Parsing Binary File Formats with PowerShell

Matt Graeber

@mattifestation

www.exploit-monday.com

PS> Get-Bio

- Security Researcher
- Former U.S. Navy Chinese linguist and U.S. Army Red Team member
- Alphabet soup of irrelevant certifications
- Avid PowerShell Enthusiast
  - Original inspiration: Dave Kennedy and Josh Kelley "Defcon 18 PowerShell OMFG…", Black Hat 2010
  - Continued motivation from @obscuresec
- Creator of the PowerSploit module
  - A collection of tools to aid reverse engineers, forensic analysts, and penetration testers during all phases of an assessment.
- Love Windows internals, esoteric APIs, and file formats

Why parse binary file formats?

- Malware Analysis
  - You need the ability to compare a malicious/malformed file against known good files.
- Fuzzing
  - You want to generate thousand or millions of malformed files of a certain format in order to stress test or find vulnerabilities in programs that open that particular file format.
- Curiosity
  - You simply want to gain an understanding of how a piece of software interprets a particular file format.

Why use PowerShell to parse binary file formats?

- Once parsed, file formats can be represented as objects
  - Objects can be inspected, analyzed, and/or manipulated with ease.
  - Its output can be passed to other functions/cmdlets/scripts for further processing.

- Automation!
  - Once a parser is written, you can analyze a large number of file formats, quickly perform analysis, and gather statistics on a large collection of files.
  - Example: You could analyze all known good file formats on a clean system, take a baseline of known good and use that as a heuristic to determine if an unknown file is potentially malicious or malformed.

Requirements

- A solid understanding of C/C++, .NET, and PowerShell data types is a must!
  - Windows C/C++ data types are described here:
    - http://msdn.microsoft.com/en-us/library/windows/desktop/aa383751(v=vs.85).aspx
  - C# value types are described here:
    - http://msdn.microsoft.com/en-us/library/s1ax56ch(v=vs.110).aspx

Validating data type equality

- Goal: Convert C/C++ DWORD to PowerShell type

MSDN Definition: DWORD – "A 32-bit unsigned integer. The range is 0 through 4294967295 decimal."

32-bit == 4 bytes

Best guess: **`[UInt32]`**

Validation steps:

1) Validate minimum value - **`[UInt32]::MinValue # 0`**
2) Validate maximum value - **`[UInt32]::MaxValue # 4294967295`**
3) Validate type size - **`[Runtime.InteropServices.Marshal]::SizeOf([UInt32]) # 4`**

DWORD == [System.UInt32]

Example: DOS Header

- The DOS header is a legacy artifact of the DOS era.
- The first 64 bytes of any portable executable file
  - .exe, .dll, .sys, .cpl, .scr, .com, .ocx, etc…
  - Size of the DOS header can be confirmed using my favorite debugger – WinDbg
  - `dt -v ntdll!_IMAGE_DOS_HEADER` or `?? sizeof(ntdll!_IMAGE_DOS_HEADER)`
- Per specification, the first two bytes of a DOS header are 'MZ' (0x4D,0x5A).
  - Trivia – What does MZ stand for?
- Nowadays, the only useful field of the DOS header is e_lfanew – the offset to the PE header.
- The fields of a non-malicious DOS header are relatively consistent.
  - To see an awesome abuse of the PE file format and DOS header, check out Alexander Sotirov's TinyPE project.

