710.1

Code Deobfuscation and Execution

SANS | GIAC CERTIFICATIONS

THE MOST TRUSTED SOURCE FOR INFORMATION SECURITY TRAINING, CERTIFICATION, AND RESEARCH | sans.org

All reference links are operational in the browser-based delivery of the electronic workbook.

FOR710.1 | Reverse-Engineering Malware: Advanced Code Analysis

Code Deobfuscation and Execution

Section FOR710.1, also known as Section 1 of the FOR710 course, focuses on understanding code deobfuscation and execution in-depth.

FOR710.1 materials are created and maintained by Anuj Soni. To learn about Anuj's background and expertise, please see https://www.sans.org/instructors/anuj-soni. You can visit his blog at https://malwology.com/ and follow him on Twitter at https://twitter.com/asoni.

This page intentionally left blank.

FOR710 Assumes Prior Experience with Malware Analysis

- Due to the advanced nature of this course, the content assumes that you have prior experience performing malware analysis on Windows.
- As discussed in the course description, this class continues where FOR610 leaves off, helping students take their RE skills to the next level.
- Labs assume you are comfortable with the following topics:
  - Examining static properties of a file
  - Performing behavioral analysis and debugging of malicious PE files
  - Reading common x86 and x64 assembly instructions during code analysis
  - Identifying key assembly logic structures with a disassembler
  - Following program control flow to understand decision points in disassembly

This advanced malware analysis course assumes the participant has some prior experience analyzing malware. It is intended to enhance the student's beginner and intermediate level malware reverse engineering skills.

Although there are no formal prerequisites for this class, the content assumes that the student has knowledge and skills equivalent to those discussed in the SANS FOR610 "Reverse-Engineering Malware: Malware Analysis Tools and Techniques" course. Specifically, the student should have some experience performing static file properties analysis, behavioral analysis, dynamic code analysis (i.e., using a debugger), and static code analysis (i.e., analyzing disassembled executable content).

Course Roadmap

- FOR710.1: Code Deobfuscation and Execution
- FOR710.2: Encryption in Malware
- FOR710.3: Automating Malware Analysis
- FOR710.4: Correlating Malware and Building Rules

SECTION 1

- ***Analyzing Code Deobfuscation***
  - Lab 1.1: Investigating Code Deobfuscation Using Steganographic Techniques
- Identifying Program Execution
  - Lab 1.2: Analyzing Malicious Program Execution
- Understanding Shellcode Execution
  - Lab 1.3: Analyzing Shellcode Execution

This page intentionally left blank.

Analyzing Code Deobfuscation

This module discusses how to analyze code deobfuscation in the context of a sophisticated malware sample.

Malware May Deobfuscate Additional Code During Execution

- The initial code (i.e., loader) may unravel additional stages of execution.
- The decoded content may include shellcode and/or PE files embedded in a file on disk or data downloaded from the Internet.
- Advanced-level malware analysts must be prepared to analyze the details of deobfuscation algorithms.
- The result of this in-depth analysis helps the analyst understand the sophistication of the adversary, assess the uniqueness of the sample, generate reliable signatures (i.e., YARA rules), and build tools to automate deobfuscation.

It is common to encounter malware that deobfuscates additional code during execution. These additional layers of code, which may include a combination of shellcode and Windows executables, help evade detection and hinder analysis. The encoded content might reside within the original file, another file on disk, or the registry. Alternatively, the next-stage executable content may need to be downloaded from the Internet and then deobfuscated in memory. The initial malicious code is often referred to as the "loader" since it loads additional code and data.

While a deep understanding of a deobfuscation algorithm is not necessary in all cases, it is required if the analyst is responsible for a comprehensive and deep dive into the malware specimen. Understanding the specific methodology used to obfuscate and deobfuscate content can inform an assessment of the threat actor's sophistication, help build a reliable YARA signature, and provide the information necessary to automate deobfuscation.

Analyzing Code Deobfuscation: Module Objectives

- Understand code used to deobfuscate executable content.
- Be able to communicate the details of how code is obfuscated.
- Recognize key Windows APIs used to allocate memory.
- Differentiate user-defined code from library code.
- Dump deobfuscated executable content to disk.
- Explain how multiple files work together to execute malicious code.
- Develop comfort with non-binary formats during malware analysis.

This page intentionally left blank.

This Malicious Loader Example Downloads and Decodes a DLL

- FUN_1000a830 deobfuscates the DLL.
- It XORs a byte of data with a byte of the key.
- DIV divides EDX:EAX by the operand.
  - EAX contains the quotient (not used).
  - EDX contains the remainder which is used to iterate over key bytes.

```
10009c22  MOV   ECX, dword ptr [EBP + local_28]   ; Number of bytes
10009c25  PUSH  ECX                               ; Address of encoded content
10009c26  PUSH  0x25                              ; Key size
10009c28  PUSH  s_FuHZu4rQgn3eqLz6FB48Deybj49xEU... ; XOR key
10009c2d  CALL  FUN_1000a830                      ; Deobfuscates DLL
[redacted]
10009c4b  CMP   ECX, 0x5a4d                       ; Check for "MZ" bytes
```

```
FUN_1000a830
[redacted]
1000a834  MOV     dword ptr [EBP + counter], 0x0
1000a83b  JMP     LAB_1000a846

          LAB_1000a83d
1000a83d  MOV     EAX, dword ptr [EBP + counter]
1000a840  ADD     EAX, 0x1
1000a843  MOV     dword ptr [EBP + counter], EAX

          LAB_1000a846
1000a846  MOV     ECX, dword ptr [EBP + counter]
1000a849  CMP     ECX, dword ptr [EBP + num_bytes]
1000a84c  JNC     LAB_1000a872
1000a84e  MOV     EAX, dword ptr [EBP + counter]
1000a851  XOR     EDX, EDX
1000a853  DIV     dword ptr [EBP + key_size]
1000a856  MOV     EAX, dword ptr [EBP + addr_xor_key]
1000a859  MOVSX   ECX, byte ptr [EAX + EDX*0x1]
1000a85d  MOV     EDX, dword ptr [EBP + addr_encoded]
1000a860  ADD     EDX, dword ptr [EBP + counter]
1000a863  MOVSX   EAX, byte ptr [EDX]
1000a866  XOR     EAX, ECX
1000a868  MOV     ECX, dword ptr [EBP + addr_encoded]
1000a86b  ADD     ECX, dword ptr [EBP + counter]
1000a86e  MOV     byte ptr [ECX], AL
1000a870  JMP     LAB_1000a83d

          LAB_1000a872
1000a872  MOV     ESP, EBP
1000a874  POP     EBP
1000a875  RET     0x10
```


Let's explore how some malicious loaders deobfuscate the next stage of execution. The code on this slide is an excerpt of malware with SHA-256 hash 5c4c9f2ed1b908522a9f24e3c91f945fb24ce95ba209c24f6e97b116205898e3. This program downloads, deobfuscates, and executes a DLL in memory.

The code excerpt on the left shows a call to FUN_1000a830. This function deobfuscates downloaded content and produces a DLL in memory. The function takes four arguments which include a pointer to an XOR key, the key size, the address of the encoded content, and the number of bytes to deobfuscate. If you're wondering *how* we determined what each argument represents, don't worry – we'll get to that. For now, we just want to gain exposure to the operations that perform deobfuscation.

The screenshot on the right shows instructions within FUN_1000a830. The loop deobfuscates the DLL, and it includes various instructions that support this goal. Variables and arguments are renamed for clarity. Each byte of encoded data is XORed with a byte of the hardcoded key. When the loop reaches the last byte in the XOR key, it resets to the first byte of the key. The DIV instruction supports this "cyclic iteration". The DIV instruction divides EDX:EAX by the specified operand. The result is stored in EAX, and the remainder is stored in EDX. A colon bet between two registers indicates the values within the registers are concatenated. For example, if EDX contains 0x44332211 and EAX contains 0xffeeddcc, EDX:EAX is 0x44332211ffeeddcc.

For more details on this malware, see https://for710.com/mbloader.

Decompiler Output for FUN_1000a830 Clarifies Deobfuscation

- The modulo operator (%) results in the remainder of dividing operands.
- It helps iterate over the XOR key without exceeding the max index value.
- Example Key: ABCD
  - 0 % 4 = 0
  - 1 % 4 = 1
  - 2 % 4 = 2
  - 3 % 4 = 3
  - 4 % 4 = 0
  - 5 % 4 = 1

```
void FUN_1000a830(int addr_xor_key,uint key_size,int addr_encoded,uint num_bytes)

{
  uint counter;

  for (counter = 0; counter < num_bytes; counter = counter + 1) {
    *(addr_encoded + counter) = *(addr_encoded + counter) ^ *(addr_xor_key + counter % key_size);
  }
  return;
}
```


This slide shows Ghidra's Decompile (i.e., pseudocode) output for the function discussed on the previous slide. While it's important to understand the assembly code representation, reading the pseudocode is certainly easier. Again, variables and arguments are renamed for clarity.

Observe the for loop that XORs a byte of encoded data with a byte of the hardcoded key. In the previous slide, we saw a DIV instruction, but here we see a modulo (%) operator; these operations are related. The modulo operator produces the remainder when dividing one operand by another. For example, in the expression 10 mod 2, 10 /2 = 5 with no remainder. In this expression, 2 is the *modulus* and the result is zero. In the expression 10 mod 3, 10/3 = 3 with a remainder of 1. In this second expression, 3 is the modulus and 1 is the resulting remainder. The modulo operator allows a value to increase and then reset once it hits a certain limit defined by the modulus. Using the example key "ABCD" on this slide, performing a modulo operation against the key length (4) ensures that as a counter (first operand) increases, the result never exceeds the max index value for the key (3).

A Different Loader Deobfuscates Content on Disk

- This loader decodes a file (`mpc.tmp`) on disk.
- The pseudocode reveals a similar decoding process, but the disassembly is more complicated.

```
do {
  *addr_data = *addr_data ^ xor_key[counter % 0xe4];
  num_bytes = num_bytes - 1;
  counter = counter + 1;
  addr_data = addr_data + 1;
} while (num_bytes != 0);
```

- Compilers may use *magic number division* to optimize performance.

```
180001bc5   MOV     R9, -0x7047dc11f7047dc1
180001bcf   NOP

                LAB_180001bd0
180001bd0   MOV     RAX, R9
180001bd3   INC     R11
180001bd6   MUL     RCX
180001bd9   MOV     RAX, RCX
180001bdc   INC     RCX
180001bdf   SHR     RDX, 0x7
180001be3   IMUL    RDX, RDX, 0xe4
180001bea   SUB     RAX, RDX
180001bed   MOVZX   EAX, byte ptr [RBP + RAX*0x1 + 0x750]
180001bf5   XOR     byte ptr [R11 + -0x1], AL
180001bf9   DEC     R8
180001bfc   JNZ     LAB_180001bd0
```


Let's look at a different malware loader with SHA-256 hash 850fad00f55153be1338382cdbc68a28292028e1213f72ea7d2f1c632c4719b7. Instead of downloading the next-stage executable content from a server, this program interacts with another file (mpc.tmp) on disk. The deobfuscation pseudocode is shown on the left side of this slide. The algorithm is similar to the previous example and involves XORing the encoded content with a defined key. Using XOR for simple deobfuscation is quite common in malware.

However, you might be surprised to see the corresponding disassembly on the right side of this slide. Instead of a DIV instruction commonly associated with a modulus operation, we have MUL (multiply), IMUL (signed multiply), and SHR (shift right) operations, among others. Also, observe the large, signed value at 180001bc5 which is moved into R9 and then used within the loop. This is not the result of clever programming or an obfuscation technique. Instead, it is an artifact of a compiler optimization that uses *magic number division*. Performing division is generally more costly (from a performance perspective) than performing multiplication. As a result, the compiler in this case decided to use a combination of other mathematical instructions and the observed signed value to execute the code faster. The result is the same as using a DIV instruction and the pseudocode helps confirm this. We will not spend time digging into the mathematics of these instructions, but it is important to be aware of this optimization.

For additional discussion regarding magic number division, see https://for710.com/magicdiv.

For more information on this malware, see https://for710.com/bronze.

Deobfuscation Can Include Decryption

- A variant of the malware just discussed *decrypts* contents of a file on disk (`vm.cfg`).
- The code includes XOR but is more complicated than earlier examples.
- This is an RC4 implementation used to decrypt a Cobalt Strike beacon.
- We'll discuss encryption and the details of RC4 in Section 2.

```
FUN_180001000(local_118);
lVar7 = DAT_18000cfd8;
uVar8 = 0;
uVar4 = 0;
if (0 < lVar7) {
  pbVar5 = pbVar2;
  do {
    uVar8 = uVar8 + 1 & 0x800000ff;
    if (uVar8 < 0) {
      uVar8 = (uVar8 - 1 | 0xffffff00) + 1;
    }
    lVar6 = uVar8;
    bVar1 = local_118[lVar6];
    uVar4 = uVar4 + bVar1 & 0x800000ff;
    if (uVar4 < 0) {
      uVar4 = (uVar4 - 1 | 0xffffff00) + 1;
    }
    local_118[lVar6] = local_118[uVar4];
    local_118[uVar4] = bVar1;
    uVar3 = local_118[lVar6] + bVar1 & 0x800000ff;
    if (false) {
      uVar3 = (uVar3 - 1 | 0xffffff00) + 1;
    }
    lVar7 = lVar7 + -1;
    *pbVar5 = local_118[uVar3] ^ (pbVar5 + 1)[(param_1 - pbVar2) + -1];
    pbVar5 = pbVar5 + 1;
  } while (lVar7 != 0);
```


This slide covers a variant of the malicious loader discussed on the previous slide. This sample's SHA-256 hash is 7acc90887bafd261352f4e52e6a73252de2196a6e1a91dde77e3be7dda371836. The loader decrypts the contents of a file named vm.cfg to reveal a Cobalt Strike beacon in memory. The code on this slide implements the RC4 algorithm instead of the simple XOR approach discussed earlier. RC4 does include an XOR operation, but it is just one component of the algorithm. We will discuss encryption and the specifics of RC4 in detail in Section 2. This initial exposure is just to reinforce the spectrum of decoding routines you will encounter when analyzing malicious loaders.

Loaders May Use Steganographic Techniques to Reveal Code

- Steganography is the art of hiding a message or file within another file.
- The technique is often referred to as "hiding in plain sight."
- Malware authors use steganography techniques to hide code/data and introduce additional stages of execution to hinder detection and analysis.
- Approaches range from simply appending data to the end of a file to interweaving data throughout the entire file.
- The publicly available Invoke-PSImage embeds a PowerShell script within the pixels of a PNG image file.

Malicious loaders may rely on steganographic techniques to deobfuscate the next stage of execution. Steganography is an approach to hiding code, a file, or some other data within another file, ideally without raising suspicion—this is why steganography is often referred to as "hiding in plain sight." For comparison, while the goal of encryption is to protect confidentiality of some data, the goal of steganography is to hide the fact that there is any interesting data at all.

Attackers have hidden code and data within graphics file formats for years. Some have used publicly available tools to facilitate this data hiding, while others employ custom techniques. Image file formats are the most common vehicle for hiding data since the visual image may distract the viewer from any embedded hidden data. For an example of malware that hides executable content within a bitmap image, see https://for710.com/blackberry-windealer.

While custom code may be used to employ steganography, public tools can assist. For example, the publicly available Invoke-PSImage script hides a PowerShell script within the pixels of a PNG image file.

Let's cover an example of malware that use steganography to evade detection and analysis.

A Simple Steganographic Approach Involves Appending Content

- Opening this file in an image viewer shows a JPG icon identical to the one in this target file's icon:

data.png

- Closer analysis requires an understanding of the file structure, including the file signature, header, and trailer.
  - A PNG file begins with:

Offset(h)	00	01	02	03	04	05	06	07	Decoded text
00000000	89	50	4E	47	0D	0A	1A	0A	‰PNG....

  - The image file is comprised of "chunks" that describe the file's content.
  - The first chunk type is `IHDR`, and the final chunk type is `IEND`.
  - Each chunk ends with a 4-byte CRC.

One common, straightforward approach to hiding content in a file is simply to append data to the file. In most cases, this does not impact typical usage of the file, though it may bloat the file size.

In our first example, we will discuss a file named data.png. You can find this file at Malware\Section1\data_png.zip within the course VMs. The file was reportedly delivered to potential victims via email (https://for710.com/trustwave-lokibot). A review of its header shows that the first few bytes match those you would expect to see in a legitimate Portable Network Graphics (PNG) file. Oddly, the preview of the file in its icon refers to a JPG graphic, another common graphics file format. This may be an attacker's attempt to confuse the analyst, but it may also cause more scrutiny.

The use of steganography forces an analyst to investigate additional file types and structures not typically encountered during binary analysis. In this case, digging deeper requires that we understand the basic structure of a PNG file.

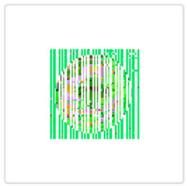
A PNG file begins with the 8-byte hexadecimal header: 89 50 4E 47 0D 0A 1A 0A. This file type is predominantly compromised of chunks that, in aggregate, describe the image. The first type of chunk is "IHDR", and the last type of chunk is "IEND". For more information about the PNG header and trailer specification, see the following:
https://for710.com/wikipng
https://for710.com/w3png

Identifying the EOF Marker May Help Extract the Target Data

- Appending data to a file may not impact normal usage of the file.
- This file has content after the final chunk, which may be a PKZip file.

```
0000A610   00 49 45 4E 44 AE 42 60   .IEND®B`
0000A618   82 50 4B 03 04 14 00 00   ‚PK.....
```

- Unzipping data.png with 7.zip reveals an executable:



RFQ
-5600005870.exe

property	value
sha256	C9DDCF7D0CD026CDEAC9586515B4D591C1CA63EE9C009CD00B198178E5E84F03
first-bytes-text	M Z . . . . . @ . . . . .
file-size	13877248 (bytes)
signature	Microsoft Visual Basic v5.0
description	pirlfORm LTd
file-type	**executable**

As mentioned earlier, when steganography is used wisely, it does not impact normal usage of the target file. With the correct PNG extension, data.png opens up in an image viewer without issue. However, if we identify the file IEND chunk, we will see additional data appended to the file beginning with the characters "PK." These two characters are often associated with the ZIP file format.

