolves relocations)
  - This section holds the literal path name of the interpreter



10

Tools for Inspecting ELF Executables

- **readelf** and **objdump** can display information about ELF files
  - Executables, shared objects, archives, object files, …
- Your brain is better than those
  - Many different ELF specifications for different platforms
  - https://en.wikipedia.org/wiki/Executable_and_Linkable_Format#Specifications
- **readelf -h <filename>** displays basic information about ELF header
- **readelf -l <filename>** displays program headers used by loader to map program into memory
- **readelf -S <filename>** displays sections, used by loader to relocate and connect different parts of the executable

11

Windows PE Format

- Portable Executable (PE) format for **.EXE**, **.SCR**, **.DLL** et al.
  - **No distinction** between these
    - EXEs contain a startup entry point/DLLs export functions
- File format consists of a number of sections, including a section for "**backwards compatibility**" with MS-DOS
  - Backwards compatibility is typically **limited to a small embedded application** that indicates Windows is required

12

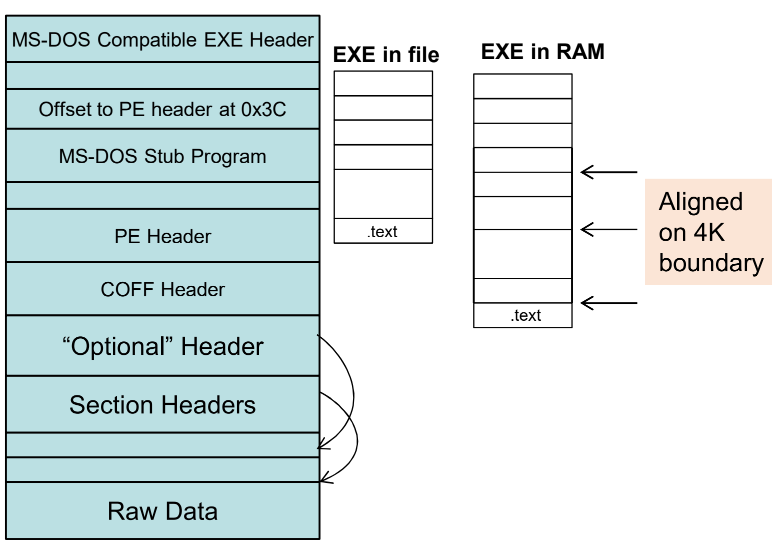
Windows PE Format (Cont.)

- Complete specification available from Microsoft:
  - "Microsoft Portable Executable and Common Object File Format Specification"
  - We will also use "Peering Inside the PE"
    - Online at: https://msdn.microsoft.com/en-us/library/ms809762(d=printer).aspx
    - PDF also on Canvas!
- Very important to be familiar with all the details!

13

14

PE Section Alignment IN FILE VS. MEMORY



- Typical **file alignment**: **512 bytes**
- Typical **section alignment**: **4K** (page size)
- These pointers are RVAs! **Relative Virtual Addresses**
- Offsets from **base address once loaded in memory**
- https://en.wikipedia.org/wiki/COFF#Relative_virtual_address

Georgia Tech

14

COFF File Header

Offset	Size	Field	Description
0	2	Machine	The number that identifies the type of target machine. For more information, see section 3.3.1, "Machine Types."
2	2	NumberOfSections	The number of sections. This indicates the size of the section table, which immediately follows the headers.
4	4	TimeDateStamp	The low 32 bits of the number of seconds since 00:00 January 1, 1970 (a C run-time time_t value), that indicates when the file was created.
8	4	PointerToSymbolTable	The file offset of the COFF symbol table, or zero if no COFF symbol table is present. This value should be zero for an image because COFF debugging information is deprecated.
12	4	NumberOfSymbols	The number of entries in the symbol table. This data can be used to locate the string table, which immediately follows the symbol table. This value should be zero for an image because COFF debugging information is deprecated.
16	2	SizeOfOptionalHeader	The size of the optional header, which is required for executable files but not for object files. This value should be zero for an object file. For a description of the header format, see section 3.4, "Optional Header (Image Only)."
18	2	Characteristics	The flags that indicate the attributes of the file. For specific flag values, see section 3.3.2, "Characteristics."

Georgia Tech

15

"Optional" Header (1)

- Not optional at all for executables…optional only for object files
- Contains information that helps the loader
- PE32 is 32-bit, PE32+ is 64-bit

Offset (PE32/PE32+)	Size (PE32/PE32+)	Header part	Description
0	28/24	Standard fields	Fields that are defined for all implementations of COFF, including UNIX.
28/24	68/88	Windows-specific fields	Additional fields to support specific features of Windows (for example, subsystems).
96/112	Variable	Data directories	Address/size pairs for special tables that are found in the image file and are used by the operating system (for example, the import table and the export table).

Georgia Tech

16

“Optional” Header (2)

Offset	Size	Field	Description
0	2	Magic	The unsigned integer that identifies the state of the image file. The most common number is 0x10B, which identifies it as a normal executable file. 0x107 identifies it as a ROM image, and 0x20B identifies it as a PE32+ executable.
2	1	MajorLinkerVersion	The linker major version number.
3	1	MinorLinkerVersion	The linker minor version number.
4	4	SizeOfCode	The size of the code (text) section, or the sum of all code sections if there are multiple sections.
8	4	SizeOfInitializedData	The size of the initialized data section, or the sum of all such sections if there are multiple data sections.
12	4	SizeOfUninitializedData	The size of the uninitialized data section (BSS), or the sum of all such sections if there are multiple BSS sections.
16	4	AddressOfEntryPoint	The address of the entry point relative to the image base when the executable file is loaded into memory. For program images, this is the starting address. For device drivers, this is the address of the initialization function. An entry point is optional for DLLs. When no entry point is present, this field must be zero.
20	4	BaseOfCode	The address that is relative to the image base of the beginning-of-code section when it is loaded into memory.

Georgia Tech

17

“Optional” Header (3)

- PE32 contains this additional field, which is absent in PE32+, following BaseOfCode.

Offset	Size	Field	Description
24	4	BaseOfData	The address that is relative to the image base of the beginning-of-data section when it is loaded into memory.

Georgia Tech

18

"Optional" Header (4)

Windows-specific Fields

Offset (PE32/PE32+)	Size (PE32/PE32+)	Field	Description
28/24	4/8	ImageBase	The preferred address of the first byte of image when loaded into memory; must be a multiple of 64 K. The default for DLLs is 0x10000000. The default for Windows CE EXEs is 0x00010000. The default for Windows NT, Windows 2000, Windows XP, Windows 95, Windows 98, and Windows Me is 0x00400000.
32/32	4	SectionAlignment	The alignment (in bytes) of sections when they are loaded into memory. It must be greater than or equal to FileAlignment. The default is the page size for the architecture.
36/36	4	FileAlignment	The alignment factor (in bytes) that is used to align the raw data of sections in the image file. The value should be a power of 2 between 512 and 64 K, inclusive. The default is 512. If the SectionAlignment is less than the architecture's page size, then FileAlignment must match SectionAlignment.