Example: DOS Header

Windows SDK winnt.h DOS header definition

```
#define IMAGE_DOS_SIGNATURE                 0x5A4D      // MZ
#define IMAGE_OS2_SIGNATURE                 0x454E      // NE
#define IMAGE_VXD_SIGNATURE                 0x454C      // LE

typedef struct _IMAGE_DOS_HEADER {      // DOS .EXE header
    WORD   e_magic;                     // Magic number
    WORD   e_cblp;                      // Bytes on last page of file
    WORD   e_cp;                        // Pages in file
    WORD   e_crlc;                      // Relocations
    WORD   e_cparhdr;                   // Size of header in paragraphs
    WORD   e_minalloc;                  // Minimum extra paragraphs needed
    WORD   e_maxalloc;                  // Maximum extra paragraphs needed
    WORD   e_ss;                        // Initial (relative) SS value
    WORD   e_sp;                        // Initial SP value
    WORD   e_csum;                      // Checksum
    WORD   e_ip;                        // Initial IP value
    WORD   e_cs;                        // Initial (relative) CS value
    WORD   e_lfarlc;                    // File address of relocation table
    WORD   e_ovno;                      // Overlay number
    WORD   e_res[4];                    // Reserved words
    WORD   e_oemid;                     // OEM identifier (for e_oeminfo)
    WORD   e_oeminfo;                   // OEM information; e_oemid specific
    WORD   e_res2[10];                  // Reserved words
    LONG   e_lfanew;                    // File address of new exe header
  } IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

Example: DOS Header

The DOS header is comprised of the following data types:

C Data Type	C# Data Type	PowerShell Data Type
`WORD`	`ushort`	`[UInt16]`
`WORD[]`	`ushort[]`	`[UInt16[]]`
`LONG`	`int`	`[Int32]`

Optional: An enum representation of e_magic since it contains only three possible, mutually-exclusive values.

Again, you can manually validate that these data types match – e.g.

- LONG->System.Int32. A 32-bit signed integer. The range is –2147483648 through 2147483647 decimal.
  - Min value: `[Int32]::MinValue`
  - Max Value: `[Int32]::MaxValue`
  - Size: `[System.Runtime.InteropServices.Marshal]::SizeOf([UInt32])`

Parsing binary file formats in PowerShell – Technique (1/3)

There are three ways to tackle this problem in PowerShell:

1. Easy - Pure PowerShell
2. Moderate – C# Compilation
3. Hard - Reflection

- Pure PowerShell – Strictly using only the PowerShell scripting language and built-in cmdlets
  - Pros:
    - Not complicated. Thus, easy to implement.
    - Works in PowerShell on the Surface RT tablet – i.e. PowerShell running in a 'Constrained' language mode.
  - Cons:
    - Very slow when dealing with large, complicated binary files

Parsing binary file formats in PowerShell – Technique (2/3)

- C# Compilation – Using the Add-Type cmdlet
  - Pros:
    - Structures and enums are easy to define and read when defined in C#
    - Many structures and enums are already defined for you on pinvoke.net.
    - After compilation occurs, this technique is much faster than the pure PowerShell approach when dealing with large, complicated file formats. Get-PEHeader in PowerSploit uses this approach.
  - Cons:
    - Doesn't work on the Surface RT tablet. You are restricted from using Add-Type.
    - Involves calling csc.exe and writing temporary files to disk in order to compile code. This is undesirable if you are trying to maintain a minimal forensic footprint.

Parsing binary file formats in PowerShell – Technique (3/3)

- Reflection – Manual assembly of data types in memory
  - Pros:
    - Fast, minimal forensic footprint (i.e. csc.exe not called and no temporary files created).
    - Ideally suited for parsing complicated, dynamic structures – i.e. structures that are defined based upon runtime information. Get-PEB in PowerSploit uses this technique.
  - Cons:
    - Doesn't work on the Surface RT tablet. You are restricted from using the .NET reflection namespace.
    - Reflection can be a difficult concept to grasp if you are not comfortable with .NET.

Reading a binary file in PowerShell

There are two generic methods for reading in a file as a byte array:

- Get-Content cmdlet
  - Great for reading small files
  - Works on the Surface RT tablet
  - You can optionally read a fixed number of bytes
  - Example: `Get-Content C:\Windows\System32\calc.exe -Encoding Byte -TotalCount 64`
- [System.IO.File]::ReadAllBytes(string path)
  - Quickly reads large files
  - Does not work on the Surface RT tablet
  - Reads all bytes in a file

Converting bytes to their respective data types

- Recall the following:

WORD == 16-bit unsigned number == 2 bytes

DWORD == 32-bit unsigned number == 4 bytes

LONG == 32-bit signed number == 4 bytes, etc...

- Note: many file formats store their values in little-endian so you must swap their values in order to read the proper values.