To unzip the embedded file, we could try using 7-Zip. Oddly, if we rename the file to a .zip extension and try 7-Zip, it encounters an error. However, if we simply remove the extension and right-click and choose 7-Zip > Extract Here, it unzips without issue. This results in an extracted executable named RFQ -5600005870.exe. Opening this file in PeStudio confirms it is a Windows executable.

This is a simpler case of steganography, where the hidden file is simply appended to the original file.

For another example of malware that employs steganography, read about the corelump loader: https://for710.com/corelump. This malware downloads a JPEG image that contains an encrypted executable after the JPEG end of file marker.

A More Sophisticated Approach Involves Manipulating Bit Values

- Replacing individual bits of an image file with the bits of a hidden file or message is usually unnoticeable to the human eye.
- A common approach involves modifying the least significant bit (LSB) of bytes within an image.

Original image

Altered image

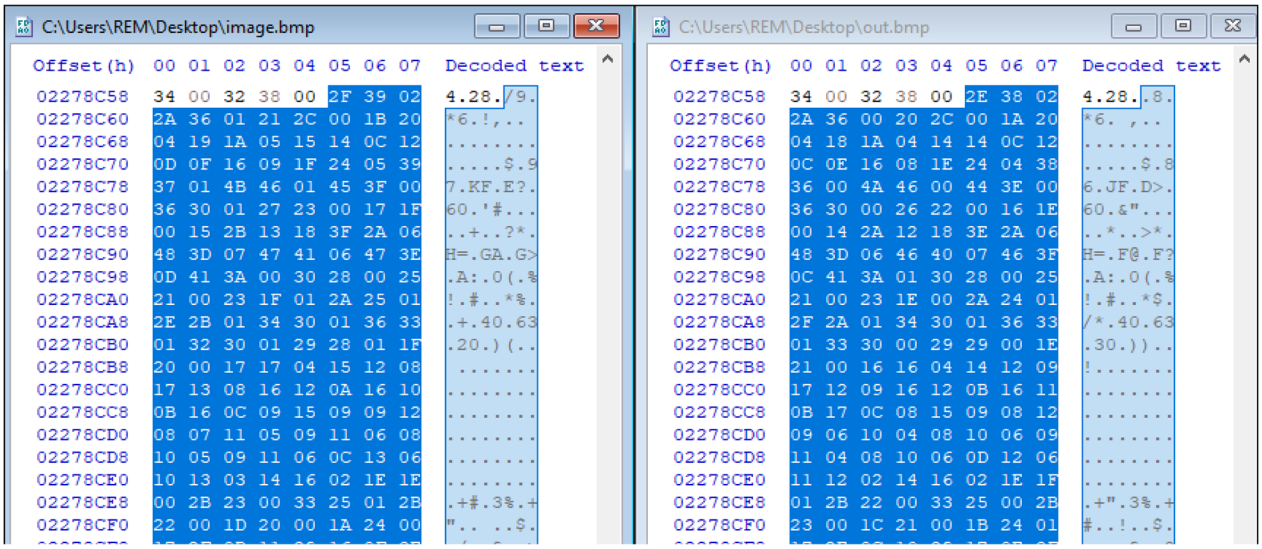
A more sophisticated approach to hiding data within a file involves manipulating bits to interweave the embedded content. This far more nuanced than appending content to a file and significantly harder to detect.

As an example, the 24-bit bitmap image files (BMP) on this slide appear identical, but there is an important difference. The image on the right has a secret message embedded within it. This message is stored within the least significant bits of individual pixel color values (Red, Green, and Blue). Changes made to the original image to store the hidden content are not perceivable to the human eye.

The altered image was created using the LSBSteg.py python script available at https://github.com/RobinDavid/LSB-Steganography. Specifically, the following command line embeds a secret image in image.bmp to product out.bmp:

```
python LSBSteg.py encode -i image.bmp -o out.bmp -f secret.txt
```

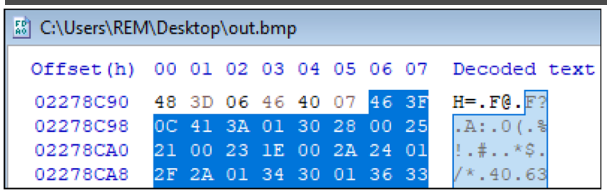
A Byte Comparison Shows Many One-Bit Differences



To better understand the impact of embedding content within the original image, we can perform a byte level comparison using a hex editor (HxD is used on this slide). From the menu bar, we browse to Analysis > Data comparison > Compare, and choose the original and modified files.

Looking through the output shows, among many differences, the chunk of bytes on this slide. A visual comparison shows that, in many cases, there is a one-bit difference between byte values. For example, the first value highlighted in image.bmp is 0x2F and the byte at that same position in out.bmp is 0x2E. The same holds true for the second value—it is 0x39 in image.bmp and 0x38 in out.bmp.

Extracting Least Significant Bits (LSBs) Reveals Hidden Data



Hex	46	3F	0C	41	3A	01	30	28
Binary	01000110	00111111	00001100	01000001	00111010	00000001	00110000	00101000
LSB	0	1	0	1	0	1	0	0
	01010100 = 0x54 = T							

"This is a secret."

If we begin extracting the least significant bits (LSBs) of these values in out.bmp, eventually we reach the the bytes highlighted on this slide. The table shows the byte under review, its binary form, and the least significant bit. Aggregating these individual bits to form a single byte reveals the ASCII character "T".

If we continue extracting LSBs in this manner to form bytes, we eventually create the text "This is a secret."

Extracting the LSB Typically Involves Bitwise Operations (2)

```
00401216    PUSH    EDI                              ; Address of storage buffer (content read from file placed here)
00401217    CALL    _fread                           ; size_t _fread(void * _DstBuf, size_t _ElementSize, size_t _Count
00401245    MOV     EAX, 0x1e                        ; Places decimal 30 into EAX
0040124a    MOV     dword ptr [EBP + local_34], EDI  ; Places pointer to file data into local_34

                    LAB_00401258                                  XREF[1]:   004012bb(j)
00401258    MOV     ECX, dword ptr [EBP + local_34]  ; Places pointer to data into ECX
0040125b    TEST    byte ptr [ECX], 0x1              ; Evaluates LSB of one byte of data
0040125e    JZ      LAB_0040126d                     ; If LSB is zero, jumps to next evaluation
00401260    LEA     ECX, [EAX + 0x1]                 ; If LSB is 1, places bit position into ECX (starting with 31)
00401263    MOV     EBX, 0x1                         ; Places 1 in EBX
00401268    SHL     EBX, CL                          ; Shifts left based on bit position
0040126a    OR      dword ptr [EBP + local_30], EBX  ; Incorporates bit into existing decoded value
                    LAB_0040126d                                  XREF[1]:   0040125e(j)
0040126d    MOV     ECX, dword ptr [EBP + local_34]
00401270    TEST    byte ptr [ECX + ESI*0x1], 0x1    ; Evaluates another byte (debugging shows it skips one in between)
00401274    JZ      LAB_00401282
00401276    MOV     EBX, 0x1
0040127b    MOV     ECX, EAX
0040127d    SHL     EBX, CL
0040127f    OR      dword ptr [EBP + local_30], EBX
                    LAB_00401282                                  XREF[1]:   00401274(j)
```


This slide shows code responsible for extracting LSBs from file data. See the end of line comments for additional detail.

Course Roadmap

- FOR710.1: Code Deobfuscation and Execution
- FOR710.2: Encryption in Malware
- FOR710.3: Automating Malware Analysis
- FOR710.4: Correlating Malware and Building Rules

SECTION 1

- Analyzing Code Deobfuscation
  - ***Lab 1.1: Investigating Code Deobfuscation Using Steganographic Techniques***
- Identifying Program Execution
  - Lab 1.2: Analyzing Malicious Program Execution
- Understanding Shellcode Execution
  - Lab 1.3: Analyzing Shellcode Execution

This page intentionally left blank.

Lab 1.1: Background Topics: Windows Memory Allocation (1)

- Deobfuscation may involve allocating memory for the decoded content.
- The Windows memory manager provides services to allocate memory, share it between processes, assign permissions, and map files into memory.
- **Virtual API**:
  - Functions: VirtualAlloc, VirtualProtect, VirtualFree
  - Lowest level Microsoft API for memory allocation
  - Allocates a minimize size of 64K from free memory
- **Heap API**:
  - Functions: HeapCreate, GetProcessHeap, HeapAlloc, HeapFree
  - Ideal for smaller allocations; for larger allocations, VirtualAlloc is called

Before malware decodes the next layer of execution, it may have to allocate memory for the decoded content. The Windows memory manager provides services to allocate memory, share it between processes, assign permissions, and map a file into memory. We will discuss four approaches to allocating memory.

Virtual API: This Windows API includes the functions VirtualAlloc, VirtualProtect, and VirtualFree, among others. It is the lowest level API for memory allocation in the Windows API. It allocates a minimum of 64K. This makes it inefficient for smaller memory allocations.

Heap API: This Windows API includes the functions HeapCreate, GetProcessHeap, HeapAlloc, and HeapFree, among others. It is used to allocate smaller sections of memory (less than a page). From a reverse engineering perspective, setting breakpoints on these APIs can be problematic because they are simply called too frequently. Each process has a default heap it can use for allocations, and additional heaps can be created. The Heap API uses `VirtualAlloc` internally to allocate larger chunks of memory.

You may also encounter `LocalAlloc` and `GlobalAlloc` APIs during your analysis. In recent versions of Windows, these serve as wrapper functions for `HeapAlloc` and have more overhead. More information on the differences between these similar APIs is available at the links below. From the malware author's perspective, these APIs simply provide more options for memory allocation.

For additional documentation on Windows memory management, see the following resources:
https://for710.com/memory-allocation
https://for710.com/memory-management-functions

For additional detail, also see the "Memory Management" chapter in the venerable Windows Internals, Part 1.

Lab 1.1: Background Topics: Windows Memory Allocation (2)

- **malloc**:
  - Function to perform dynamic memory allocation
  - Part of the standard C library, though it can also be used in C++
  - On Windows, it calls HeapAlloc
- **new operator**:
  - It's an operator (not a function) used in C++ programs only
  - It invokes the function `operator new`
  - On Windows, it calls HeapAlloc

Malloc: `malloc` is a standard C library function for allocating memory. On Windows systems, it will call HeapAlloc. When developers use `malloc` to allocate memory, they should use `free` to deallocate memory.

New operator: This operator (not function) is available in C++, and it invokes the function "operator new". Similar to `malloc`, it will call HeapAlloc on Windows systems. When developers use `new` to allocate memory, they should use `delete` to free memory. See https://for710.com/new for more information.

Both `malloc` and `new` allocate memory on the heap.

Lab 1.1: Background Topics: Win API vs. C/C++ Libraries (1)

- When performing code analysis, Windows API calls are sometimes buried deep within function calls; other times, they are closer to the entry point.
- This is one difference between a program developed using the Windows API directly vs. a program that uses standard C/C++ libraries.
- Programs compiled with C/C++ libraries can be more challenging to follow.
- A program developed using the Windows API will *only* run on Windows.
- A program using the standard C/C++ libraries can be compiled for multiple OS's; but when compiled for Windows, it *will* call Windows APIs.

We may have to work our way through layers of standard library code to arrive at the user code, where the malicious content of interest usually resides.

While we will not spend much time on the development of malware, we need to be familiar with some development concepts to improve our understanding of the corresponding disassembly.

Malware is often developed using C or C++. When building a program for the Windows operating system, a developer may choose, among other options, to use the C/C++ runtime libraries or use the Windows API (WinAPI) directly.

Writing code using C/C++ libraries means the program can be compiled for multiple operating systems (i.e., it is cross platform). C/C++ runtime libraries are implemented on top of the operating system, so they will use the underlying APIs of the operating system. This means that a program using C/C++ runtime libraries that is compiled for Windows *will* call the Windows API functions—but they will be referenced under multiple layers of library calls.

Alternatively, when a developer uses the WinAPI directly, the code can *only* be compiled for the Windows operating system. A benefit of this approach is that Windows API functions often expose more functionality than the standard C/C++ library functions.

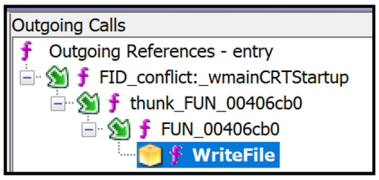
From a code analysis perspective, programs that use C/C++ libraries can be a bit more challenging to analyze due to the numerous layers of functions calls. Usually, our goal is to analyze code written by the malicious developer, not standard library code. This means we may need to work our way through the various layers of library code to locate code written by the developer.

Lab 1.1: Background Topics: Win API vs. C/C++ Libraries (2)

C++ Code Using WinAPI

```
1  #include <windows.h>
2  #include <stdio.h>
3
4  void main()
5  {
6      HANDLE hFile;
7      char DataBuffer[] = "I just wrote this!";
8      DWORD dwBytesToWrite = (DWORD)strlen(DataBuffer);
9      DWORD dwBytesWritten = 0;
10     BOOL bErrorFlag = FALSE;
11
12     hFile = CreateFile("output_winapi.txt", GENERIC_WRITE, 0,
13     NULL, CREATE_NEW, FILE_ATTRIBUTE_NORMAL, NULL);
14
15     bErrorFlag = WriteFile(hFile, DataBuffer, dwBytesToWrite,
16     &dwBytesWritten, NULL);
17
18     CloseHandle(hFile);
19 }
```

Function Call Tree



Function call trees provide valuable context for function calls.

When comparing a program that uses the Windows API directly vs. a program that uses C/C++ libraries, the flow of execution looks different.

On the left of this slide, we see code that uses the Windows API (via windows.h). It is a simple program that writes text to a file. It includes the Windows API functions `CreateFile, WriteFile,` and `CloseHandle.` The code on the left is derived from the example here: https://docs.microsoft.com/en-us/windows/win32/learnwin32/your-first-windows-program.

We'll focus on `WriteFile` to illustrate a point.

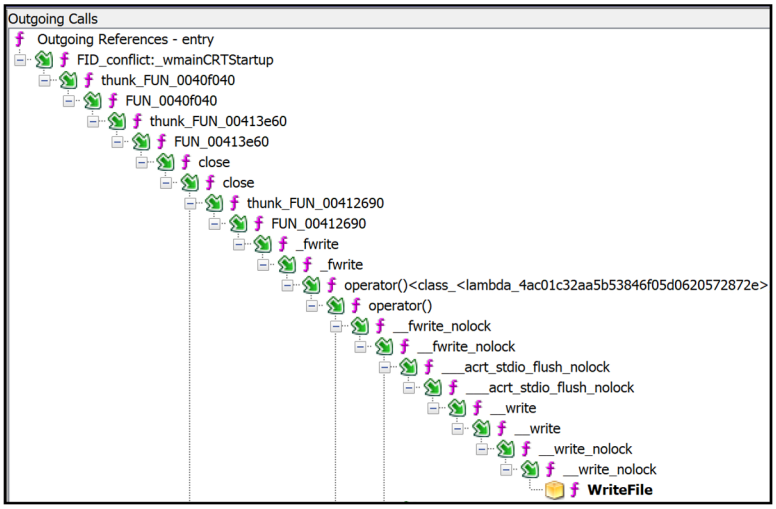
On the right, we have the calls that occur along the path between the entry point and a call to `WriteFile`. Notice the path is short, and there are only a few functions in between.

Lab 1.1: Background Topics: Win API vs. C/C++ Libraries (3)

C++ Code Using fstream Class

```
1   #include <iostream>
2   #include <fstream>
3   using namespace std;
4
5   int main()
6   {
7     // Create and open a file
8     ofstream ExampleFile("output.txt");
9
10    // Write to the file
11    ExampleFile << "I just wrote this!";
12
13    // Close the file
14    ExampleFile.close();
15  }
```

Function Call Tree



Now, let's look at a program that does the same thing as the program on the previous slide (i.e., it writes the same text to a text file), but this program uses the C++ fstream class for operating on files. There is no explicit mention of `WriteFile` in the code because this program uses C/C++ functions rather than the Windows API. However, when compiled for windows, it does call `WriteFile` to write a file to disk. The function call tree on this slide shows a reference to `WriteFile` under multiple layers of function calls.

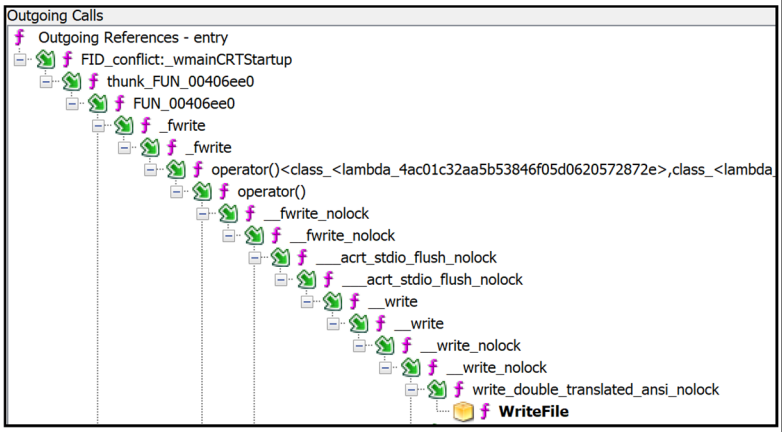
While this function call tree is more complicated, Ghidra assists us by automatically recognizing many C/C++ library functions (more on this shortly).