Impacts Relative Virtual Addresses (RVAs)

Georgia Tech

19

"Optional" Header (5)

More Windows-specific Fields

Offset (PE32/PE32+)	Size (PE32/PE32+)	Field	Description
40/40	2	MajorOperatingSystemVersion	The major version number of the required operating system.
42/42	2	MinorOperatingSystemVersion	The minor version number of the required operating system.
44/44	2	MajorImageVersion	The major version number of the image.
46/46	2	MinorImageVersion	The minor version number of the image.
48/48	2	MajorSubsystemVersion	The major version number of the subsystem.
50/50	2	MinorSubsystemVersion	The minor version number of the subsystem.
52/52	4	Win32VersionValue	Reserved, must be zero.
56/56	4	SizeOfImage	The size (in bytes) of the image, including all headers, as the image is loaded in memory. It must be a multiple of SectionAlignment.

Georgia Tech

20

“Optional” Header (6)

More More Windows-specific Fields

Offset (PE32/PE32+)	Size (PE32/PE32+)	Field	Description
60/60	4	SizeOfHeaders	The combined size of an MS-DOS stub, PE header, and section headers rounded up to a multiple of FileAlignment.
64/64	4	CheckSum	The image file checksum. The algorithm for computing the checksum is incorporated into IMAGHELP.DLL. The following are checked for validation at load time: all drivers, any DLL loaded at boot time, and any DLL that is loaded into a critical Windows process.
68/68	2	Subsystem	The subsystem that is required to run this image. For more information, see “Windows Subsystem” later in this specification.
70/70	2	DllCharacteristics	For more information, see “DLL Characteristics” later in this specification.
72/72	4/8	SizeOfStackReserve	The size of the stack to reserve. Only SizeOfStackCommit is committed; the rest is made available one page at a time until the reserve size is reached.
76/80	4/8	SizeOfStackCommit	The size of the stack to commit.

21

“Optional” Header (7)

More More More Windows-specific Fields

Offset (PE32/PE32+)	Size (PE32/PE32+)	Field	Description
80/88	4/8	SizeOfHeapReserve	The size of the local heap space to reserve. Only SizeOfHeapCommit is committed; the rest is made available one page at a time until the reserve size is reached.
84/96	4/8	SizeOfHeapCommit	The size of the local heap space to commit.
88/104	4	LoaderFlags	Reserved, must be zero.
92/108	4	NumberOfRvaAndSizes	The number of data-directory entries in the remainder of the optional header. Each describes a location and size.

22

“Optional” Header: Data Directories

```
typedef struct _IMAGE_DATA_DIRECTORY {
    // relative virtual address (RVA) of table
    DWORD   VirtualAddress;
    DWORD   Size; // size of table in bytes
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

23

On screen- Brief summary- transition

24

Advanced Topics in Malware Analysis

Static Malware Analysis Tools and Techniques

Brendan Saltaformaggio, PhD
Assistant Professor
School of Electrical and Computer Engineering

Data Directories and Section Headers

Georgia Tech

25

Data Directories (1)

These are important in RE:

Offset (PE/PE32+)	Size	Field	Description
96/112	8	Export Table	The export table address and size. For more information see section 6.3, "The .edata Section (Image Only)."
104/120	8	Import Table	The import table address and size. For more information, see section 6.4, "The .idata Section."
112/128	8	Resource Table	The resource table address and size. For more information, see section 6.9, "The .rsrc Section."
120/136	8	Exception Table	The exception table address and size. For more information, see section 6.5, "The .pdata Section."
128/144	8	Certificate Table	The attribute certificate table address and size. For more information, see section 5.7, "The attribute certificate table (Image Only)."
136/152	8	Base Relocation Table	The base relocation table address and size. For more information, see section 6.6, "The **.reloc** Section (Image Only)."
144/160	8	Debug	The debug data starting address and size. For more information, see section 6.1, "The .debug Section."
152/168	8	Architecture	Reserved, must be 0

Georgia Tech

26

Data Directories (2)

These are important in RE:

Offset (PE/PE32+)	Size	Field	Description
160/176	8	Global Ptr	The RVA of the value to be stored in the global pointer register. The size member of this structure must be set to zero.
168/184	8	TLS Table	The thread local storage (TLS) table address and size. For more information, see section 6.7, “The .tls Section.”
176/192	8	Load Config Table	The load configuration table address and size. For more information, see section 6.8, “The Load Configuration Structure (Image Only).”
184/200	8	Bound Import	The bound import table address and size.
192/208	8	IAT	The import address table address and size. For more information, see section 6.4.4, “Import Address Table.”
200/216	8	Delay Import Descriptor	The delay import descriptor address and size. For more information, see section 5.8, “Delay-Load Import Tables (Image Only).”
208/224	8	CLR Runtime Header	The CLR runtime header address and size. For more information, see section 6.10, “The .cormeta Section (Object Only).”
216/232	8	Reserved, must be zero	

27

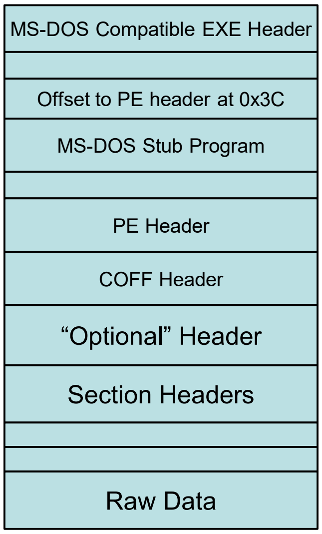
Section Headers (1)

These are important in RE:

Offset (PE/PE32+)	Size	Field	Description
160/176	8	Global Ptr	The RVA of the value to be stored in the global pointer register. The size member of this structure must be set to zero.
168/184	8	TLS Table	The thread local storage (TLS) table address and size. For more information, see section 6.7, “The .tls Section.”
176/192	8	Load Config Table	The load configuration table address and size. For more information, see section 6.8, “The Load Configuration Structure (Image Only).”
184/200	8	Bound Import	The bound import table address and size.
192/208	8	IAT	The import address table address and size. For more information, see section 6.4.4, “Import Address Table.”
200/216	8	Delay Import Descriptor	The delay import descriptor address and size. For more information, see section 5.8, “Delay-Load Import Tables (Image Only).”
208/224	8	CLR Runtime Header	The CLR runtime header address and size. For more information, see section 6.10, “The .cormeta Section (Object Only).”
216/232	8	Reserved, must be zero	

28

Section Headers (2)



- Section headers area follows the “optional” header in the PE file
- Contains a series of 40 byte entries
- Each entry describes one of the sections in the file
- Section headers are followed by section data
  - .text, .bss, etc.

29

Section Headers (3)

Offset	Size	Field	Description
0	8	Name	An 8-byte, null-padded UTF-8 encoded string. If the string is exactly 8 characters long, there is no terminating null. For longer names, this field contains a slash (/) that is followed by an ASCII representation of a decimal number that is an offset into the string table. Executable images do not use a string table and do not support section names longer than 8 characters. Long names in object files are truncated if they are emitted to an executable file.
8	4	VirtualSize	The total size of the section when loaded into memory. If this value is greater than SizeOfRawData, the section is zero-padded. This field is valid only for executable images and should be set to zero for object files.
12	4	VirtualAddress	For executable images, the address of the first byte of the section relative to the image base when the section is loaded into memory. For object files, this field is the address of the first byte before relocation is applied; for simplicity, compilers should set this to zero. Otherwise, it is an arbitrary value that is subtracted from offsets during relocation.

30

Section Headers (4)

Offset	Size	Field	Description
16	4	SizeOfRawData	The size of the section (for object files) or the size of the initialized data on disk (for image files). For executable images, this must be a multiple of FileAlignment from the optional header. If this is less than VirtualSize, the remainder of the section is zero-filled. Because the SizeOfRawData field is rounded but the VirtualSize field is not, it is possible for SizeOfRawData to be greater than VirtualSize as well. When a section contains only uninitialized data, this field should be zero.
20	4	PointerToRawData	The file pointer to the first page of the section within the COFF file. For executable images, this must be a multiple of FileAlignment from the optional header. For object files, the value should be aligned on a 4-byte boundary for best performance. When a section contains only uninitialized data, this field should be zero.

31

Section Headers (5)

Offset	Size	Field	Description
24	4	PointerToRelocations	The file pointer to the beginning of relocation entries for the section. This is set to zero for executable images or if there are no relocations.
28	4	PointerToLineNumbers	The file pointer to the beginning of line-number entries for the section. This is set to zero if there are no COFF line numbers. This value should be zero for an image because COFF debugging information is deprecated.
32	2	NumberOfRelocations	The number of relocation entries for the section. This is set to zero for executable images.
34	2	NumberOfLinenumbers	The number of line-number entries for the section. This value should be zero for an image because COFF debugging information is deprecated.
36	4	Characteristics	The flags that describe the characteristics of the section. For more information, see section 4.1, "Section Flags."

32

Important PE Sections

- .bss section
  - Uninitialized data
- .data section
  - Initialized data
- .edata section
  - Export table for file
- .idata section
  - Import table for file
- .text section
  - Executable code
- Others
- Names are by convention—they can vary!

Many tools exist for static analysis of PE file sections!

dumpbin

PEView

PE Explorer

Georgia Tech

33

Dumpbin Notepad.exe