```
function Local:ConvertTo-Int
{
    Param (
        [Parameter(Position = 1, Mandatory = $True)]
        [Byte[]]
        $ByteArray
    )

    switch ($ByteArray.Length)
    {
        # Only convert words and dwords
        2 { Write-Output ( [UInt16] ('0x{0}' -f (($ByteArray | % {$_.ToString('X2')}) -join '')) ) }
        4 { Write-Output (  [Int32] ('0x{0}' -f (($ByteArray | % {$_.ToString('X2')}) -join '')) ) }
    }
}
```

Helper function to convert bytes into either a UInt16 or an Int32.

DOS header parsing script requirements

- Defines the necessary structures and enums present in the DOS header
- Reads in a file (or set of files) as a byte array via a function parameter or via the pipeline – i.e. BEGIN/PROCESS/END and ValueFromPipelineByPropertyName property
- Converts the flat byte array to a properly parsed DOS header – represented as either a custom object or a .NET type
- Only returns output from files with a valid DOS header size and e_magic field
- Displays a properly formatted DOS header using a ps1xml file.
  - I want all the fields to be displayed in hexadecimal rather than the default decimal.
- Provides detailed comment-based help

Building a DOS header parser Technique: Pure PowerShell

- This technique relies heavily on reading offsets into a byte array using array offset notation. For example:

```
PS> $array = [Byte[]] @(1,2,3,4,5,6)
PS> # If these were little-endian fields, then the array offsets
would need to be reversed.
PS> $array[1..0]
2
1
PS> $array[0..1]
1
2
```

- A custom object will be formed in this technique since the PowerShell scripting language has no way to define a native .NET type.
- You must be aware of the offsets to each field in the DOS header definition
- Demo: Source code analysis and script usage

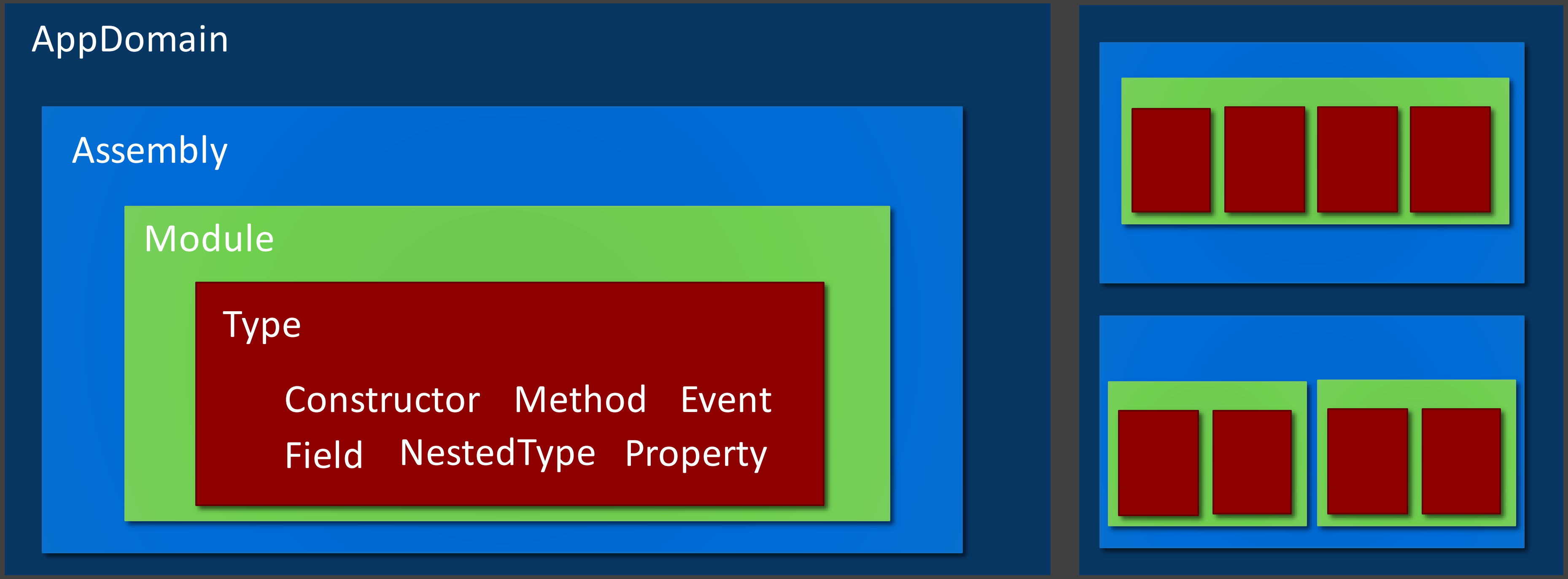
Building a DOS header parser Technique: C# Compilation

- The DOS header is defined in C# code. It is then compiled using the Add-Type cmdlet.
- After compilation, a custom .NET type is created and can be used directly in PowerShell - [PE+_IMAGE_DOS_HEADER] in out example.
- Note: Once a .NET type is defined, it cannot be redefined in the same PowerShell session. Restart PowerShell if you need to make changes to your C# code.
- The C# technique relies upon obtaining a pointer to our byte array and calling [System.Runtime.InteropServices.Marshal]::PtrToStructure to cast the array into a [PE+_IMAGE_DOS_HEADER] structure.
- Demo: Source code analysis and script usage

Building a DOS header parser Technique: Reflection (1/2)

- Rather than compiling our .NET type, we are going to manually assembly it.
- This technique, although more complicated to implement should be preferred to C# compilation if maintaining a minimal forensic footprint is your goal or if you are creating dynamic structures that must be defined at runtime.
- Reflection allows you to perform code introspection and code assembly. Requires a basic understanding of the .NET architecture.

.NET assembly layout



- AppDomain – An execution 'sandbox' for a set of assemblies
- Assembly – The dll or exe containing your code
- Module – A container for a logical grouping if types. Most assemblies only have a single module.
- Type – A class definition
- Members – The components that make up a type – Constructor, Method, Event, Field, Property, NestedType

Building a DOS header parser Technique: Reflection (2/2)

- The following steps are required to build the DOS header .NET type using reflection:

1. Define a dynamic assembly in the current AppDomain.
2. Define a dynamic module.
3. Define an enum type to represent e_magic values.
4. Define a structure type to represent the remainder of the DOS header.
5. e_res and e_res2 fields require custom attributes to be defined since they are arrays.

- Once the type is defined, a .NET type representing the DOS header will be defined and be nearly identical to the type created in C# previously.
- Demo: Source code analysis and script usage

The DOS header parser is complete. Now what?

- Let analyze every the DOS header of every PE in Windows!
- With analysis complete, we can find commonality across DOS headers and form the basis for what a ‘normal’ DOS header should look like.

Total PE Files	PE Files with non-standard MS-DOS stub programs	"Rich" Present	"Rich" Absent
6695	2	4431	2264

Extension	Count
.dll	3230
.mui	2028
.sys	718
.exe	539
.lrc	58
.cpl	21
.ax	15
.winmd	15
.rs	13
.tlb	13
.ocx	7
.com	7
.scr	6
.acm	6
.tsp	5
.efi	4
.rll	3
.drv	3
.ime	1
.olb	1
.dat	1
.iec	1

e_lfanew	Count
176	2207
240	1186
232	1000
248	794
224	568
256	318
200	194
216	168
192	91
264	56
128	33
272	28
64	23
280	13
296	6
288	4
208	3
184	1
312	1
360	1

e_cblp	Count
144	6670
0	23
96	2

e_cp	Count
3	6670
0	23
1	2

e_cparhdr	Count
4	6672
0	23

e_maxalloc	Count
65535	6672