Lab 1.1: Background Topics: Win API vs. C/C++ Libraries (4)

C++ Code Using C-style file I/O

```
1  #include <cstdio>
2
3  int main()
4  {
5      FILE *pFile;
6      char buffer[] = "I just wrote this!";
7
8      pFile = fopen("output.txt","wb");
9      fwrite(buffer,sizeof(buffer),1,pFile);
10     fclose(pFile);
11
12     return 0;
13 }
```

Function Call Tree



The source code on this slide writes to a text file, like the prior two slides. However, this program uses C-style functionality, making use of fopen, fwrite, and fclose. The result is the same—when compiled for Windows, `WriteFile` is called under many layers of function calls.

As malware analysts, our goal is to analyze code written by the developer, not standard library code. When analyzing a program that uses C/C++ libraries, we may need to work our way through more layers of standard library code to arrive at the code written by the attacker. As we tackle our first lab, we will learn how to navigate this situation in Ghidra.

Lab 1.1: Background Topics: EXE Entry Point Terminology

- **EntryPoint:** The start of executable content specified in the AddressOfEntryPoint field in the optional header.
- **WinMain:**
  - The user-defined entry point for a graphical Windows application.
  - If built with Visual Studio, **WinMainCRTStartup** will call this function.
- **main:**
  - The user-defined entry point for a C++ console application.
  - If built with Visual Studio, **mainCRTStartup** will call this function.
- Code between the EntryPoint and WinMain/main is likely generated by the compiler.

During in-depth reverse engineering efforts, it is often beneficial to perform code analysis from the entry point. This provides insight into both code functionality and the order in which it occurs during execution. However, our efforts should focus on user-defined functionality and not compiler generated code. The following slides clarify entry point terminology and approaches to identifying code written by the malware author.

When an EXE is launched, certain initialization activities occur. The entry point (specified in the AddressOfEntryPoint field in the optional header of an EXE) directs the system to the code that performs these setup activities.

WinMain is the user-defined entry point for GUI applications while **main** is the user-defined entry point for console applications. PeStudio can help you determine if a program is a GUI or console application. Note that a GUI application is not required to have graphical elements, but it can.

If an EXE is built using Visual Studio, the Visual C++ run-time library (VCRuntime) provides an entry point called WinMainCRTStartup or mainCRTStartup for GUI or console applications, respectively. In this case, the code at the entry point will call one of these startup functions, and the startup function will call `WinMain` or main.

You may also encounter `wWinMain` or wmain—these are similar to `WinMain` and main, but for Unicode command line arguments.

A different user-defined entry point can be specified using the /ENTRY linker option (https://for710.com/entryoption).

From a malware analyst's perspective, there is no functional difference between `` `Winmain` `` and `` `main` ``—both signal the beginning of the developer's code. However, you will encounter both terms while performing and reading about malware analysis, so it's helpful to know why programs refer to one entry point name versus another.

A key take away is that the code between the EntryPoint and Winmain or main is likely generated by the compiler, and therefore it can often be ignored. As a malware analyst, you want to focus your attention on code written by the malware author.

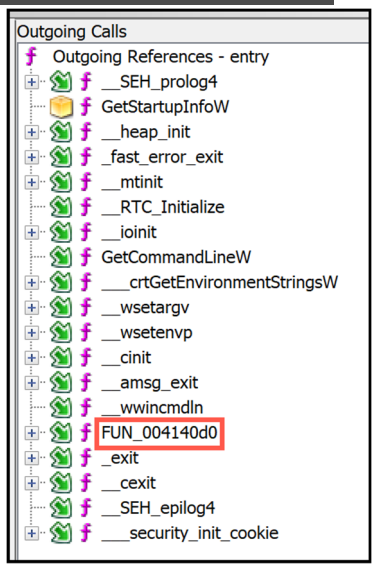
For more information on entry points, see:
https://for710.com/winmain-ep
https://for710.com/winmain
https://for710.com/entrypoint

Lab 1.1: Background Topics: Identifying WinMain (2)

- Review function calls after the entry point.
- The FID analyzer identifies many library functions.
- Inspect functions after the command line is retrieved.
- FUN_004140d0 may be WinMain.

```
                    entry
004014de    CALL    ___security_init_cookie
004014e3    JMP     LAB_00401361
```

```
                    LAB_00401361
00401361    PUSH    0x58
00401363    PUSH    DAT_0041a670
00401368    CALL    __SEH_prolog4
```



To discuss approaches for locating WinMain, we will use the Windows executable with SHA-256 hash 0BF8F22C889018E03C85EA73BBACBF00C3EB714D918F5B4C34F6876969788B1A. You can find this executable at Malware\Section1\gui_example.zip within the course VM.

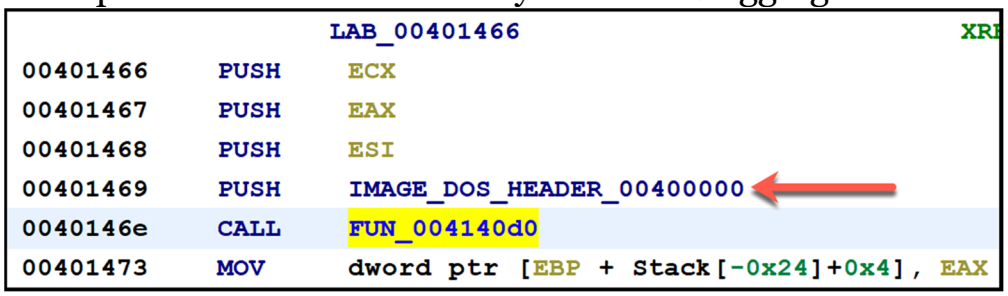
First, browse to the entry point via the Symbol Tree or Functions window. At 401fde, we first see a CALL to an FID-identified library function __security_init_cookie. While a detailed discussion of this function is out of scope of this course, it is worth noting that this is a C Runtime (CRT) function and names beginning with a single or double underscore generally represent reserved functions. You can read more about this function here: https://for710.com/securitycookie.

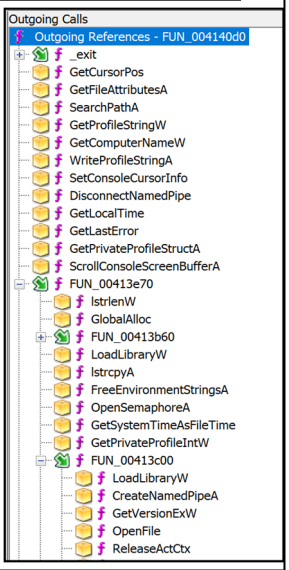
At 4014e3, we see a JMP instruction. Following the JMP takes us to 401361 and we see additional calls to library functions. We can use the function call tree for an overview of all functions called from the entry point.

When identifying `WinMain` candidates, focus on functions that are not identified as library functions. In this case, there is only one—FUN_004140d0. However, we can look for additional evidence to support the theory that this function is WinMain. For example, the previous slide discussed that WinMain requires the command line as the third argument. Looking above FUN_004140d0 in the function call tree (pictured on the right) we can see multiple functions that could contribute to retrieving the command line, including `GetCommandLineW` and __wwincmdln. Keep in mind that there are multiple functions that can retrieve command line information—these are just examples.

Lab 1.1: Background Topics: Identifying WinMain (3)

- The first argument passed to FUN_004140d0 points the file in memory.
- FUN_004140d0 references few reserved functions.
- This function is WinMain, though confirming this may require additional code analysis and debugging.



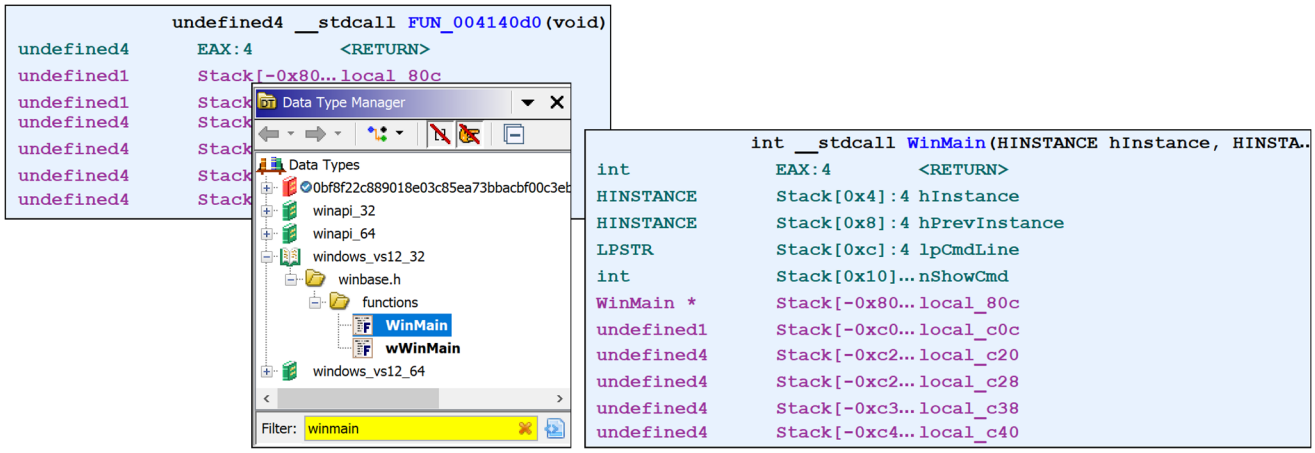


As a reminder, Microsoft documentation explains that WinMain's first argument points to the executable in memory. At 401469, the PUSH instruction passes the address of the DOS header. This observation supports our hypothesis that FUN_004140d0 is WinMain.

Also, FUN_004140d0 references only one reserved function (i.e., functions beginning with a single or double underscore) and many Windows API functions. This is typical of user-defined code, so this observation supports our theory as well.

Lab 1.1: Background Topics: Identifying WinMain (4)

If Ghidra does not recognize WinMain arguments, we can apply the appropriate function signature from the Data Type Manager.



When you browse to FUN_004140d0 within Ghidra, you may notice it displays no arguments. Ghidra did not correctly identify the arguments passed to this function. We have high confidence this function is WinMain, so we can apply the function signature from the Data Type Manager. Search for "winmain" and choose the option shown on this slide (this program is 32-bit, so we choose an option from a 32-bit data type archive). Then, drag-and-drop the selection to the function name (above the list of arguments and/or variables).

Lab 1.1: Background Topics: Identifying Main (1)

We can identify the main function for console applications using a process that is similar to the one used to locate WinMain:

- Review functions after the entry point.
- Pay close attention to functions after the command line is retrieved.
- Inspect the code to determine if it is likely library code or user generated.

```
main( int argc, char *argv[ ], char *envp[ ] )
```

- *argc* contains the number of arguments.
- *argv* is an array of command line arguments.
- *envp* is an array of environment variables.

We can apply a similar process to identify the main function for C++ console applications. Specifically, we can begin at the entry point and look for unidentified functions executed after command line arguments are retrieved.

For this brief discussion we will use the Windows executable with SHA-256 hash A37A290863FE29B9812E819E4C5B047C44E7A7D7C40E33DA6F5662E1957862AB. You can find this executable at Malware\Section1\console_example.zip within the course VM.

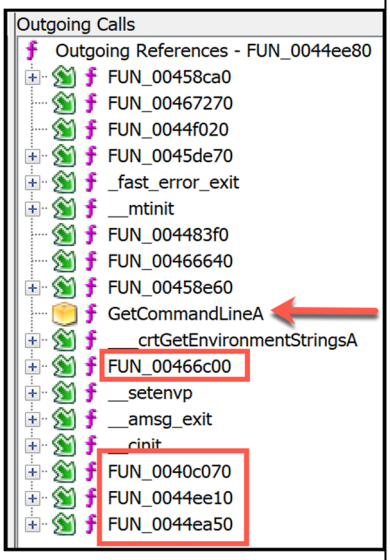
Standard arguments for a Microsoft Visual Studio-compiled main function include an integer that represents the number of arguments, an array of command line arguments, and an array of environment variables. More information about the main function is available at https://for710.com/main.

Lab 1.1: Background Topics: Identifying Main (2)

Reviewing functions after the command line is retrieved reveals multiple candidates for review.

```
                        entry
0044f0e0    PUSH        EBP
0044f0e1    MOV         EBP, ESP
0044f0e3    CALL        ___security_init_cookie
0044f0e8    CALL        FUN_0044ee80
0044f0ed    POP         EBP
0044f0ee    RET
```

```
                        FUN_0044ee80
0044ee80    PUSH        EBP
0044ee81    MOV         EBP, ESP
```

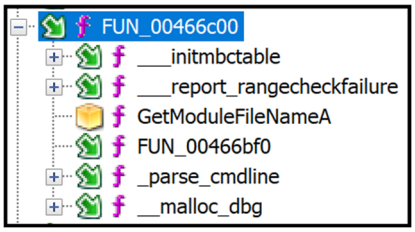


At the entry point, we see one call to a non-library function FUN_0044ee80. Jumping to that location and viewing the function call tree reveals many functions with the generic "FUN" label. These are all main function candidates. However, since both `WinMain` and main require the command line arguments as a parameter, we will focus on the functions *after* the reference to `GetCommandLineA`. This leaves four functions:

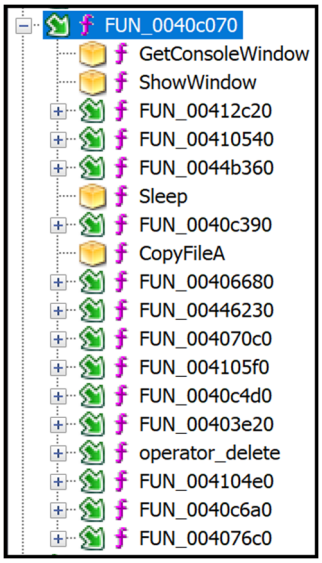
1. FUN_00466c00
2. FUN_0040c070
3. FUN_0044ee10
4. FUN_0044ea50

Lab 1.1: Background Topics: Identifying Main (3)

- FUN_00466c00 contains largely reserved functions.



- FUN_0040c070, however, contains Windows API references and numerous non-library functions.



Looking at the function call tree of the first function reveals calls to many reserved functions. This is likely not user-defined code.

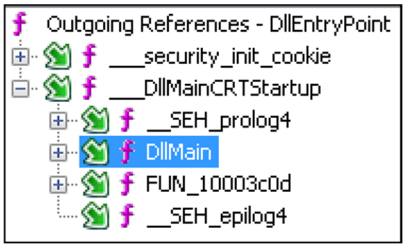
The call tree for FUN_0040c070 shows no reserved functions and multiple Windows API references. This is worthy of additional investigation.

Lab 1.1: Background Topics: DLL Entry Point Terminology

- **DllEntryPoint:**
  - The address specified in the AddressOfEntryPoint field in the optional header.
  - It is called when the DLL is loaded and unloaded.
  - *fdwReason* specifies the reason for the call:

    DLL_PROCESS_ATTACH, DLL_PROCESS_DETACH,

    DLL_THREAD_ATTACH, DLL_THREAD_DETACH



- **DllMainCRTStartup:** VCRuntime entry point that calls DllMain.
- **DllMain:** Optional user or library supplied entry point with the same prototype as DllEntryPoint.

```
BOOL WINAPI DllMain(
    HINSTANCE hinstDLL, // handle to DLL module
    DWORD fdwReason,    // reason for calling function
    LPVOID lpReserved ) // reserved
```


When a DLL is loaded or unloaded, certain initialization and cleanup activities may occur. The entry point of a DLL directs the system to the code that performs these setup and teardown activities. An entry point function is optional for DLLs and not explicitly exported.

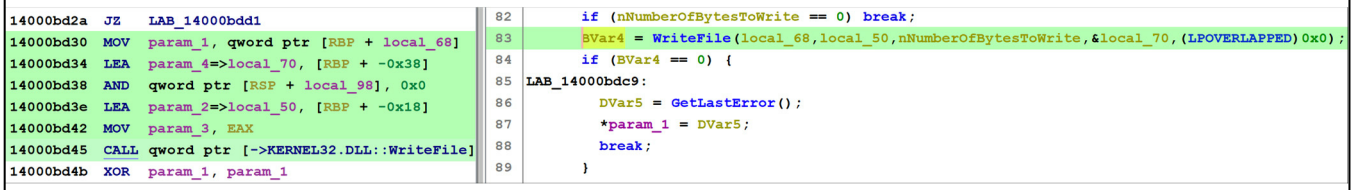
The `DllEntryPoint` is the address specified by the `AddressOfEntryPoint` field in the optional header. It takes an argument `fdwReason` that specifies whether the function is called due to loading or unloading. An entry-point function is optional for DLLs.

DllMain is a similar concept to the `Dllentrypoint` (they have the same prototype), but it is defined by the user. If DllMain exists, it is called from the `DllEntryPoint`. The `fdwReason` argument is passed through to DllMain, and it is usually DllMain that acts on the reason. If a DLL is built using Visual Studio, the Visual C++ run-time library (`VCRuntime`) provides an entry point called `DllMainCRTStartup` to help set up the runtime environment. In this case, `DllEntryPoint` will call `DllMainCRTStartup`, which will in turn call `DllMain`. The graphic on this slide demonstrates this function call tree. Note that both `DllEntryPoint` and `DllMain` were renamed to those names for clarity—Ghidra does not automatically rename these functions. The malware sample used for the screenshot has SHA-256 hash 4279ec72f96a2ff976962de421c621791248c6916c27eec42952945a4adaf995.