```
Administrator: C:\Windows\system32\cmd.exe
Microsoft (R) COFF/PE Dumper Version 9.00.30729.01
Copyright (C) Microsoft Corporation.  All rights reserved.


Dump of file notepad.exe

PE signature found

File Type: EXECUTABLE IMAGE

FILE HEADER VALUES
             14C machine (x86)
               4 number of sections
        4A5BC60F time date stamp Mon Jul 13 18:41:03 2009
               0 file pointer to symbol table
               0 number of symbols
              E0 size of optional header
             102 characteristics
                   Executable
                   32 bit word machine

OPTIONAL HEADER VALUES
             10B magic # (PE32)
            9.00 linker version
            A800 size of code
           22400 size of initialized data
               0 size of uninitialized data
            3689 entry point (01003689)
            1000 base of code
            C000 base of data
         1000000 image base (01000000 to 0102FFFF)
            1000 section alignment
             200 file alignment
            6.01 operating system version
            6.01 image version
            6.01 subsystem version
               0 Win32 version
           30000 size of image
             400 size of headers
           39741 checksum
:_
```

Remember Relative Virtual Addresses (RVAs)!

34

Dumpbin Notepad.exe (2)

```
Administrator: C:\Windows\system32\cmd.exe
                   NX compatible
                   Terminal Server Aware
           40000 size of stack reserve
           11000 size of stack commit
          100000 size of heap reserve
            1000 size of heap commit
               0 loader flags
              10 number of directories
               0 [       0] RVA [size] of Export Directory
            A048 [     12C] RVA [size] of Import Directory
            F000 [   1F160] RVA [size] of Resource Directory
               0 [       0] RVA [size] of Exception Directory
               0 [       0] RVA [size] of Certificates Directory
           2F000 [     E34] RVA [size] of Base Relocation Directory
            B62C [      38] RVA [size] of Debug Directory
               0 [       0] RVA [size] of Architecture Directory
               0 [       0] RVA [size] of Global Pointer Directory
               0 [       0] RVA [size] of Thread Storage Directory
            6D58 [      40] RVA [size] of Load Configuration Directory
             278 [     128] RVA [size] of Bound Import Directory
            1000 [     400] RVA [size] of Import Address Table Directory
               0 [       0] RVA [size] of Delay Import Directory
               0 [       0] RVA [size] of COM Descriptor Directory
               0 [       0] RVA [size] of Reserved Directory


SECTION HEADER #1
   .text name
    A68C virtual size
    1000 virtual address (01001000 to 0100B68B)
    A800 size of raw data
     400 file pointer to raw data (00000400 to 0000ABFF)
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
60000020 flags
         Code
         Execute Read