0	23

e_sp	Count
184	6672
0	23

e_res	Count
{0, 0, 0, 0}	6677
{0, 0, 22094, 12336}	14
{0, 0, 22094, 12848}	4

e_ss	Count
0	6695

e_crlc	Count
0	6695

e_minalloc	Count
0	6695

e_csum	Count
0	6695

e_ip	Count
0	6695

e_cs	Count
0	6695

e_lfarlc	Count
64	6695

e_ovno	Count
0	6695

e_oemid	Count
0	6695

e_oeminfo	Count
0	6695

e_res2	Count
{0,0,0,...,0}	6695

Fields whose values are always 0x00

e_crlc, e_cparhdr, e_minalloc, e_ss, e_csum, e_ip, e_cs, e_ovno, e_oemid, e_oeminfo, e_res2

Fields whose values are always 64

e_lfarlc

While e_lfanew can be less than 64, normal PEs are never less than 64

e_lfanew is always divisible by four

Conclusion

- Parsing binary file formats in PowerShell is not a trivial matter. However, once structure is applied to a binary blob and is stored in an object, this is where PowerShell really shines.
- There are three primary strategies for parsing binary data in PowerShell: pure PowerShell, C# compilation, and reflection. Each strategy has their respective pros and cons.
- Parsing binary data in PowerShell requires knowledge of C-style structure definitions and data types.

Thanks!

- Twitter: @mattifestation
- Blog: www.exploit-monday.com
- Github: PowerSploit

Bonus: Rich Signature (1/2)

- Located between the DOS header and the NT header (i.e. PE header)
- An XOR encoded blob produced by Microsoft compilers and describes information about the linker used to link external dependencies.
- Not documented by Microsoft. Daniel Pistelli gives a throuough description here: http://ntcore.com/files/richsign.htm
- Completely useless part of a binary aside from being semi-useful in malware analysis.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0000h:	4D	5A	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00	MZ..........ÿÿ..
0010h:	B8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00	¸.......@.......
0020h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0030h:	00	00	00	00	00	00	00	00	00	00	00	00	F0	00	00	00	............ð...
0040h:	0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68	..º..´.Í!¸.LÍ!Th
0050h:	69	73	20	70	72	6F	67	72	61	6D	20	63	61	6E	6E	6F	is program canno
0060h:	74	20	62	65	20	72	75	6E	20	69	6E	20	44	4F	53	20	t be run in DOS
0070h:	6D	6F	64	65	2E	0D	0D	0A	24	00	00	00	00	00	00	00	mode....$.......
0080h:	5F	04	0F	B6	1B	65	61	E5	1B	65	61	E5	1B	65	61	E5	_..¶.eaå.eaå.eaå
0090h:	E7	12	DA	E5	19	65	61	E5	E7	12	DC	E5	35	65	61	E5	ç.Úå.eaåç.Üå5eaå
00A0h:	1B	65	60	E5	9C	64	61	E5	E7	12	D8	E5	04	65	61	E5	.e`åœdaåç.Øå.eaå
00B0h:	E7	12	DD	E5	02	65	61	E5	E7	12	A7	E5	44	65	61	E5	ç.Ýå.eaåç.§åDeaå
00C0h:	E7	12	D6	E5	1A	65	61	E5	E7	12	DB	E5	1A	65	61	E5	ç.Öå.eaåç.Ûå.eaå
00D0h:	52	69	63	68	1B	65	61	E5	00	00	00	00	00	00	00	00	Rich.eaå........
00E0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00F0h:	50	45	00	00	64	86	06	00	96	98	10	50	00	00	00	00	PE..d†..–˜.P....

Bonus: Rich Signature (2/2)

- After XOR decoding, the Rich signature is comprised of the following:
  - A Rich signature 'signature' – 'DanS'
    - Legend has it, 'DanS' is named after Dan Ruder - http://web.archive.org/web/20111219190947/http://mirror.sweon.net/madchat/vxdevl/vxmags/29a-8/Articles/29A-8.009
  - An array or three fields containing linker information:
    - Build Number
    - Product Identifier
    - Link Count
  - The word 'Rich' to indicate the presence of a Rich signature
  - The DWORD XOR used to decode the signature

  - Let's extend our DOS header parser to parse the Rich signature…