For more information on DllMain see:
https://for710.com/dllmain
https://for710.com/run-time-library-behavior
https://for710.com/dll-ep
https://for710.com/entrypoint

Lab 1.1: Background Topics: Ghidra's Decompiler Output (1)

- It associates the disassembly with a high-level C representation.
- We will learn to use the decompiler output to support and expedite our code analysis; it does not supersede analysis of the disassembly.
- Click and drag to highlight code and compare disassembly with C representation, and vice versa.



During the upcoming lab, we will make use of Ghidra's decompiler output to accelerate our analysis. However, this output is considered supplemental and not our sole source (pun intended) of information to understand the program at a code level.

Ghidra attempts to keep the disassembly and decompiler output in sync, meaning that each view should update appropriately when the cursor is placed at a new location. To make the relationship between one view and the other even clearer, analysts can click and drag to highlight code in one view, and the corresponding code in the other view should highlight. The screenshot on this slide shows an example of this highlighting feature.

Lab 1.1: Background Topics: Ghidra's Decompiler Output (2)

- Within your Ghidra configuration, printing of type casts is initially disabled to allow for a cleaner reading experience.
- We will enable this option later to improve our understanding of the code.

```
if ((*(byte *)((longlong)iVar5 + (longlong)_DstBuf) & 1) != 0) {
  uVar7 = uVar7 | 1 << (uVar8 & 0x1f);
```

Type casts enabled

```
if ((*(iVar5 + _DstBuf) & 1) != 0) {
  uVar7 = uVar7 | 1 << (uVar8 & 0x1f);
```

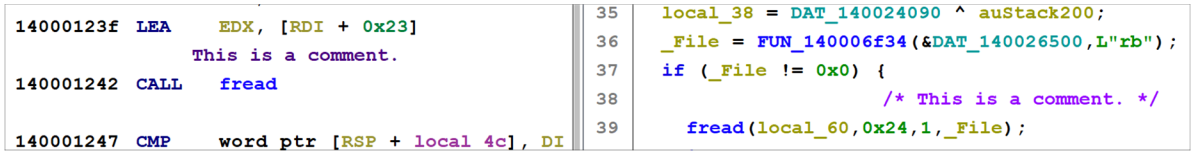
Type casts disabled

The asterisk (*) indicates a dereferenced pointer.

By default, Ghidra shows type casting information in the decompiler view. This refers to additional information about the type of data referenced. For example, (int) refers to an integer and (byte *) refers to a byte of data located at an address specified by a pointer. Type casting provides helpful detail but also adds clutter to the decompiler output, so it was disabled for this class. However, depending upon your prior experience and comfort level, you may find it helpful to enable. To change this configuration setting, browse to Edit > Tool Options, Decompiler > Display, and check/uncheck "Disable printing of type casts".

Lab 1.1: Background Topics: Ghidra's Decompiler Output (3)

- Ghidra includes comment types: EOL, Pre, Post, Plate, and Repeatable.
- Comments can be displayed in both the Listing and Decompile windows, and the specific comment types shown are configurable.
- We will use the default approach—EOL comments in the disassembly window, and Pre comments in the C code.
- EOL comments inserted in the disassembly only appear there, but Pre comments in the decompiler output also appear in the Listing view.



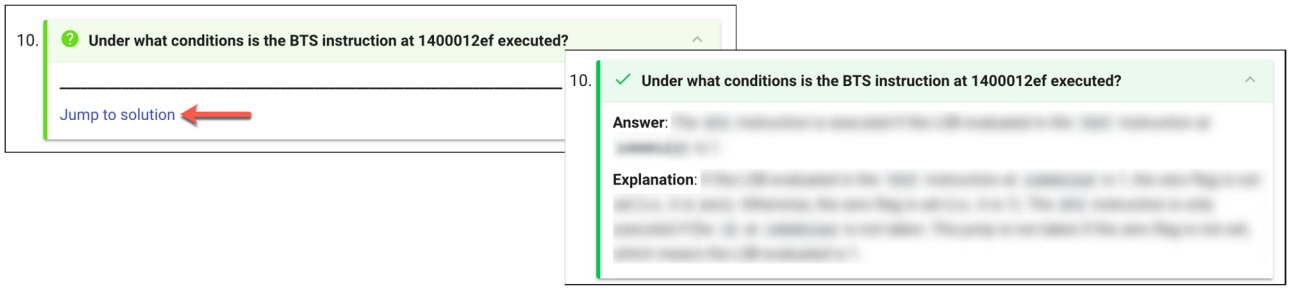
Writing comments during code analysis is a great way to document your work and share information with others. As you proceed through the upcoming labs, you should comment code frequently. To make a comment in the Decompile window, press the semicolon key and type your comment in the "Pre Comment" window (the default comment type for the Decompiler output). Then, hit **OK**. The comment will appear immediately above the line you described in both the Decompiler and Listing (i.e., disassembler) view.

Note that EOL comments inserted in the Listing view will *not* appear in the Decompile view. However, Pre comments inserted in the decompiler output will appear in *both* the Decompile and Listing views.

You can view further config options for comments under Edit > Tool Options > Listing Fields (each comment type has a section) and Edit > Tool Options > Decompiler > Display (each comment type has options to display or not).

Answer a Question and View the Solution for Immediate Feedback

- Students are encouraged to view the solution after answering a question.
- This provides immediate feedback and helps adjust your analysis approach as needed for upcoming questions.
- Click "Jump to solution" if viewing the workbook within the browser (recommended); otherwise, browse past all questions to see the solutions.

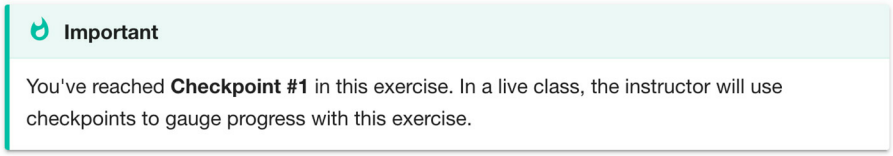


The upcoming lab has many questions. To ensure you optimize time dedicated to the lab, you are encouraged to check your answer to a question soon after considering your response. This will help correct your analysis approach as needed and reinforce key concepts as you proceed through the lab.

If you are viewing questions in a browser (recommended), simply click on "Jump to solution" after each question to arrive at the detailed answer. If you are using the PDF or hard copy, you can find all solutions in the "Lab Solutions" section, which is located after all questions for this lab.

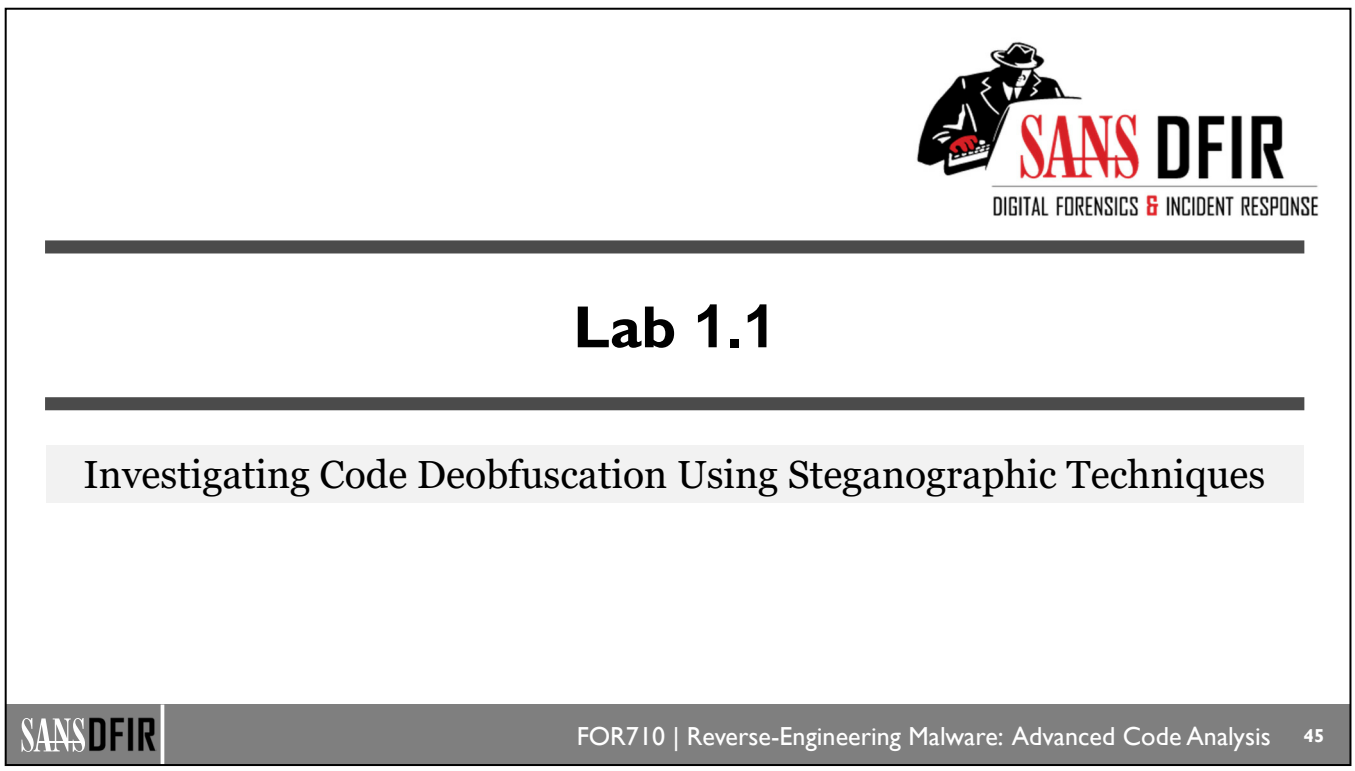
Lab Checkpoints Measure Progress and Maintain Momentum

- Longer labs like the one coming up have checkpoints in the workbook:



- Checkpoints help the instructor understand how students are progressing with a lab.
- The instructor will periodically discuss the steps leading up to a checkpoint and answer any questions.
- If you are confident with your progress, feel free to continue working through the lab while the instructor reviews the material.

When working through a long lab in the workbook, students will encounter checkpoints. In a live class, the instructor will use checkpoints to determine how students are progressing with a lab. The instructor will also periodically explain the steps leading up to a checkpoint to emphasize key concepts and ensure students are maintaining some momentum as they work through questions. If you're comfortable with your progress and answers, feel free to continue working as the instructor reviews the material.



Please begin Lab 1.1 now.

Analyzing Code Deobfuscation: Module Objectives, Revisited

- ✓Understand code used to deobfuscate executable content.
- ✓Be able to communicate the details of how code is obfuscated.
- ✓Recognize key Windows APIs used to allocate memory.
- ✓Differentiate user-defined code from library code.
- ✓Dump deobfuscated executable content to disk.
- ✓Explain how multiple files work together to execute malicious code.
- ✓Develop comfort with non-binary formats during malware analysis.

In this module, we analyzed malware that deobfuscates additional code during execution. Specifically, we reviewed a sample that employed steganography techniques to decode executable content from a traditionally benign file format. Advanced-level malware reverse engineers must be prepared to communicate the details of code deobfuscation, and Lab 1.1 provided an opportunity to perform this work. As part of this analysis effort, we explored the use of various Windows APIs and C functions responsible for memory allocation and learned how to differentiate user-defined code from library code within Ghidra.

Understanding *how* data is obfuscated can be tedious and time consuming, but it helps better characterize the attacker, build signatures, and develop tools to automate deobfuscation.

Identifying Program Execution

In the last module, we discussed an approach to code deobfuscation. This module discusses the next logical step in running malware—executing the deobfuscated code.

Malware That Debofuscates Code Then Executes the Content

- When a program is executed from disk, the Windows loader parses the Portable Executable (PE) file and prepares to launch it.
- If malware plans to run an in-memory EXE, it must perform the heavy-lifting to load the next stage content—this is "reflective" loading.
- During analysis of malware that deobfuscates and launches a program, we will encounter code that contributes to execution.
- We need to understand the structure of a Portable Executable (PE) file and the steps involved in loading and running a program.

In the last lab, we analyzed steg techniques used to deobfuscate a hidden, embedded executable. After a program is deobfuscated, the next logical step is code execution.

Typically, when a program is executed on Windows, the Windows "loader" manages the process of loading the executable into memory, resolving its dependencies, and performing other activities to prepare for execution. However, when malware decodes and executes a next stage binary in memory only (i.e., the executable content does not reside on disk), the Windows loader is not involved, and the initial malware code must prepare for execution. This is necessary when the malware decodes an EXE or DLL from within the primary executable (the scenario we evaluated in Lab 1.1), or when the initial code downloads the next stage from a server for execution in memory. The malicious code must include its own loader, and this is referred to as "reflective" loading. During code analysis, you will encounter code that prepares for and performs execution, and you must be able to identify its role in that process.

Before we can identify code that assists with malicious program execution, we need to better understand the normal steps involved in launching an executable from disk. This includes details of the Portable Executable file format and the various header fields that describe the organization and contents of the executable.

Note that this module is focused on analyzing the execution of an in-memory EXE, not shellcode. We will cover shellcode execution in the next module.

Analyzing Program Execution: Module Objectives

- Understand the key components of a Windows Executable header.
- Identify the structures and fields associated with a program's imports.
- Identify the structures and fields associated with a program's exports.
- Understand the steps necessary to prepare a program for execution.
- Recognize code that maps an executable into memory.
- Determine the code execution entry point for a second-stage binary.

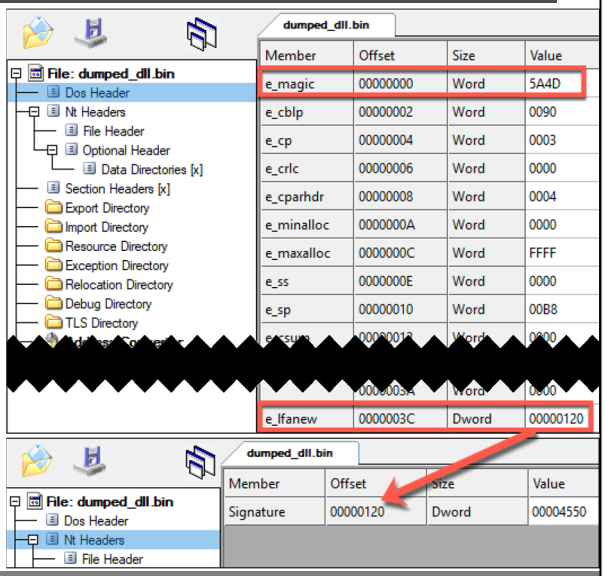
Code that supports program execution is not inherently malicious, but we must recognize it so we can comfortably examine the rest of the program.

This slide describes the objectives of this module.

It is important to remember that program execution activities are not intrinsically malicious, just as a function's prologue and epilogue code are not malicious. However, in both cases, we must learn to recognize this code so we can safely and comfortably shift our attention to other aspects of the program. Once you learn to identify code that supports program execution, you will not need to perform in-depth analyses each time you encounter this code.

PE File Headers Describe a Program's Structure and Content

- The MS-DOS header begins with "MZ".
- Most fields in the DOS header are not relevant to newer operating systems.
- The `e_lfanew` field specifies the offset of the PE header ("Nt Headers" in CFF).
- The PE header begins with the signature 4-byte signature "PE\0\0".
- The PE header consists of the PE signature, COFF file header, and optional header.



Just as a surgeon should understand the human body and its parts to excel in surgery, a malware reverse engineer should understand the structure and components of a binary to be proficient in malware analysis. Within the Windows operating system, we are referring to the Portable Executable (PE) format.

We begin our travels through the PE file format at the start of a Windows executable. In this slide, we load the dumped DLL from our first lab into CFF Explorer. You can find this file in the Malware directory within Section1\dumped_dll.zip. On the left side, we see headers that comprise the first bytes of a Windows executable. These headers describe the rest of the file, including the executable content, resources, and imports.

Let's start with the MS-DOS header (also called the MS-DOS Stub), which displays "This program cannot be run in DOS mode" when the executable is run in MS-DOS. At the beginning of this header (see top-right of the figure on this slide) is the e_magic field, and it contains the well-known "MZ" characters represented by the hexadecimal value 0x4D5A (shown as 0x5A4D above because the value is interpreted as little-endian). Most fields in this header are not relevant to newer operating systems, but the final `field e_lfanew` (see below) is significant because it contains the offset of the PE header, shown in CFF Explorer as Nt Headers.

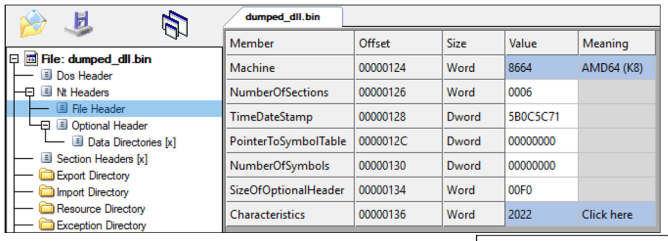
Clicking on Nt Headers on the left takes us to file offset 0x120, which matches the value of the `e_lfanew` field. The value translates to the string "PE\0\0", which appears at the beginning of the PE header (similar to the "MZ" characters, shown as 0x00004550 because the value is interpreted as little-endian). The PE header consists of the PE signature, COFF file header, and optional header.