:_
```

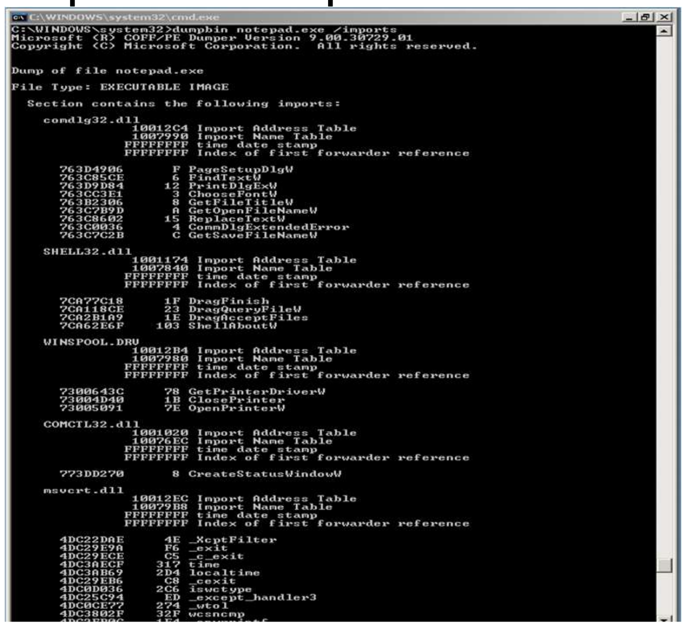
35

Dumpbin Notepad.exe (3)

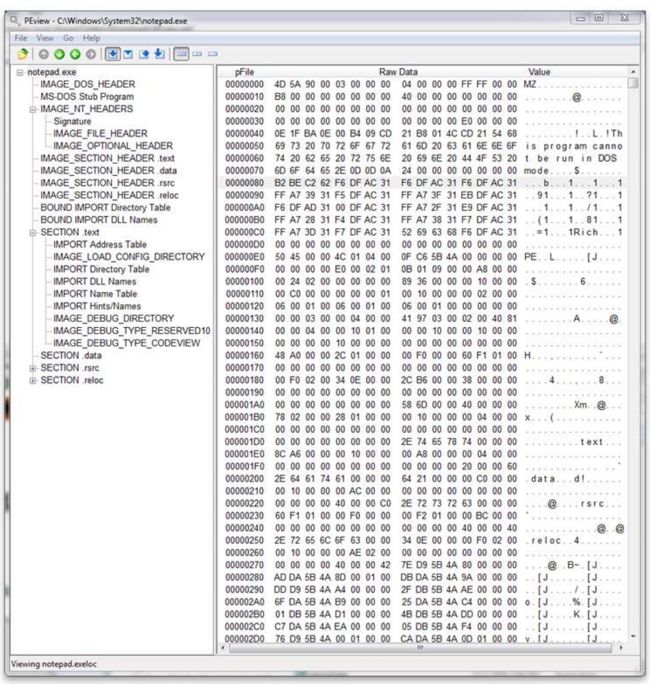
```
Administrator: C:\Windows\system32\cmd.exe
SECTION HEADER #2
   .data name
    2164 virtual size
    C000 virtual address (0100C000 to 0100E163)
    1000 size of raw data
    AC00 file pointer to raw data (0000AC00 to 0000BBFF)
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
C0000040 flags
         Initialized Data
         Read Write

SECTION HEADER #3
   .rsrc name
   1F160 virtual size
    F000 virtual address (0100F000 to 0102E15F)
   1F200 size of raw data
    BC00 file pointer to raw data (0000BC00 to 0002ADFF)
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
40000040 flags
         Initialized Data
         Read Only

SECTION HEADER #4
  .reloc name
     E34 virtual size
   2F000 virtual address (0102F000 to 0102FE33)
    1000 size of raw data
   2AE00 file pointer to raw data (0002AE00 to 0002BDFF)
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
42000040 flags
         Initialized Data
         Discardable
         Read Only

:_
```

36

Dumpbin Notepad.exe imports



37



PEView Notepad.exe

38

Strings, Even

- Can sometimes gain some clues by running strings command against executable
  - **strings malware.exe** → print all ASCII strings
  - **strings -el malware.exe** → print all Unicode strings
- Will not always be helpful
  - Packed/encrypted malware has no readable strings
- But memory forensics and reverse engineering started this way…
- Occasionally still useful

Georgia Tech

39

Strings: Example: ASCII

- Real World Malware Investigation: W32.Gruel.a Email Worm
- **`$ strings Email-Worm.Win32.Gruel.a`**
- …

```
kIlLeRgUaTe 1.03, I mAke ThIs vIrUs BeCaUsE I dOn'T hAvE
NoThInG tO dO!!

We have created an error report thet you cand send to us. we
will treat this report as confidential and anounymous.

Please tell microsoft about this problem.

Windows X found serious error.
```

…

Georgia Tech

40

Strings: Example: UNICODE

- Real World Malware Investigation: W32.Gruel.a Email Worm
- **$ strings -e l Email-Worm.Win32.Gruel.a**

…

```
Norton Security Response: has detected a new virus in the
Internet. For this reason we made this tool attachement, to
protect your computer from this serious virus. Due to the
number of submissions received from customers, Symantec
Security Response has upgraded this threat to a Category 5
(Maximum )
```

…

Georgia Tech

41

But these won't help you with packed binaries… ☹

SO LET'S TALK ABOUT PACKERS

Georgia Tech

42

Advanced Topics in Malware Analysis

Static Malware Analysis Tools and Techniques

Brendan Saltaformaggio, PhD
Assistant Professor
School of Electrical and Computer Engineering

Packers

Georgia Tech

43

Reversing Modern Malware

- State-of-the-art malware doesn't present an easy target for reverse engineering attempts
- The stuff you've learned so far is essential for understanding what's going on once you can get a reasonable disassembly
- Modern malware doesn't want you to **even access** the disassembly
- Furthermore, modern malware doesn't want you to be able to closely analyze its runtime behavior either
- So packers evolved to add layers upon layers of misery to reverse engineering!

Georgia Tech

44

Packers: Basics

- Packers add obfuscation/encryption to parts of the virus body to thwart static analysis
- Packing will hide one binary inside a benign outer “host” binary
  - Often the hidden binary is encrypted in the file
  - Decrypted only at execution time (possibly 1 small chunk at a time)

45

Packers: Basics

General Structure:

- Compress/encrypt a **target EXE file's code**
- Generate **new** EXE with **encrypted code** as a payload
- New EXE has modified entry point, which points to an **unpacking module**
- Unpacking module responsible for **decryption**
- Unpacking module will likely contain **anti-debugging tricks**
- Unpacker’s goal is to **jump to entry point** of original code

- Packers have both **commercial** and “**malicious**” uses
- Packed doesn’t necessarily imply malware, but very common in modern malware

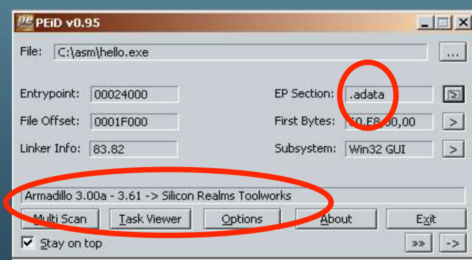
Georgia Tech

46

Static Analysis Question #1: Is It Packed?

- Is it packed? If so, important to **determine what sort of packer** was used
- Is it something hand-rolled? Something known?

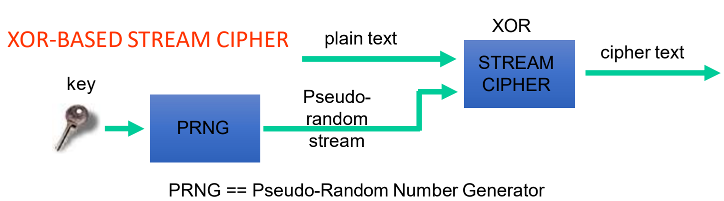
- Known packers have **signatures** too!
  - Many tools have been designed to detect packing
  - **PEiD** is still a fan favorite! https://www.aldeid.com/wiki/PEiD
  - Detects ~600 different packers



47

Worst Case: Entirely Hand-Made Packer

- PEID or similar may not help — could be an entirely new/home made packer
- If it's hand-rolled, commonly XOR-based stream cipher over part of the virus body (and possibly the original EXE contents), with only the decryption code in the clear



- PRNG code will probably be small — the virus is trying to be stealthy
- Luckily! The key hopefully has to be stored somewhere in the PE file
- Unless…

48

REAL Worst Case: Keyless Hand-Made Packer!

- It can always get worse…
- The key could be downloaded from the malware's command & control servers!
- Or, there could be no key at all! The decrypt and hash loop 😭
- Read More: "**The Art of Unpacking**" Mark Vincent Yason. Blackhat USA 2007.
  - PDF on Canvas
- Read Even More: "**Binary-code obfuscations in prevalent packer tools**" Roundy, Kevin A., and Barton P. Miller. *ACM Computing Surveys (CSUR)* 46.1 (2013)
- If PEID (or the like) fail, resort to **manual inspection** ☹
- Next, we will discuss ways to find an infection in general, not just packing

Georgia Tech

49

Manual Inspection of Section Names

- Where are they hiding?
- Typical section names:
  - .code / .text [executable code]
  - .bss [uninitialized data]
  - .data [initialized data]
  - .reloc [relocation info]
  - .idata [imports]
  - .edata [exports]
  - .pdata [execution handling]
  - .tls
  - …
- There are others…have a close look at our old friend, the PE/COFF specification
- Do section names look unusual?
  - If yes, **suspicious**

Georgia Tech

50

Section Names: Not Unusual