For more detail on Microsoft's PE Format, see:
https://for710.com/pe-format
https://for710.com/pe-tour
https://for710.com/winnt

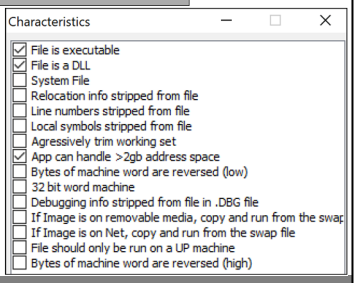
The COFF File Header Includes Key Details about the Binary

- The file header is a structure of type IMAGE_FILE_HEADER.

```
C++

typedef struct _IMAGE_FILE_HEADER {
  WORD  Machine;
  WORD  NumberOfSections;
  DWORD TimeDateStamp;
  DWORD PointerToSymbolTable;
  DWORD NumberOfSymbols;
  WORD  SizeOfOptionalHeader;
  WORD  Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```



- The Characteristics member specifies, for example, if the binary is executable and if it is a DLL.
- Malware may check these fields to determine the number of sections, file type, and other characteristics in preparation for execution.



Next on our path is the COFF File Header, displayed simply as File Header in CFF Explorer. This header is of type IMAGE_FILE_HEADER (for more detail on this structure, see https://for710.com/imagefileheader). This header includes information such as the target machine type (e.g., x64), the number of sections, the compile timestamp (seconds since January 1, 1970, UTC), and file characteristics (e.g., is the executable a DLL or EXE?).

Malware often checks these fields after unpacking or decoding executable content to determine the architecture and executable type of the next stage of execution.

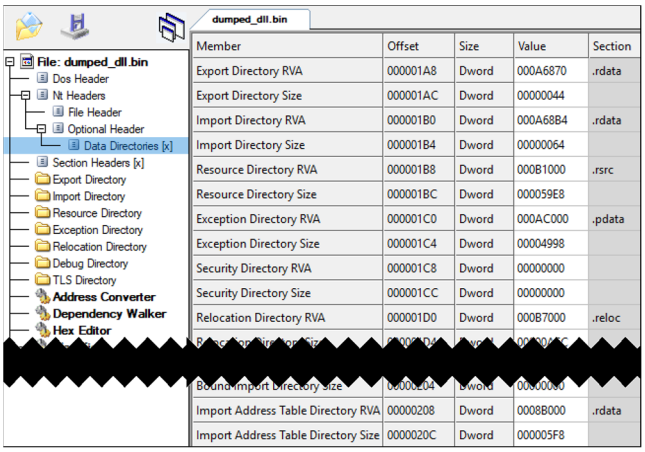
Visit https://for710.com/fileheader for more information on the COFF File Header.

Data Directories Point to Various Tables, Including Imports

- Each data directory is structure of type IMAGE_DATA_DIRECTORY.

```
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD   VirtualAddress;
    DWORD   Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

- The VirtualAddress field specifies the RVA of the table.
- During code analysis, we may encounter SUB and CMP operations that check the directory size.
- A zero-directory size indicates no corresponding information.



At the end of the Optional Header is Data Directories, which points to tables that contain supporting information, including imported and exported functions. Each data directory entry is a structure of type `IMAGE_DATA_DIRECTORY`. This structure has two fields:

- VirtualAddress: The Relative Virtual Address (RVA) of the specified table. It is "virtual" because this is an address after the executable is loaded into memory. It is relative to the ImageBase, so adding the RVA to the Imagebase provides the Virtual Address (VA) in memory of the specified table.
- Size: The size, in bytes, of the table.

During code analysis of content that loads additional executable content, we'll often encounter SUB and CMP instructions that evaluate the size a particular directory. For example, the loading code may evaluate if the Import Directory has a nonzero size to determine if it needs to load dependencies. The loading code may also check the size of the Export Directory to determine if the next-stage executable has any exports.

For more information on data directories, see:
https://for710.com/data-directories
https://for710.com/imagedatadirectory

The Section Table Provides Details on Each Upcoming Section

Each section header is a structure of type IMAGE_SECTION_HEADER.

```
typedef struct _IMAGE_SECTION_HEADER {
  BYTE  Name[IMAGE_SIZEOF_SHORT_NAME];
  union {
    DWORD PhysicalAddress;
    DWORD VirtualSize;
  } Misc;
  DWORD VirtualAddress;
  DWORD SizeOfRawData;
  DWORD PointerToRawData;
  DWORD PointerToRelocations;
  DWORD PointerToLinenumbers;
  WORD  NumberOfRelocations;
  WORD  NumberOfLinenumbers;
  DWORD Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

dumped_dll.bin

Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations...	Linenum...	Characteristics
00000228	00000230	00000234	00000238	0000023C	00000240	00000244	00000248	0000024A	0000024C
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	000897B0	00001000	00089800	00000400	00000000	00000000	0000	0000	60000020
.rdata	0001CC6E	0008B000	0001CE00	00089C00	00000000	00000000	0000	0000	40000040
.data	00003CE4	000A8000	00001800	000A6A00	00000000	00000000	0000	0000	C0000040
.pdata	00004998	000AC000	00004A00	000A8200	00000000	00000000	0000	0000	40000040
.rsrc	000059E8	000B1000	00005A00	000ACC00	00000000	00000000	0000	0000	40000040
.reloc	00000AEC	000B7000	00000C00	000B2600	00000000	00000000	0000	0000	42000040

```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17  Decoded text
00000210   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ........................
00000228   2E 74 65 78 74 00 00 00 B0 97 08 00 00 10 00 00 00 98 08 00 00 04 00 00  .text...°—.....˜.....
00000240   00 00 00 00 00 00 00 00 00 00 00 00 20 00 00 60 2E 72 64 61 74 61 00 00  ............ ..`.rdata..
00000258   6E CC 01 00 00 B0 08 00 00 CE 01 00 00 9C 08 00 00 00 00 00 00 00 00 00  nÌ...°...Î...œ..........
00000270   00 00 00 00 40 00 00 40 2E 64 61 74 61 00 00 00 E4 3C 00 00 00 80 0A 00  ....@..@.data...ä<...€..
00000288   00 18 00 00 00 6A 0A 00 00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 C0  .....j..............@..À
000002A0   2E 70 64 61 74 61 00 00 98 49 00 00 00 C0 0A 00 00 4A 00 00 00 82 0A 00  .pdata..˜I...À...J...‚..
000002B8   00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 40 2E 72 73 72 63 00 00 00  ............@..@.rsrc...
000002D0   E8 59 00 00 00 10 0B 00 00 5A 00 00 00 CC 0A 00 00 00 00 00 00 00 00 00  èY.......Z...Ì..........
000002E8   00 00 00 00 40 00 00 40 2E 72 65 6C 6F 63 00 00 EC 0A 00 00 00 70 0B 00  ....@..@.reloc..ì....p..
```


The number of entries in the section table is specified in the NumberOfSections field in the file header. The section table is comprised of section headers, and each header is a structure of type `IMAGE_SECTION_HEADER`.

Each section header provides important information about the name, location (both on disk and in memory), and characteristics of each section. Key sections include ".text" for executable code, ".rdata" for read-only data, and ".rsrc" for resources like icons (and potentially additional executable content).

Within the section headers, there are several important fields to note:

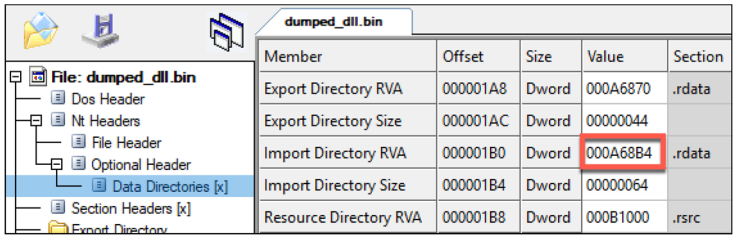
- Virtual Size: References to *virtual* sizes and addresses refer to values after the executable is loaded into memory. The virtual size refers to the size of a section's content when it is loaded into memory.
- Virtual Address: The RVA of the section in memory. This value is added to the image base to identify the VA of the section in memory.
- Raw Size: References to *raw* sizes and addresses refer to values relevant to the image file on disk. This raw size refers to the size of a section on disk, including any padding necessary to meet file alignment requirements.

Note that while the raw size includes padding to meet file alignment requirements, a section's virtual size does not include any padding. This means the raw and virtual sizes of a section may be different even when the content (not including padding) is identical. Also, if the virtual size for a section is larger than the raw size, the section will be padded with additional zeroes so it can accommodate the virtual size.

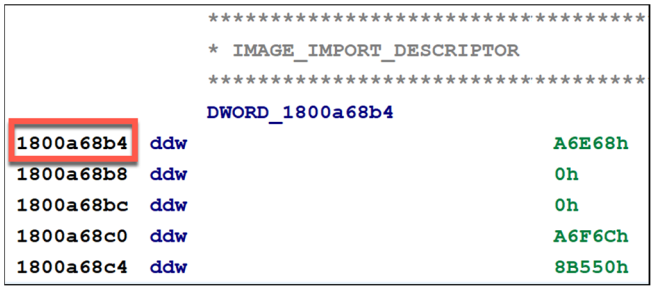
For more information on the section table, see:
https://for710.com/section-table
https://for710.com/imagesectionheader

To Locate Imports, Begin with the Import Directory RVA

- Add the RVA to the image base to calculate the VA of the import directory table.



- The table has structures of type IMAGE_IMPORT_DESCRIPTOR for each imported DLL.



With a basic understanding of the PE file format and header information, we can navigate a PE file to locate important content like the import table.

First, we return to the data directories within the optional header to identify the RVA of the import directory. In the case of our dumped DLL, it is 000A68B4. Let's go to this location within the DLL. However, looking at this offset within the file on disk will not be helpful since, as mentioned earlier, the RVA is an address in memory. We will need to use Ghidra since it loads our executable into memory like how the Windows loader would in preparation for execution.

Ghidra loads the dumped DLL at the preferred image base, 180000000. Adding this value to the import directory RVA equals 1800A68B4. We can jump to this location to arrive at the beginning of the import directory table, which contains all the references we need to understand the program's imports. There is one `IMAGE_IMPORT_DESCRIPTOR` structure for each imported DLL.

For more information on the import directory table, see:
https://for710.com/import-directory-table
https://for710.com/pe-tourimports

Each IMAGE_IMPORT_DESCRIPTOR Has Five Fields

1. **Import Lookup Table RVA:** The table includes the name or ordinal for each imported function within the DLL.
2. **Time Stamp:** This field is usually zero.
3. **Forwarder Chain:** This field is usually zero.
4. **DLL Name RVA:** This is a string that specifies the imported DLL.
5. **Imported Address Table (IAT) RVA:**
   - The IAT initially mirrors the Import Lookup Table.
   - At load time, it is overwritten with the addresses of external functions

Load DLL (DLL Name) → Find Function Addresses (via IAT/ILT Entries) → Overwrite IAT Entries

Each `IMAGE_IMPORT_DESCRIPTOR` entry consists of the following elements, described below.

Import Lookup Table RVA: The relative virtual address of the Import Lookup Table (ILT), which includes a name or ordinal for each imported function within the DLL. This table is also referred to as the Import Name Table (INT).

Time Stamp: This value is usually zero. If this value is nonzero, it means the DLL is bound. When an executable is bound, the binary on disk has the in-memory addresses of imported DLLs. In other words, functions do not need to be resolved during the loading process. You are unlikely to encounter this scenario during malware analysis, so we will not discuss binding in detail. For more information, browse to https://for710.com/inside-windows-part2 and view the section titled "Binding."

Forwarder Chain: A DLL may send references to its functions to another DLL. However, like the Time Stamp field above, this value is generally zero and does not warrant further discussion. For more information, see https://for710.com/pe-tourimports.

DLL Name RVA: The RVA of the string that specifies the imported DLL.

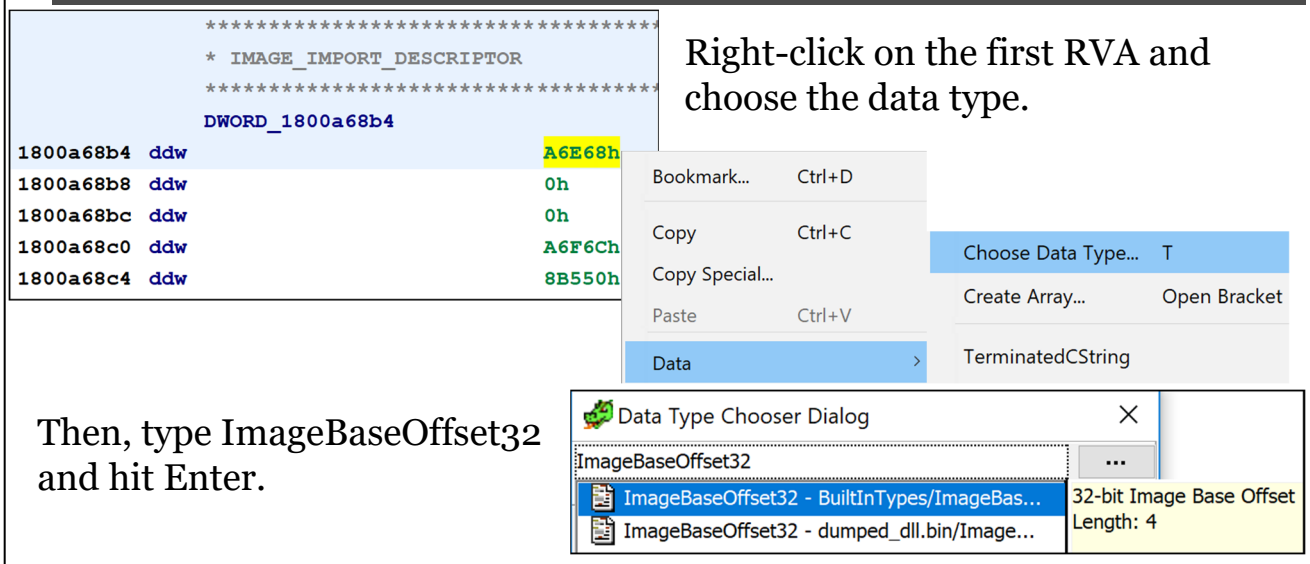
Import Address Table (IAT) RVA: The relative virtual address of the Import Address Table. The Import Address Table is populated by the loader when the executable and its imported DLLs are mapped into memory, and it is a table of pointers to the imported functions. Each entry in the table is called a "thunk" and the table is referred to as a "thunk table".

Keeping this terminology in mind, an external function can be called because each IMAGE_IMPORT_DESCRIPTOR structure is processed as follows:

1. Load the specified DLL into memory.
2. Process each entry in the IAT table (which mirrors the Import Lookup Table) to find the address of a desired function in the loaded external DLL.
3. Overwrite each IAT entry with the address of an external function.

Once these steps are completed for all imported DLLs and functions, the program can call external functions by referencing addresses in the IAT.

Help Ghidra Interpret the Three RVAs Properly



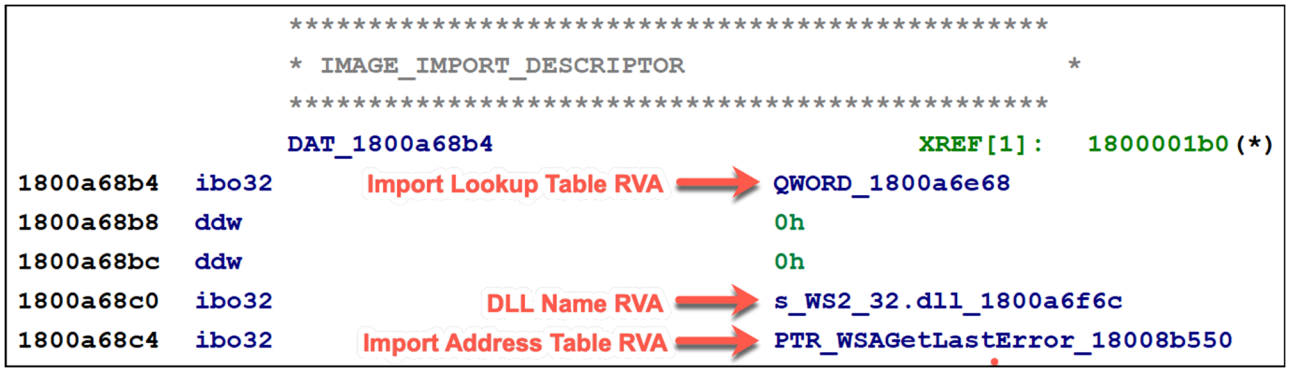
The three RVA fields within the `IMAGE_IMPORT_DESCRIPTOR` are not recognized as 4-byte RVAs by default. To help Ghidra interpret these bytes correct, right-click the first RVA value and go to Data > Choose Data Type…

Then, type `ImageBaseOffset32` and hit Enter on the keyboard (you could also choose one of the options that appear while typing, as shown on this slide).

After choosing the data type and hitting Enter (or clicking OK), you will notice the data does not look any different. See the next slide for the final step.

Select the "Last Used" Data Type for All Three RVAs

For all three RVAs, right-click and choose **Data > Last Used: ImageBaseOffset32**.



Oddly, Ghidra does not immediately represent bytes differently when you choose `ImageBaseOffset32` for the first time. However, if you now right-click on each RVA and choose Data > Last Used: `ImageBaseOffset32`, the field will be shown correctly with the data type "ibo32."

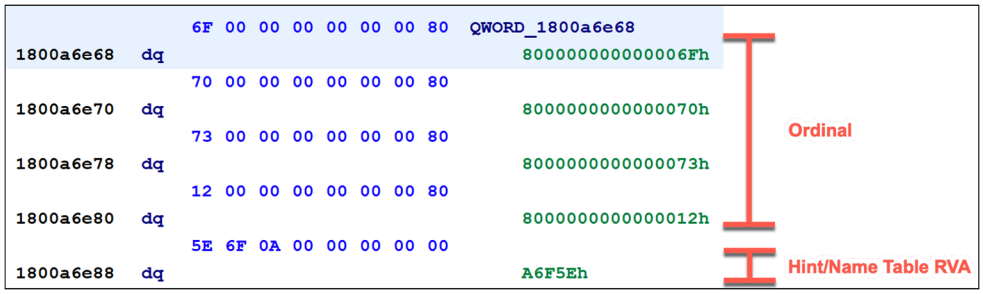
The slide shows the appropriate data types and highlights the RVA fields in the IMAGE_IMPORT_DESCRIPTOR structure.

The DLL Name RVA clearly points to a string that specifies the imported DLL. The IMAGE_IMPORT_DESCRIPTOR structure on this slide refers to ws2_32.dll.

The other two RVAs are less straightforward and require additional explanation. We will discuss these two RVAs in the upcoming slides.

There Is One Import Lookup Table (ILT) Per Imported DLL

- The Import Lookup Table (ILT) has a value for each imported function.
- If the MSB is set, the value refers to a function imported by ordinal.
- If the MSB is *not* set, the value is an RVA to the Hint/Name Table.



- The ILT includes 32 or 64-bit values based on the file's architecture.

The first RVA within the `IMAGE_IMPORT_DESCRIPTOR` structure points to the Import Lookup Table (ILT). This table is an array of 32 or 64-bit values (depending upon the executable's target architecture) that are a structure of type `IMAGE_THUNK_DATA`, where each value corresponds to an imported function from the DLL specified by the DLL Name RVA. Since the decoded DLL we've been discussing is 64-bit, values in this binary's import lookup table are 64-bit.

If the highest bit of a value is set (i.e., it is 1), the value corresponds to a function imported by ordinal. In this case, the least significant bits indicate the ordinal number. The ordinal value is used to locate the exported function in the relevant DLL.

If the highest bit is *not* set (i.e., it is zero), the least significant bits are the RVA to an entry in Hint/Name table, which provides additional detail on the function imported by name.

Double-clicking on the Import Lookup RVA shown on the previous slide brought us to content on this slide. The image shows the first five entries of the import lookup table for functions imported from ws2_32.dll. The first four entries have their highest bit set, which means these values refer to functions imported by ordinal. The ordinal values are represented by the least significant bits. For example, the first ordinal value is 6F, or decimal 111. The fifth value shown on the slide does not have its highest bit set, so it is an RVA to an entry in the Hint/Name table (discussed on the next slide). We can change this value to data of type ImageBaseOffset32 or ImageBaseOffset64 depending upon the executable's architecture. Since the dumped DLL used in these slides is 64-bit, we would convert the RVA to data of type ImageBaseOffset64. This allows us to easily double-click the value to jump to the appropriate virtual address.

For additional detail on the import lookup table, see https://for710.com/import-lookup-table.

There Is One Hint/Name Table for All Imported Functions

- The Hint/Name table helps locate functions imported by name.
- The table includes structures of type IMAGE_IMPORT_BY_NAME.
- Each entry has three components:
  1. Hint: Index into imported DLL
  2. Name: Function name
  3. Padding: 1 or 0 bytes

```
                        ************************************************
                        * IMAGE_IMPORT_BY_NAME                         *
                        ************************************************
                        3A 00
1800a6f52  dw                                   3Ah
                        57 53 41 49 6F 63 74 6C 00
1800a6f54  ds                                   "WSAIoctl"
                        00
1800a6f5d  ??                                   00h

                        ************************************************
                        * IMAGE_IMPORT_BY_NAME                         *
                        ************************************************
                        4A 00                                   WORD_1800a6f5e
1800a6f5e  dw                                   4Ah
                        57 53 41 52 65 63 76 46 72 6F 6D 00
1800a6f60  ds                                   "WSARecvFrom"
```


One Hint/Name table covers all imported functions for the file. Each entry in the table has three components:

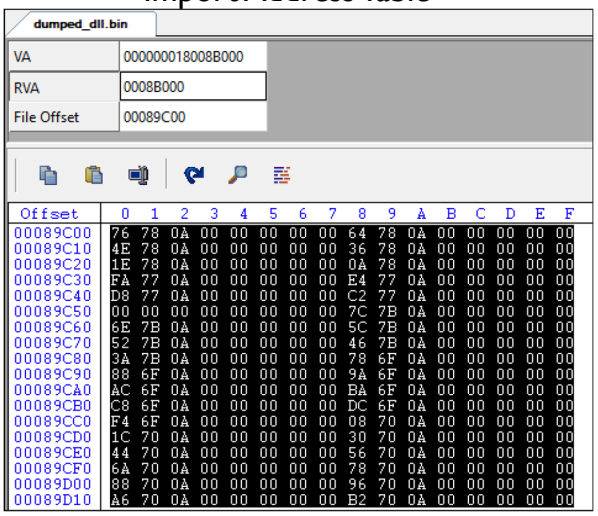
1. Hint: This is an index into the imported DLL, and it is used to help locate the required function's ordinal. Specifically, this value is an index into the export name pointer table, which we will discuss later.
2. Name: The name of the imported function, null terminated. This is used to find the imported function within a DLL when using the Hint does not suffice.
3. Padding: You may see additional zero values to ensure each entry is on an even boundary.

When imports are resolved, an attempt is made to locate the function based on the specified Hint. If this effort is unsuccessful, the import is found by name.

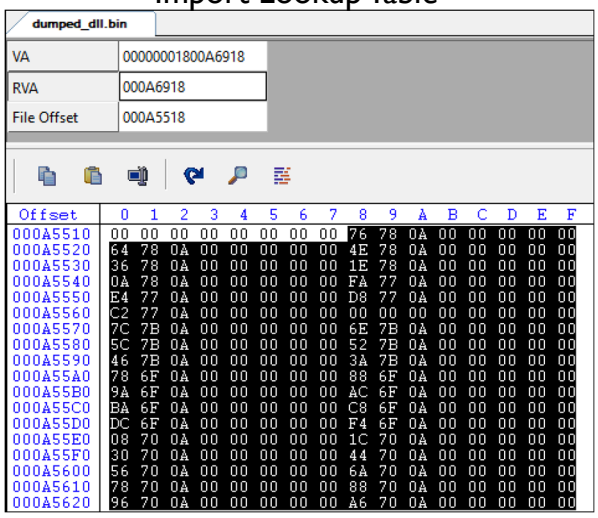
The highlighted address on this slide matches the Hint/Name Table RVA in the previous slide. This excerpt of the Hint/Name table shows two entries. Notice that the first has one byte of padding, while the second does not.

On Disk, the IAT and Import Lookup Table Match

Import Address Table



Import Lookup Table



On disk, the import address table and the import lookup table have the same structure and content.

On the left is the import address table shown using the address converter feature within CFF Explorer. This feature allows the user to provide the VA, RVA, or File Offset, and it translates that address to the other two addresses and shows the corresponding content in the hex dump output. According to the Data Directories section of the PE file, Import Address Table Directory RVA is 8B000, so that value was typed into the RVA field.

On the right, we have the the first import lookup table. To determine the starting address of the first import lookup table, you can locate the ws2_32.dll Import Lookup Table within Ghidra (this is the same one we reviewed earlier) and scroll up to find the first ILT. You will find it begins at 1800a6918 within Ghidra, so this value was typed into the VA field.

As you can see from the excerpts of both the import address and import lookup tables, the content is identical.

The import lookup table is found in most executables, but it is not required for a program to load properly. It is only required if the executable is bound, a rare case that we will not discuss in detail. For more information on binding, browse to https://for710.com/inside-windows-part2 and view the section titled “Binding.”

In Memory, the IAT Is Overwritten with Imported Function VAs

Import Address Table

```
                        **********************************************************
                        *            POINTER to EXTERNAL FUNCTION                *
                        **********************************************************
                        BOOL __fastcall CryptAcquireContextA(HCRYPTPROV * phProv, LPCSTR sz
                        DWORD           Stack[0x28]...dwFlags
                        192  CryptAcquireContextA  <<not bound>>
                        PTR_CryptAcquireContextA_18008b000        XREF[3]:   180000208(*),
                                                                             FUN_18003f810
18008b000     76 78 0a 00 00 00 00 00 addr     ADVAPI32.DLL::CryptAcquireContextA
```

Import Lookup Table

```
                    76 78 0A 00 00 00 00 00
1800a6918     dq                                        A7876h
                    64 78 0A 00 00 00 00 00
1800a6920     dq                                        A7864h
                    4E 78 0A 00 00 00 00 00
1800a6928     dq                                        A784Eh
                    36 78 0A 00 00 00 00 00
1800a6930     dq                                        A7836h
                    1E 78 0A 00 00 00 00 00
```

Dump 1 | Dump 3 | Dump 2 | Dump 4 | Dump 5 | Watch 1

Address	Hex
0000000018008B000	60 71 24 F1 FC 7F 00 00 50 7A 24 F1
0000000018008B010	90 76 24 F1 FC 7F 00 00 A0 F9 23 F1
0000000018008B020	D0 79 24 F1 FC 7F 00 00 50 6B 24 F1

A debugger confirms that external function addresses are written to the IAT.

```
00007FFCF1247160 jmp qword ptr ds:[7FFCF12DA110] CryptAcquireContextA
00007FFCF1247167 int3
00007FFCF1247168 int3
```

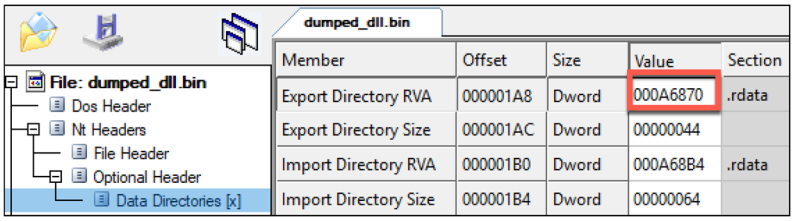
Address	Size	Info
00007FFCF1230000	000000000000	advapi32.dll
00007FFCF1231000	000000000006	".text"
00007FFCF1299000	000000000003	".rdata"
00007FFCF12D0000	000000000000	".data"

When an executable is loaded into memory and processed in preparation for execution, the IAT content is overwritten with the virtual addresses of the functions that are imported. These virtual addresses are referenced when the corresponding imported function is called (an example of this is shown on the next slide).

As shown on the previous slide, the VA of the import address table is 18008b000. If we jump to this address within Ghidra for the decoded DLL (see image on top-left), we find that Ghidra now recognizes this location as a pointer to an external function. Although the bytes shown in Ghidra seem identical to those shown in the Import Lookup Table (see top-right image), this is actually not the case for a running process. If we load the DLL into a debugger and jump to 18008b000, we observe the virtual address of CryptAcquireContextA (little-endian). The screenshots on the bottom of this slide confirm that the pointer at 18008b000 is the address of CryptAcquireContextA located within advapi32.dll.

To Locate Exports, Begin with the Export Directory RVA

- Add the RVA and image base to calculate the Export Directory Table VA.



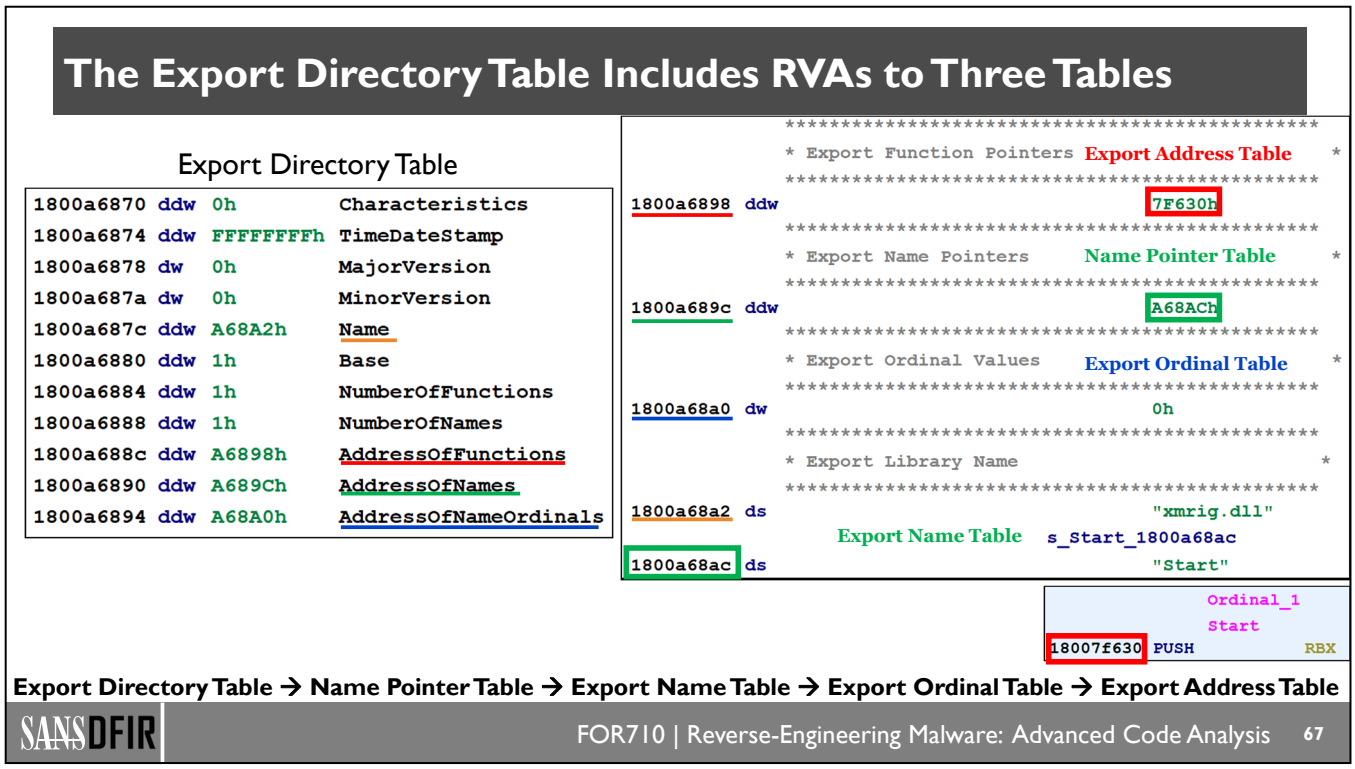
- The export directory table is a structure of type IMAGE_EXPORT_DIRECTORY, and it references four other tables.

```
                    IMAGE_DIRECTORY_ENTRY_EXPORT_1800a6870
1800a6870     IMAGE_DIRECTORY_ENTRY_EXPORT
   1800a6870 ddw     0h            Characteristics
   1800a6874 ddw     FFFFFFFFh     TimeDateStamp
   1800a6878 dw      0h            MajorVersion
```


We can navigate a PE file to locate exported functionality, similar to the process we followed for imported functions.

First, we return to the data directories within the optional header to identify the RVA of the Export Directory. In the case of our dumped DLL, it is 000A6870. Let's go to this location within the DLL using Ghidra. Ghidra loads the dumped DLL at the preferred image base, 180000000. Adding this value to the Import Directory RVA equals 1800A6870. We can jump to this location to arrive at the beginning of the export directory table, which is a structure of type `IMAGE_EXPORT_DIRECTORY`. The bottom image on this slide shows an excerpt of the Export Directory Table.

The export directory table contains pointers to several other tables. The code responsible for loading a program uses these tables to help locate functions exported by name or ordinal. We'll discuss this table in more detail in the upcoming slides.



The export directory table contains the following fields:

- Characteristics: This is a reserved value and is always zero.
- TimeDateStamp: This specifies when the export was created. In our example, this appears to be inaccurate since the value has all bits set.
- MajorVersion/MinorVerision: This is linker version information and often contains zero values even in legitimate DLLs.
- Name: RVA of DLL name. This is specified by the developer, and it may be different from the file name.
- Ordinal Base: This is the first ordinal number for exports in the DLL. This value is typically 1, and it is subtracted from a function's ordinal number to calculate the index into the export address table (discussed below). For example, the address of an exported function with ordinal number 1 is located at index 1 -1 = 0 (zero) within the export address table.
- NumberOfFunctions: This represents the number of exported functions.
- NumberOfNames: This represents the number of functions exported by name.
- AddressOfFunctions: RVA of the export address table, which contains RVAs of the exported functions. Ordinals (minus the ordinal base mentioned above) are used as an index into this table.
- AddressOfNames: RVA of the name pointer table, which contains RVAs into the export name table. The export name table contains the strings that other executables can use to import functions. Note that this table is not directly referenced by the export directory table.
- AddressOfNameOrdinals: RVA of the export ordinal table, which contains an array of indexes into the export address table. The values begin at zero. The export ordinal table and export name table work together; the export ordinal table provides the corresponding ordinal for a function exported by name. Items in each table are associated because they have the same index in their respective tables.

Though not shown on this slide, all RVA should manually be converted to data of type ImageBaseOffset32 as previously discussed. Then, the analyst can double-click on each RVA to jump to the specified address.

To locate a function imported by name, the Windows loader (or code responsible for loading an executable) first identifies the export directory table based on the Export Directory RVA in the Data Directories. Then, it will consult the name pointer table to locate the export name table. The name table is searched to find the appropriate function name. Once located, it will read the corresponding ordinal value (using the same index number) from the export ordinal table. The ordinal value is used as an index into the export address table to identify the address of the exported function. Based on this flow of events, it may be clear that importing by ordinal is slightly faster than importing by name. In fact, importing and exporting by function name exists only has a convenience for developers.

To export functions by ordinal, only the export directory table and export address table are required. The other three tables allow functions to be exported by name.

For additional information, see:
https://for710.com/export-data

When a Process Is Spawned, the Loader Prepares for Execution

Key activities include (not necessarily in this order):

- Confirm the file is a Windows executable
- Resolve critical APIs
- Map the executable into memory
- Load imported DLLs
- Resolve imported functions
- Apply relocations, if necessary
- Update section permissions, if necessary
- Identify the entry point (EP) for execution
- Execute code beginning at the EP

We discussed the details of a Windows executable header, including where information on imported and exported functions are stored. Now, we will put that knowledge into the context of program execution.

Executing a process is no simple task. Numerous initialization activities must take place to prepare an executable for actual execution. The operating system (i.e., the Windows loader) generally takes care of these tasks automatically when an executable is launched from disk. However, if an executable is unpacked or decoded in memory, it cannot rely on the OS to manage its startup automatically. Instead, the loading program must manually perform the work to prepare the second stage content for execution.

Key activities to load a program include confirming it is a Windows executable, loading the relevant file components into memory, resolving dependencies, identify the entry point for execution, and beginning execution. The upcoming slides discuss each step, in detail.

This slide lists the common steps involves in preparing a program for execution. Note that none of these activities are inherently malicious. Our goal is to identify code that is performing these initialization activities as quickly as possible during code analysis and move on. Also, the steps listed on this slide do not necessarily occur in this order.

The Loader Confirms the Binary Is a Valid Windows Executable

- The loading code evaluates key fields in the file's header.
- Common checks include "MZ", "PE", and the architecture.
- The evaluated bytes appear reversed because they are read as little-endian data.
- This is an example of code within a a Cobalt Strike loader.