```
C:\WINDOWS\system32\cmd.exe
C:\asm>dumpbin /summary hello.exe.PreARM
Microsoft (R) COFF/PE Dumper Version 9.00.30729.01
Copyright (C) Microsoft Corporation.  All rights reserved.


Dump of file hello.exe.PreARM

File Type: EXECUTABLE IMAGE

  Summary

        1000 .data
        1000 .rdata
        1000 .text

C:\asm>
C:\asm>
C:\asm>
C:\asm>
C:\asm>
```

Georgia Tech

51

Section Names: Yes Unusual

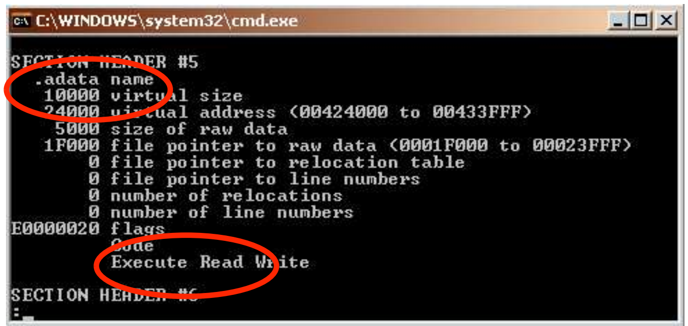
```
C:\WINDOWS\system32\cmd.exe
C:\asm>dumpbin /summary hello.exe
Microsoft (R) COFF/PE Dumper Version 9.00.30729.01
Copyright (C) Microsoft Corporation.  All rights reserved.


Dump of file hello.exe

File Type: EXECUTABLE IMAGE

  Summary

       10000 .adata
        1000 .data
       20000 .data1
       30000 .pdata
        1000 .rdata
        1000 .text
       20000 .text1