```
10008a77    MOVZX       EDX, word ptr [ECX]
10008a7a    CMP         EDX, 0x5a4d
10008a80    JNZ         LAB_10008ab0
10008a82    MOV         EAX, dword ptr [EBP + local_c]
10008a85    MOV         ECX, dword ptr [EAX + 0x3c]
10008a88    MOV         dword ptr [EBP + local_8], ECX
10008a8b    CMP         dword ptr [EBP + local_8], 0x40
10008a8f    JC          LAB_10008ab0
10008a91    CMP         dword ptr [EBP + local_8], 0x400
10008a98    JNC         LAB_10008ab0
10008a9a    MOV         EDX, dword ptr [EBP + local_8]
10008a9d    ADD         EDX, dword ptr [EBP + local_c]
10008aa0    MOV         dword ptr [EBP + local_8], EDX
10008aa3    MOV         EAX, dword ptr [EBP + local_8]
10008aa6    CMP         dword ptr [EAX], 0x4550
10008aac    JNZ         LAB_10008ab0
10008aae    JMP         LAB_10008abb
```

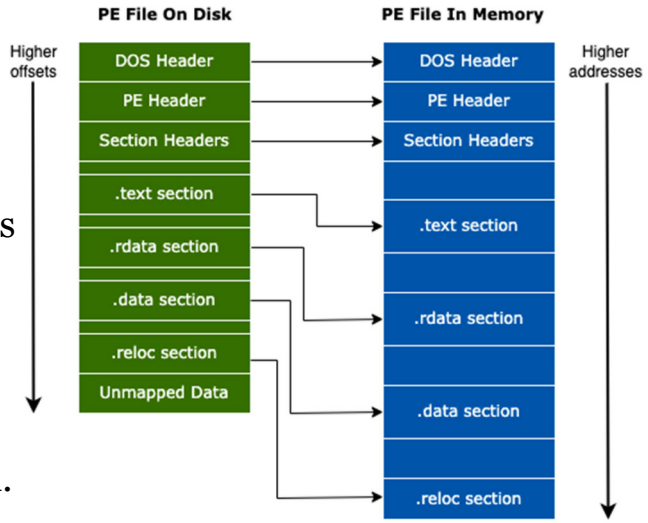

The code responsible for loading the next stage executable will often check if the binary is a valid Windows executable. Simple checks include looking for the well-known "MZ" characters represented by the hexadecimal value 0x4D5A. Other may checks may include looking for the "PE" signature (0x5045) in the header and checking the architecture (i.e., 32-bit or 64-bit).

This slide shows a couple checks performed by malicious code that executes a decoded DLL. Specifically, this code is associated with executing a Cobalt Strike beacon (the loader has SHA-256 hash 27b788aeb2c79b807d586b6ae3f5f30a7e11c6855f1a7685cf45cea9d24891c5).

The two checks on this slide evaluate if the underlying binary begins with the "MZ" characters and if the "PE" signature is present.

The Loader Maps the Windows Executable into Memory

- When a program is launched from disk, it is *mapped* (i.e., loaded) into memory prior to execution.
- The loader parses the executable's headers, allocates memory, and loads the content from disk.
- The memory mapped file is similar, but not identical, to the file on disk.
- A malicious loader maps an in-memory executable before execution.



When a legitimate executable is run from disk, the executable must be loaded into memory to be executed. The process of reviewing the PE headers and loading the necessary components of the binary file is referred to as "mapping" the executable into memory. The layout of the executable in memory is similar, but not identical, to the file on disk.

The image on this slide is based on the figure located in the Microsoft article at https://for710.com/pe-detail. It shows that most individual components of a PE file are loaded into memory, but the space between them change.

During malware analysis, we will often encounter in-memory executables that must be run. Although these executables are already in memory, they are still mapped to a different location in memory prior to execution.

The Mapped Executable Differs from the Unmapped One

- The ImageBase field is updated with the mapped EXE's base address.
- The .text section may change if base relocations apply.
- The .rdata section, which typically includes import information, may be updated if imports are bound.
- The .data section may be updated based on global variables.
- Content in other sections (.pdata, .rsrc, .reloc) typically is identical.
- Content not described in section headers (i.e., overlays) is not mapped.
- FileAlignment field is usually 0x200 (512), while SectionAlignment is 0x1000 (4096)—this results in different spacing between sections.

When comparing a PE file in memory vs. the one on disk, several differences are clear.

First, the `ImageBase` field is updated to reflect the actual base address of the mapped executable vs. the preferred address specified in the unmapped executable's ImageBase field.

The .text section in the mapped executable may differ if some addresses were updated with relocation information contains in the .reloc section. Base relocations describe locations in the code that must be updated if the executable is not loaded at the preferred base address (i.e., the address specified by the `ImageBase` in the optional header).

The .rdata section typically includes information about imported functions from external DLLs. This section on disk typically matches the section in memory, unless imports are bound. When an executable is bound, the binary on disk has the in-memory addresses of imported DLLs. In other words, functions do not need to be resolved during the loading process. For more information, browse to https://for710.com/inside-windows-part2 and view the section titled "Binding."

The .data section is updated based on static variables and global variables.

When Windows loads an executable into memory to prepare for execution, it consults the PE header. If the PE header doesn't describe certain data, it simply won't be loaded into memory. For example, overlay content (i.e., data after the end of file) is not loaded into memory.

An earlier slide explained the `FileAlignment` and `SectionAlignment` fields within the optional header—the difference in these values will result in additional space between sections in memory.

The `MEM_RESERVE | MEM_COMMIT` allocation type performs both operations. Although we can specify both a reserve and commit in one CALL, note that the operations occur separately in the background. When using this allocation type, the memory is reserved and committed on a range rounded to a multiple of allocation granularity (64K).

Memory can be committed *without* explicitly reserving a region if the `lpAddress` argument is NULL. On the other hand, if an address range is specified, committing without reserving first will fail. Also, committing the same memory region multiple times has no adverse impact, so a developer does not need to check if memory is already committed before committing.

`VirtualFree` (https://for710.com/virtualfree) is used to decommit and/or release a memory region.

VirtualAlloc and VirtualProtect Accommodate Mapped Content

- An initial CALL to VirtualAlloc typically allocates space for the entire mapped executable.
- For example:

```
VirtualAlloc(NULL, 0x14000, 0x3000, PAGE_READWRITE)
```

- Only one CALL to VirtualAlloc is necessary, but some loaders have additional CALLs to VirtualAlloc for each section.
- Malware may not need to allocate additional memory if it overwrites the current program in memory; VirtualProtect is used to allow write access.

An initial CALL to `VirtualAlloc` typically allocates space for the entire mapped executable. If this first call only reserves a memory region (i.e., it does not commit as well), additional CALLs to `VirtualAlloc` may be necessary.

For example, an Emotet malware loader with SHA-256 hash 18235AC8C4482D9C0CA96BE91ED18CBC601FA793F03D1820D8FFE492D6FF42EC uses the following `VirtualAlloc` arguments to reserve and commit memory for a large region that will accommodate the entire mapped executable:
VirtualAlloc(NULL, 0x14000, 0x3000, PAGE_READWRITE)

You can find this executable at Malware\Section1\hD53056.zip within the course VMs.

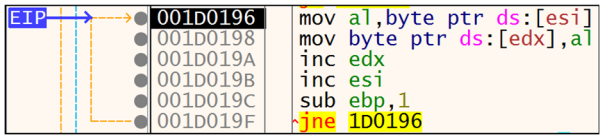
With this approach, only one CALL to VirtualAlloc is necessary. However, even if the initial `VirtualAlloc` function calls reserves and commits, some loaders will execute additional `VirtualAlloc` function calls for each section of the Windows executable. In this case, the additional function calls typically only include a `MEM_COMMIT` allocation type (Hint: we might see this in the upcoming lab).

It is also possible that the loader requires no additional memory for the mapped executable. For example, some malware overwrites the current executable in memory rather than allocate a new region. In this case, the loader code is often contained in second-stage shellcode located elsewhere in memory. This shellcode uses `VirtualProtect` to ensure the memory region is writeable, and then it overwrites the existing executable.

See an `IcedID` downloader with SHA-256 hash F28FDB464E38B3974EBFF5FE21B24B54B064C6A7076BDB733B6E5D55AE119BFB for an example of this approach.

Next, Header and Section Content Are Written

- A simple loop can move bytes from an unmapped to mapped section:



ESI: Pointer to unmapped executable
EDX: Pointer to memory region for mapped executable
Both registers increment by 1 per iteration

- Alternatively, the REP (Repeat String Operation) prefix can be combined with the MOVSB instruction to copy bytes
  - Example: `rep movsb`
  - REP performs the specified operation the number of times specified in ECX, or until a specified ZF condition is no longer met
  - MOVSB moves a single byte from the address in ESI to the address in EDI

After memory is allocated for the mapped executable, header, and section content is written to the memory region. Note that some malware loaders do not copy the DOS Header or the PE header—this header information must be processed by the loader, but it is not necessary to execute the mapped executable. Omitting the DOS and/or PE header from the mapped executable also means the "MZ" and "PE" signatures are not present—analysts often look for these visual cues to identify a Windows executable.

Multiple approaches can be used to copy header and section content to memory region set aside for the mapped executable. For example, the first image on this slide shows a loop that uses the MOV instruction to transfer bytes from the unmapped executable to the mapped executable. This loop executes for each section. This code resides in an Emotet loader with SHA-256 hash 18235AC8C4482D9C0CA96BE91ED18CBC601FA793F03D1820D8FFE492D6FF42EC.

Another approach you may encounter uses the REP (Repeat String Operation) instruction prefix. When placed in front of an instruction, it performs multiple operations repeatedly. Specifically, it repeats a string instruction based on the number contained in ECX or until a specified zero flag (ZF) is no longer met. REP is often paired with the MOVSB instruction (move byte) to copy content from an unmapped section to the address allocated for the mapped section. We will discuss an example on the next slide.

For more information on the REP prefix, see https://for710.com/rep.
For more information on MOVS, see https://for710.com/movs.

Memmove Can Copy Section Data for the Mapped Executable

- The C function `memmove` is commonly used to copy data from one location in memory to another.
- Ghidra's FID feature may identify this function within a malicious loader:

```
void *memmove(
   void *dest,
   const void *src,
   size_t count
);
```

```
CALL   memmove
```

- Alternatively, memmove may be imported by MSVCRT.DLL:

```
CALL  dword ptr [->MSVCRT.DLL::memmove]
```


The C function memmove is commonly used to move data from one location in memory to another.

As shown on the slide, the key arguments are:
- `dest`: Address of destination where data should be copied to.
- `src`: Address of source data.
- `count`: Number of bytes to copy.

The code for memmove may be identified by Ghidra's Function ID feature. Alternatively, it may be imported from MSVCRT.DLL.

For more information on memmove, see https://for710.com/memmove.

The Loader Loads Required DLLs and Resolves APIs

- With access to LoadLibrary and GetProcAddress, the loader loads DLLs and resolves the required functions.
- These APIs are often referenced in a loop to iterate over DLLs and functions.
- In this excerpt from a malware loader:
  - LoadLibraryA is called if the DLL is not already loaded in memory.
  - If the library loads as expected, GetProcAddress resolves functions.
  - Both APIs are in the loader's import table.

```
hModule = GetModuleHandleA(lpModuleName);
if (hModule == 0x0) {
[redacted]
  hModule = LoadLibraryA(lpModuleName);
  _DAT_140020c7c = DAT_140020c84 | _DAT_140020f34;
  if (hModule == 0x0) {
    return 0;
  }
[redacted]
while (pFVar2 != 0x0) {
  FUN_140003870();
  pFVar2 = *ppFVar3;
  if ((pFVar2 & 0x8000000000000000) == 0) {
    _DAT_140020d8c = 0x5a;
    lpProcName = pFVar2 + 2 + param_2;
  }
  else {
    lpProcName = pFVar2 & 0xffffffffffffff;
  }
  pFVar2 = GetProcAddress(hModule,lpProcName);
```


Next, the loader will load any DLLs and resolve APIs for the next-stage executable. At this stage, the loader has already resolved a `LoadLibrary` variant and `GetProcAddress`, so it just needs to call these APIs to load DLLs and resolve APIs in those DLLS. If a DLL is already in memory, it is simply mapped into the address space of the current process. Note that the `LoadLibrary` API will also recursively load DLLs as needed (i.e., if a loaded DLL depends on other DLLs, it will load those too).

The code excerpt on this slide is from a malware loader with SHA-256 hash ED5FBEFD61A72EC9F8A5EBD7FA7BCD632EC55F04BDD4A4E24686EDCCB0268E05. This loader imports `LoadLibraryA` and `GetProcAddress`, so the Windows Loader resolve these APIs. When it comes time to resolve the next-stage executable's dependencies, the loader simply calls these APIs as shown. The loader checks if the DLL is already loaded via a call to `GetModuleHandleA`. If it is not loaded, a call to `LoadLibraryA` is executed. After the necessary libraries are loaded, a while loop will call `GetProcAddress` to resolve each API in the specified DLL.

The Loader Processes the Base Relocation Table

- The Base Relocation Table contains locations that need to be fixed up if the executable is not loaded at its preferred address.

```
                28 54 09 80 01 00 00 00 PTR_DAT_1800957a0
1800957a0  addr                        DAT_180095428
```

- With ASLR enabled by default on recent Windows OS's, fixups need to be applied unless the executable's header is modified to opt out of ALSR.
- This Base Relocation Table is found via the Data Directories, and code responsible for loading an executable will check the RVA and Size fields:

Relocation Directory RVA	000001D0	Dword	000B7000	.reloc
Relocation Directory Size	000001D4	Dword	00000AEC	

The Base Relocation Table is used to fix up addresses in memory if an executable is not loaded at its preferred image base address. Since ASLR is enabled system-wide by default in all recent versions of Windows, executables are usually *not* loaded at their preferred base address (unless a change is made to a file's PE header to opt out of ASLR). Therefore, processing an executable's base relocation table is an import step to ensure code and data are executed and referenced properly.

For example, the image on this slide references an address stored at 1800957a0. The address stored at this location is 180095428 (little endian), and it assumes the executable is loaded at its preferred address 180000000. If it *is* loaded at this address, no additional work is necessary. However, if the executable is *not* located at 180000000, the content at 1800957a0 needs to be updated based on the detail between the executable's preferred base address and its actual base address.

Relocation information is processed by the loader that prepares for execution. Once the specified addresses are fixed up, the relocation information is no longer needed.

Discussing the relocation table in greater detail is out of scope for this course. For more information on the base relocation table, see:
https://for710.com/reloc
https://for710.com/inside-windows-part2 (see section "Base Relocations")

The Loader Updates Section Permissions of the Mapped Executable

- One malware loader includes the following sequence of CALLs:
  - `VirtualAlloc(0, 0x14000, 0x3000, PAGE_READWRITE)` → `0x1E0000`
  - *PE sections copied to allocated region*
  - `VirtualProtect(0x1E1000, <size>, PAGE_EXECUTE_READ, <address>)`
  - `VirtualProtect(0x1ED000, <size>, PAGE_READONLY, <address>)`
  - `VirtualProtect(0x1EE000, <size>, PAGE_READWRITE, <address>)`
  - `VirtualProtect(0x1F2000, <size>, PAGE_READONLY, <address>)`
  - `VirtualProtect(0x1F3000, <size>, PAGE_READONLY, <address>)`
- Another loader calls VirtualAlloc once with no references to VirtualProtect:
  - `VirtualAlloc(0, 0x2B000, 0x1000, PAGE_EXECUTE_READWRITE)`

As we discussed earlier, an initial call to `VirtualAlloc` is often used to allocate memory for the entire mapped executable. Depending upon the memory protection applied during this initial call, `VirtualProtect` may be called later to modify permissions on sections in the mapped executable.

For example, an Emotet loader makes the sequence of CALLs listed first on this slide. An initial CALL to `VirtualAlloc` allocates memory for the entire mapped executable with the `PAGE_READWRITE` protection attribute. Then, section content is copied into that location (the details of that step are not important here). After the section content is copied, multiple `VirtualProtect` calls modify the protection attributes of each section (for simplicity and readability, all arguments passed to `VirtualProtect` are not shown). For example, we see one section is modified to be executable; this is likely the .text section for the unpacked executable. The SHA-256 of this executable is 18235AC8C4482D9C0CA96BE91ED18CBC601FA793F03D1820D8FFE492D6FF42EC.

Alternatively, if the initial CALL to `VirtualAlloc` marks the entire region as read/write/executable, it may be unnecessary to call `VirtualProtect`. For example, another malware loader makes an initial CALL to `VirtualAlloc` with no CALLs to `VirtualProtect`. This example is shown at the bottom of the slide, and it corresponds to an executable with SHA-256 hash ED5FBEFD61A72EC9F8A5EBD7FA7BCD632EC55F04BDD4A4E24686EDCCB0268E05.

For a DLL, the Loader Often Executes an Exported Function (1)

```
00402145    PUSH    s_TaskStart_0040f4e8                ; Pointer to string "TaskStart"
0040214a    PUSH    EAX                                 ; Pointer to VA of PE signature in mapped DLL
0040214b    CALL    FUN_00402924

            FUN_00402924
00402924    PUSH    EBP
00402925    MOV     EBP, ESP
00402927    PUSH    ECX
00402928    MOV     EAX, dword ptr [EBP + param_1]      ; Pointer to VA of PE signature placed into EAX

00402932    MOV     EAX, dword ptr [EAX]                ; Places VA of PE signature in mapped DLL into EAX
00402934    ADD     EAX, 0x78                           ; 0x78 + VA of PE header = VA of Export
                                                        ; Directory RVA field.