C:\asm>
```

Georgia Tech

52

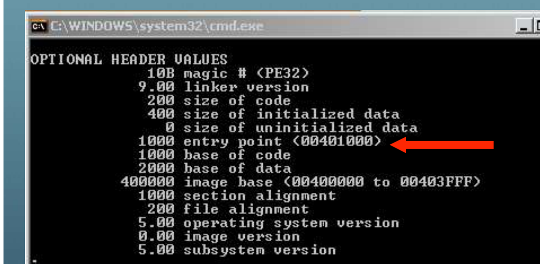
Inspection of Section Permissions

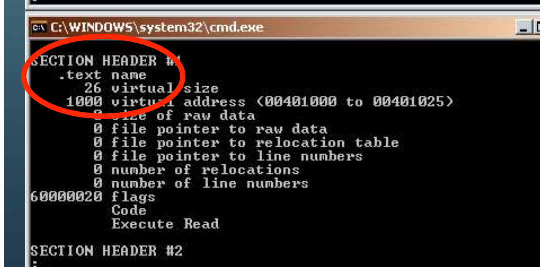
- Only .code / .text should be executable
- See other executable sections?

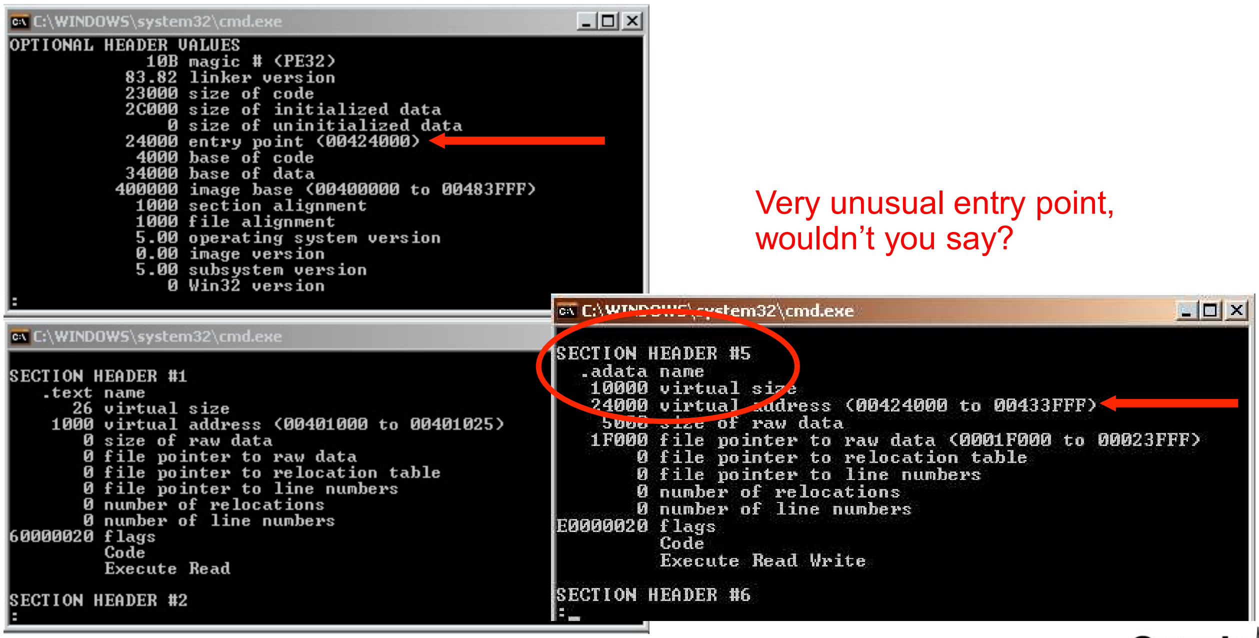


Manual Inspection of Entry Point

- Is the **entry** point in the **.text** or **.code** section?
  - If not, suspicious
- Entry point should typically be **in first code section**
- Code should only be in **.text** / **.code**







Very unusual entry point, wouldn't you say?

55

On Screen- Brief summary and transition

56

Advanced Topics in Malware Analysis

Static Malware Analysis Tools and Techniques

Brendan Saltaformaggio, PhD

Assistant Professor

School of Electrical and Computer Engineering

Finding Strings, Tables, and Code

Georgia Tech

57

Manual Inspection to Reveal Printable Strings

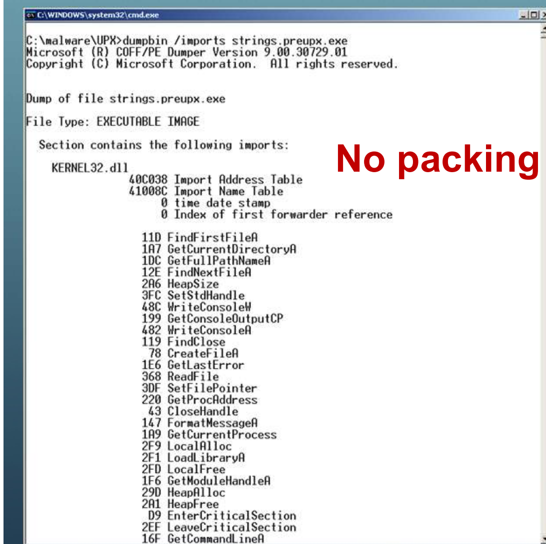
- **Strings** again? Really??
  - Yes.
- Unusual for executables not to have human-readable strings
- Some **packed executables** do have **printable strings**, particularly commercial wrappers that need to provide some help for users
- Run **strings** against **.EXE** file
- What do you see?

Georgia Tech

58

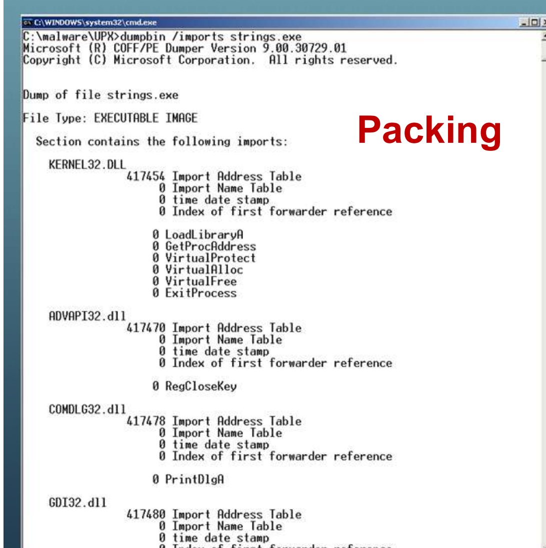
Manual Inspection of the Import Table

- Does the IAT seem suspiciously empty?
- Most Windows programs have a lot of imports



Manual Inspection of IMPORT TABLE

- Packers typically don't need many imports
  - LoadLibrary()
  - GetProcAddress()
  - etc.
- Complete import table for packed code will be reassembled before it is executed
- But won't be found by static analysis



Finally, Manual Inspection of Code

- Attempt disassembly, e.g. in IDA Pro
- Do large parts of the executable, particularly the targets of JMP/CALLs appear to be obfuscated?
- Suspect packing/encryption
- Ultimate goal is to get **unpacked** & **de-obfuscated executable** for analysis
- Need to **isolate** packer code/encrypted payload and then find original entry point of the payload
- Essentially, locating the point at which decryption is complete
- But How??
- Real Malware: Lucius (e.g.) ➔ Hand-rolled encryption of virus body

Georgia Tech

61

Finding Entry Point of Payload

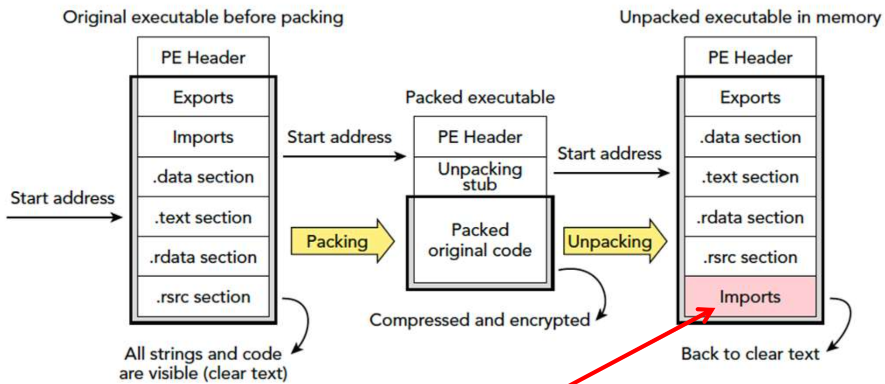
- Static analysis — just look for it … I'm exhausted already!
- Single step in a debugger or emulator
  - e.g., in ollydbg (details coming soon!)
  - **Advantage:** Allows you to gain deeper understanding of the effects of the unpacking process (or ignore/modify them)
  - **Disadvantages**:
    - Slow
    - VERY DANGEROUS --- Where does unpacking end and "kill everyone" payload begin??
- Scan function call graph in IDA Pro
  - **Advantage:** May save large amounts of time
  - **Disadvantages:**
    - Tightly rolled packers won't show you much
    - But for tightly rolled packers, it may be easier to discover where they terminate

Georgia Tech

62

Packed Executables in-Memory Analysis

Figure 8-5 in The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory



- Grab **memory image** at every **JMP**?
- In many cases, unpacking will be complete (but not always)
- Then proceed with **static** analysis.

63

Brief Packer Case Study: Themida

- Commercial from Oreans Technologies
- Typically used to protect commercial software
- Encryption
- Anti-debugger tricks
- Anti-acquisition tricks to prevent process memory dumping
- Garbage instruction insertion
- FPU bugs to injure debuggers

64

Brief Packer Case Study: Themida (Cont.)

- Kernel mode (Ring 0) components
- VM-based emulation of x86 code
- See http://www.oreans.com/themida_features.php
- Definitely used to protect some malware
- See: http://www.wilderssecurity.com/showthread.php?t=184840
- Defeats anti-virus
- Result: Themida-protected executables may simply be disallowed in secure environments!

65

Themida

- And has a nice UI!

66

Packer Case Study: UPX

- Very commonly used open source packer
  - http://upx.sourceforge.net
- Goal is to **decrease** executable **size** and **load time**
- Not to defeat debuggers, etc.
- Good as basic practice for unpacking
- Of course, **upx –d** will unpack for you ☺
- Go try it for yourself!

Georgia Tech

67

Advanced Topics in Malware Analysis

Static Malware Analysis Tools and Techniques

Brendan Saltaformaggio, PhD

Assistant Professor

School of Electrical and Computer Engineering

Static Analysis Tools and Techniques

Georgia Tech

68

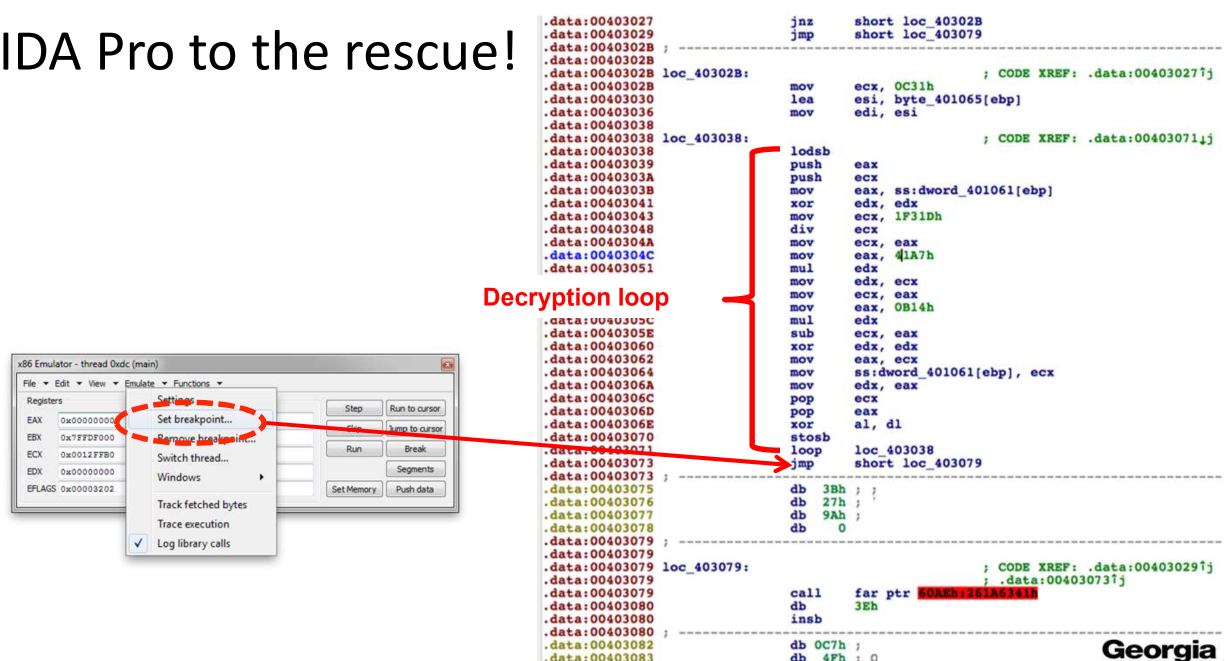
The Swiss Army Knife: Disassemblers

- **Goal**: Automate process of generating ASM source from executable
- **Difficulties:**
  - High Level Language (HLL) → binary is a VERY lossy process
  - Code/data mixture
    - What's code? What's data?
    - Different approaches to tackling this problem
  - Branch targets computed at runtime
  - Dynamically loaded code (e.g., DLLs)
  - Branches that do not target the beginning of instructions
    - E.g., Jump inside multi-byte instructions or inside data areas
  - Self-modifying code
  - Deliberate obfuscation/packing/encryption (e.g., Armadillo from before)

Georgia Tech

69

IDA Pro to the rescue!



Georgia Tech

70

IDA Also Analyzes The File Format

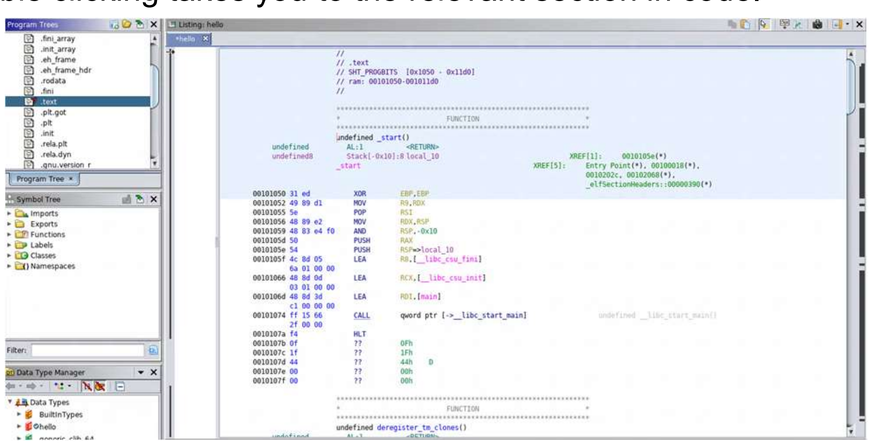
- For example, IDA will show the section each offset falls within:

Georgia Tech

71

File Format in Ghidra Also

- Ghidra analyzes the file format as well
- It stores section header names in the Program Trees window
- Double clicking takes you to the relevant section in code.



Georgia Tech

72

HLL → Binary is VERY Lossy

- One to many
  - Compilation process is not unique
  - HLL code may result in many different binaries
    - Depending on compilation environment/optimization/target architecture
- Data types are lost
- Names and useful symbols are lost
- Debugging information is probably stripped
- Intention of programmer is even further obfuscated

73

Disassembly: Flavor 1 of 2

Recursive-descent Disassembly:
  - Attempt to reconstruct and follow the program control flow
  - Disassemble sequences of bytes only if they can be reached from another valid instruction
- **Bad**: Can't easily handle:
  - Indirect jumps/dynamically computed branches
  - Self-modifying code
- **Good**: Better at handling interleaved code/data
- IDA Pro, GHIDRA, OllyDbg (debugger—covered later)

74

Disassembly: Flavor 2 of 2

Linear-sweep Disassembly:
- From first instruction, disassemble entire stream of bytes
- Next instruction is assumed to follow previous valid instruction

Bad:
- For dense instruction sets (e.g., Intel) not easy to tell if you're off track
- Easily tripped up by interleaved code and data

Good:
- Coverage: If data and code aren't interleaved, not confused by indirect/indexed jumps

- WinDbg, SoftICE (discontinued), gdb, objdump

Georgia Tech

75

Disassembly: Speculative or Hybrid

- Can also use hybrid approaches
  - Example: Do both recursive descent and linear sweep and note similarities/differences
- **Speculative**: Mark portions of binary that have been disassembled and speculatively disassemble others
  - "See what happens"
  - Speculative portions are marked for possible human intervention
- One description of hybrid disassembly:
  - B. Schwarz, S. Debray, G. Andrews, "Disassembly of Executable Code Revisited", IEEE Working Conference on Reverse Engineering (WCRE 2002).

Georgia Tech

76

Advanced Topics in Malware Analysis

Static Malware Analysis Tools and Techniques

Brendan Saltaformaggio, PhD
Assistant Professor
School of Electrical and Computer Engineering

Decompilers

Georgia Tech

77

King of Static Analysis Tools: Decompilers

- Goal: Reverse compilation process
  - Binary → HLL
- **Very** difficult to do
- Producing original source is **impossible**, since compilation is lossy
- Some limited open source solutions
  - e.g., http://boomerang.sourceforge.net/, https://github.com/avast-tl/retdec
- Commercial systems are **expensive**
  - Hex-Rays Decompiler (plug in for IDA Pro) is probably the most famous
- Does a fair job in many cases…
- Very very expensive!!! Only researchers at Georgia Tech have access

Georgia Tech

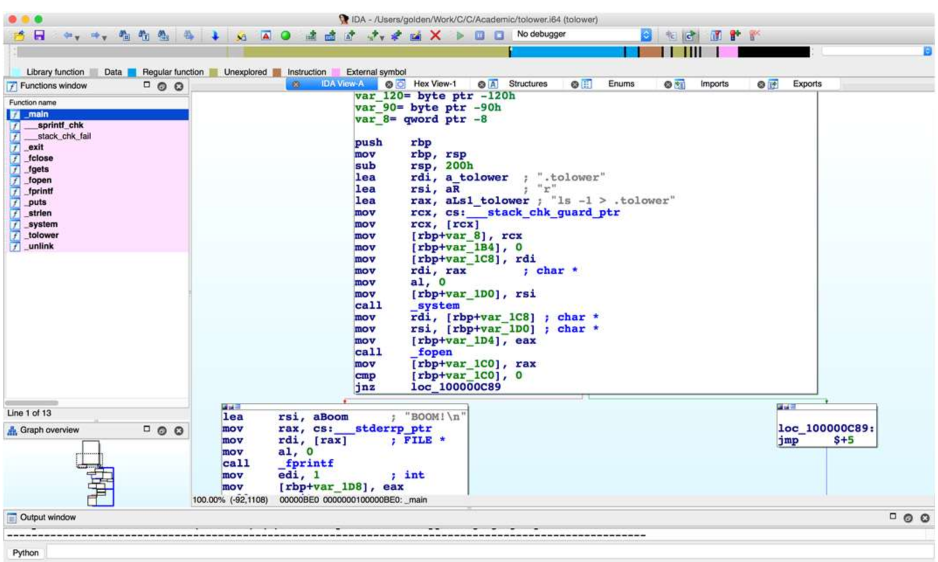
78

Decomp Example: Original Source

```
{
  char buf[132], buf2[132], lowbuf[132];
  int i;
  FILE *fp;

  system("ls -1 > .tolower");

  fp = fopen(".tolower","r");
  if (fp == NULL) {
    fprintf(stderr,"BOOM!\n");
    exit(1);
  }
  else {
    while (fgets(buf,131,fp) != NULL) {
      buf[strlen(buf) - 1] = NULL;
      for (i=0; i < strlen(buf); i++) {
        lowbuf[i]= tolower(buf[i]);
      }
      lowbuf[strlen(buf)] = NULL;

      /* don't mess up filenames of compressed files */

      if (strlen(lowbuf) > 2 && lowbuf[strlen(buf)-1] == 'z' &&
                                lowbuf[strlen(buf)-2] == '.') {
          lowbuf[strlen(buf)-1] = 'Z';
      }
      sprintf(buf2,"mv %s %s",buf,lowbuf);
      puts(buf2);
      system(buf2);
    }
    fclose(fp);
    unlink(".tolower");
  }
}
:
```

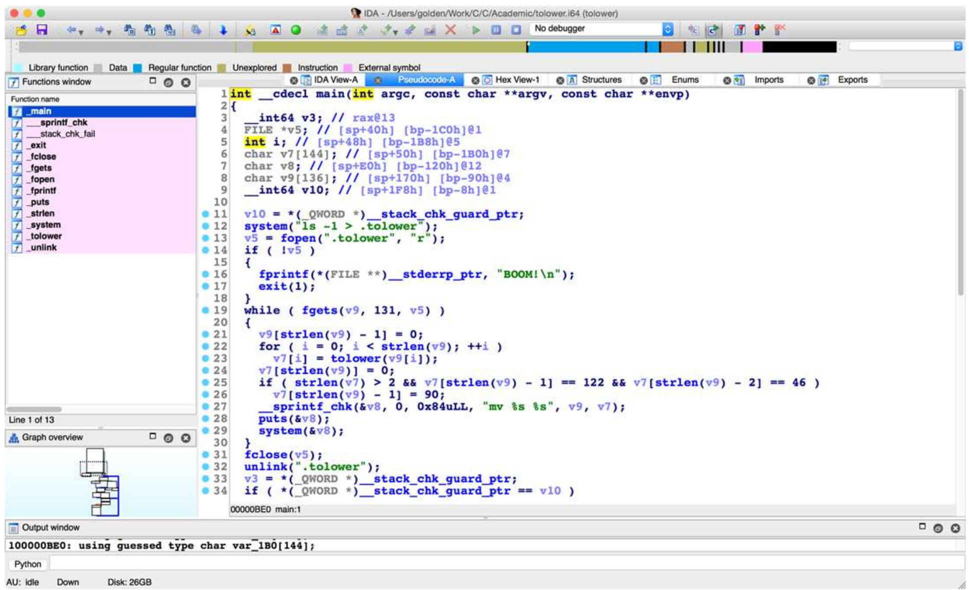
79

Decomp Example: Disassembly



80

Decomp Example: Decompilation



Georgia Tech

GHIDRA: Built-In Decompiler

- Ghidra comes with a decompiler out of the box!
  - One of the primary reasons for Ghidra's widespread adoption
- Synchronizes well with code viewer – highlights, renames, comments in one window show up in the other ☺
  - Can be configured to show more/less information if needed
  - e.g.: show unreachable code, hide variable casting, etc.
- ...Of course, since it's open source, there's already a port that put the code into IDA
  - https://github.com/Cisco-Talos/GhIDA

Georgia Tech

Lesson Summary

- Analyze executable files
- Evaluate sections, entry point, and dependencies of executable files.
- Describe the appropriate static executable inspection tool necessary for analysis and debugging.
- Discuss how to use GHIDRA to analyze and debug executable files.

Georgia Tech

83