00402940    MOV     ESI, dword ptr [EAX]                ; Places Export Directory RVA into ESI
00402942    MOV     EAX, dword ptr [ESI + ECX*0x1 + 0x18]  ; Places sum of Export Dir RVA, mapped image
                                                        ; base and 0x18 into EAX (VA of NumberOfNames)
00402946    ADD     ESI, ECX                            ; Adds image base (ECX) and Export Dir RVA
                                                        ; (ESI), placing the Export Dir VA into ESI
```


If the next-stage executable is a DLL, the loader will usually execute an exported function. One approach to accomplishing this task is to identify the function by name. The loader will follow the process we previously discussed for identifying an export function by name.

This slide and the two that follow include code excerpts from a malicious loader that executes a DLL's exported function by name. The details comments describe what key instructions accomplish to arrive at the appropriate export function address.

The loader referenced in these slides is associated with WannaCry and has SHA-256 hash EC3FD41B2298954946999DCB3145CBDC927A5CA9A150A8C57741DA5FE3198CDA. The DLL it loads has SHA-256 hash 1BE0B96D502C268CB40DA97A16952D89674A9329CB60BAC81A96E01CF7356830. Both these files can be found in the Malware\Section1 directory within the course VM (see tasksche.zip and kbdlv.zip).

For a DLL, the Loader Often Executes an Exported Function (2)

```
0040296b    MOV     EDI, dword ptr [ESI + 0x20]          ; Places Name Pointer Table RVA into EDI
0040296e    MOV     EBX, dword ptr [ESI + 0x24]          ; Places Export Ordinal Table RVA into EBX
00402971    ADD     EDI, ECX                             ; Adds image base (ECX) to place Name Pointer
                                                         ; Table VA into EDI
00402973    ADD     EBX, ECX                             ; Adds image base (ECX) to place Export
                                                         ; Ordinal Table VA into EBX

00402981    MOV     EAX, dword ptr [EDI]                 ; Places Name Pointer Table entry into EAX.
00402983    ADD     EAX, ECX                             ; Adds image base (ECX) to place VA of Name
                                                         ; Table entry into EAX (address of a string)
00402985    PUSH    EAX                                  ; Pushes VA of string in Export Name Table
00402986    PUSH    dword ptr [EBP + param_2]            ; Pushes pointer to string passed as argument
00402989    CALL    dword ptr [->MSVCRT.DLL::_stricmp]   ; Compares argument with a string in the
                                                         ; Export Name Table

00402993    JZ      LAB_004029b4                         ; If match, jump
```


This page intentionally left blank.

For a DLL, the Loader Often Executes an Exported Function (3)

```
004029b4    MOVZX   EAX, word ptr [EBX]                 ; Dereference offset in Export Ordinal Table
                                                        ; (increments as it looks for matching name).
                                                        ; In this case, zero.

004029bf    MOV     EDX, dword ptr [ESI + 0x1c]         ; Adding Export Dir VA (ESI) and 0x1c =
                                                        ; AddressOfFunctions field VA; Export Address
                                                        ; Table RVA placed into EDX
004029c2    LEA     EAX, [EDX + EAX*0x4]                ; Ordinal value (EAX) x size of RVA (4 bytes)
                                                        ; is added to Export Address Table RVA (EDX)
                                                        ; and placed into EAX
004029c5    MOV     EAX, dword ptr [EAX + ECX*0x1]      ; RVA of Export Address Table entry is added
                                                        ; to image base and dereferenced, placing RVA
                                                        ; of exported function into EAX
004029c8    ADD     EAX, ECX                            ; Image base (RCX) is added to export function
                                                        ; RVA and export function VA is placed in EAX

004029b3    RET                                         ; Function returns

00402158    CALL    EAX                                 ; Call export function
```


This page intentionally left blank.

Course Roadmap

- FOR710.1: Code Deobfuscation and Execution
- FOR710.2: Encryption in Malware
- FOR710.3: Automating Malware Analysis
- FOR710.4: Correlating Malware and Building Rules

SECTION 1

- Analyzing Code Deobfuscation
  - Lab 1.1: Investigating Code Deobfuscation Using Steganographic Techniques
- Identifying Program Execution
  - ***Lab 1.2: Analyzing Malicious Program Execution***
- Understanding Shellcode Execution
  - Lab 1.3: Analyzing Shellcode Execution

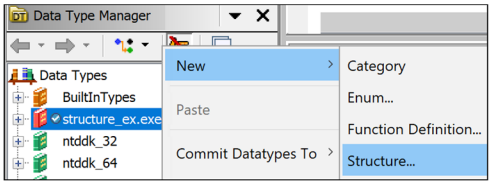
This page intentionally left blank.

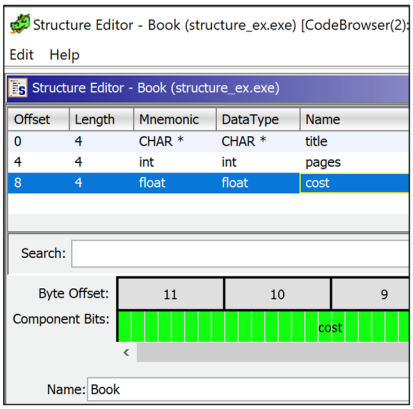
Lab 1.2: Background Topics: Define a New Structure

- The disassembled main() function includes three variables with no reference to a structure.

```
                    main                                            main
00407190    PUSH    EBP
00407191    MOV     EBP, ESP
00407193    SUB     ESP, 0xc
00407196    MOV     dword ptr [EBP + local_10], .rdata   ; = "Best Title Ever"
0040719d    MOV     dword ptr [EBP + local_c], 200
004071a4    MOVSS   XMM0, dword ptr [__real@41cfeb85]    ; = 25.99
004071ac    MOVSS   dword ptr [EBP + local_8], XMM0
```

- We can define the structure using Ghidra's Data Type Manager.

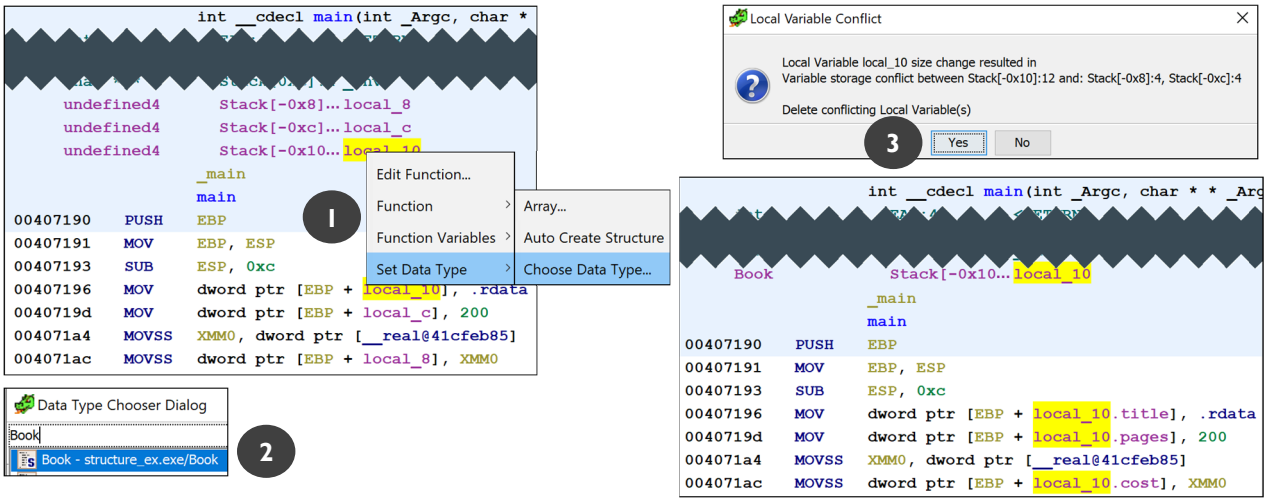




If we build a 32-bit Windows executable from the source code on the previous slide and disassemble it, we observe the code shown on the top left of this slide. Within the main function we observe three 4-byte variables that are assigned values. By the way, the XMM0 operand refers to an XMM register. XMM registers are used to perform calculations on values, though in this case XMM0 just contains the floating point number associated with the book cost.

Ghidra does not have any context about the user-defined Book structure, but we can explicitly define the structure using the Data Type Manager. First, right-click on the program name (called "structure_ex.exe" in this case) and browse to New > Structure. In the resulting Structure Editor, define the three structure members by clicking on each DataType field and typing the data types as shown. We can also add the structure member names under the Name column. When you close the Structure Editor, you will be prompted to save the structure (alternatively, click the the floppy disk icon on the top right of the window, not shown on this slide).

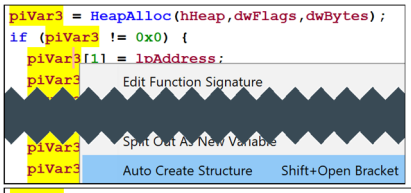
Lab 1.2: Background Topics: Apply a Defined Structure



After defining the structure, we can apply it to the values within the main function. Right-click on the first variable in the data structure (in this case, the book title is placed into local_10) and browse to Set Data Type > Choose Data Type. Then, search for the recently defined Book structure and click OK. When a prompt appears, agree to delete any existing variables as it applies the structure to data referenced in the function. Comparing the resulting code (see bottom right) with the original code (see top left), there is only one variable now with references to each structure member.

Lab 1.2: Background Topics: Auto Create a Structure (1)

```
piVar3 = HeapAlloc(hHeap,dwFlags,dwBytes);
if (piVar3 != 0x0) {
  piVar3[1] = lpAddress;
  piVar3[5] = *(piVar7 + 0x16) >> 0xd & 1;
  piVar3[7] = FUN_00401ca0;
  piVar3[8] = FUN_00401cc0;
  piVar3[9] = FUN_00401ce0;
  piVar3[10] = FUN_00401cf0;
  piVar3[0xb] = FUN_00401d10;
  piVar3[0xc] = 0;
  piVar3[0xe] = local_30.dwPageSize;
  iVar4 = FUN_00401740(piVar7[0x15]);
  if (iVar4 != 0) {
    local_8 = VirtualAlloc(lpAddress,piVar7[0x15],0x1000,4);
    FUN_00403910(local_8,param_1,piVar7[0x15]);
    iVar4 = param_1[0xf];
    *piVar3 = local_8 + iVar4;
```




```
piVar3 = HeapAlloc(hHeap,dwFlags,dwBytes);
if (piVar3 != 0x0) {
  piVar3->field_0x4 = lpAddress;
  piVar3->field_0x14 = *(piVar7 + 0x16) >> 0xd & 1;
  piVar3->field_0x1c = FUN_00401ca0;
  piVar3->field_0x20 = FUN_00401cc0;
  piVar3->field_0x24 = FUN_00401ce0;
  piVar3->field_0x28 = FUN_00401cf0;
  piVar3->field_0x2c = FUN_00401d10;
  piVar3->field_0x30 = 0;
  piVar3->field_0x38 = local_30.dwPageSize;
  iVar3 = FUN_00401740(piVar7[0x15]);
  if (iVar3 != 0) {
    local_8 = VirtualAlloc(lpAddress,piVar7[0x15],0x1000,4);
    FUN_00403910(local_8,param_1,piVar7[0x15]);
    iVar3 = param_1[0xf];
    piVar3->field_0x0 = local_8 + iVar3;
```

We can apply a structure to the above array to better understand how individual members are used.

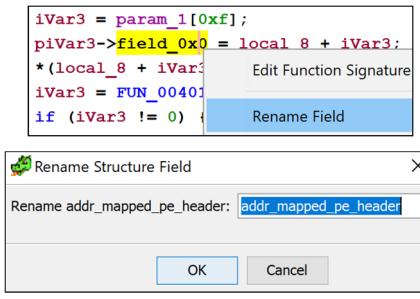
Ghidra has the ability to automatically create structures. Let's explore this feature when reviewing code from the malware sampled with SHA-256 B2A8F6976EFF160CEED3673C0979E37BC87EED2700881E9DFEA274C8D62D692B. You can find this sample at Malware\Section1\fileReader.zip.

Within the function FUN_00401d20, there is code that assigns values to an array named piVar3 in the decompiler output (see image on left). Additional analysis reveals this array is used to store data of various types, and the array is passed to functions that operate on its values. It is helpful to define this array as a structure, and Ghidra's ability to automatically create a structure can help.

To automatically create a structure, right-click on the array in the decompiler output and choose Auto Create Structure. The result is shown in the image on the bottom right of this slide.

Lab 1.2: Background Topics: Rename a Structure Member

- The structure's first member is the address of the mapped PE header.
- We can rename this field using the context menu.



- Notice piVar3 is passed to FUN_00401900.

```
iVar3 = param_1[0xf];
piVar3->addr_mapped_pe_header = local_8 + iVar3;
*(local_8 + iVar3 + 0x34) = lpAddress;
iVar3 = FUN_00401760(param_1,param_2,piVar7);
if (iVar3 != 0) {
  iVar3 = *(piVar3->addr_mapped_pe_header + 0x34) - local_c[0xd];
  if (iVar3 == 0) {
    piVar3->field_0x18 = 1;
  }
  else {
    bVar1 = FUN_00401a80(iVar3);
    piVar3->field_0x18 = CONCAT31(extraout_var,bVar1);
  }
  iVar3 = FUN_00401b20();
  if (((iVar3 != 0) &&
      (bVar1 = FUN_00401900(piVar3), CONCAT31(extraout_var_00,bVar1)
```

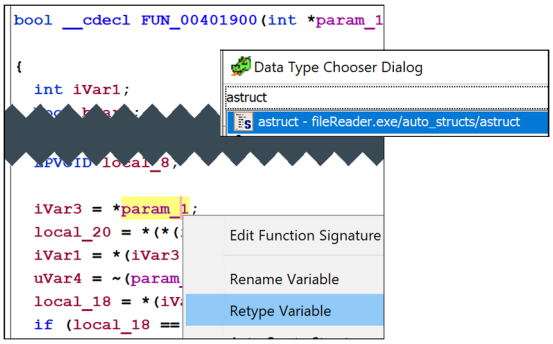

If we continue code analysis of this sample to understand the purpose of each member within the defined structure, we discover that the first element contains the address of a mapped PE header. This information is relevant in the context of a malware loader preparing to execute next stage content.

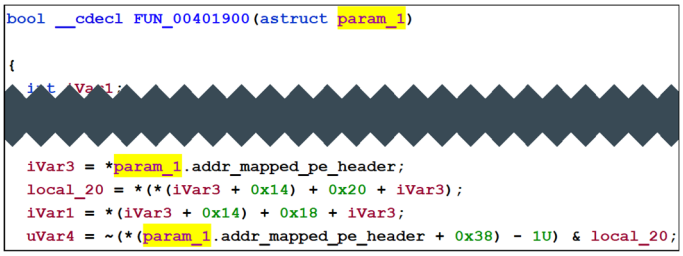
We can rename the first structure member to document our analysis. As shown on this slide, right-click on the relevant member and choose to Rename Field. The result, shown on the right, shows that our renamed member is clearly referenced multiple times in the code.

Later in the code, FUN_00401900 is called and the structure is passed as an argument. Let's investigate.

Lab 1.2: Background Topics: Apply an Auto Created Structure

- When we arrive at FUN_00401900, we can "retype" the argument to apply our structure.
- The resulting code now references the structure's members.

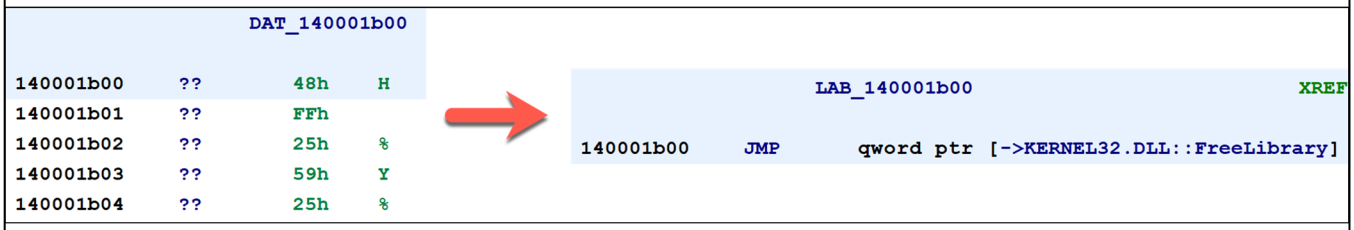




On the previous slide, we noticed that the defined structure is passed as an argument to FUN_00401900. When we analyze that function, we can retype the argument to add more context to this code. To retype a variable, right-click on the variable and choose Retype Variable. Then, type the name of the structure (in this case, astruct) and choose the resulting option. The function's code now properly references the structure's individual members. In this case, we see multiple references to the member we renamed on the previous slide. We will stop our analysis of this sample here, but you will have an opportunity to explore similar code in the upcoming exercise.

Lab 1.2: Background Topics: Identifying Code vs. Data in Ghidra

- Within Ghidra, `DAT_` refers to generic data.
- Evaluating the context of the reference might provide some direction.
  - If the data is passed as an argument to a function, we could jump to the function to better understand how the argument is used.
  - We could debug the code to observe what occurs during execution.
  - Alternatively, type "D" to disassemble the content and assess if it is code.

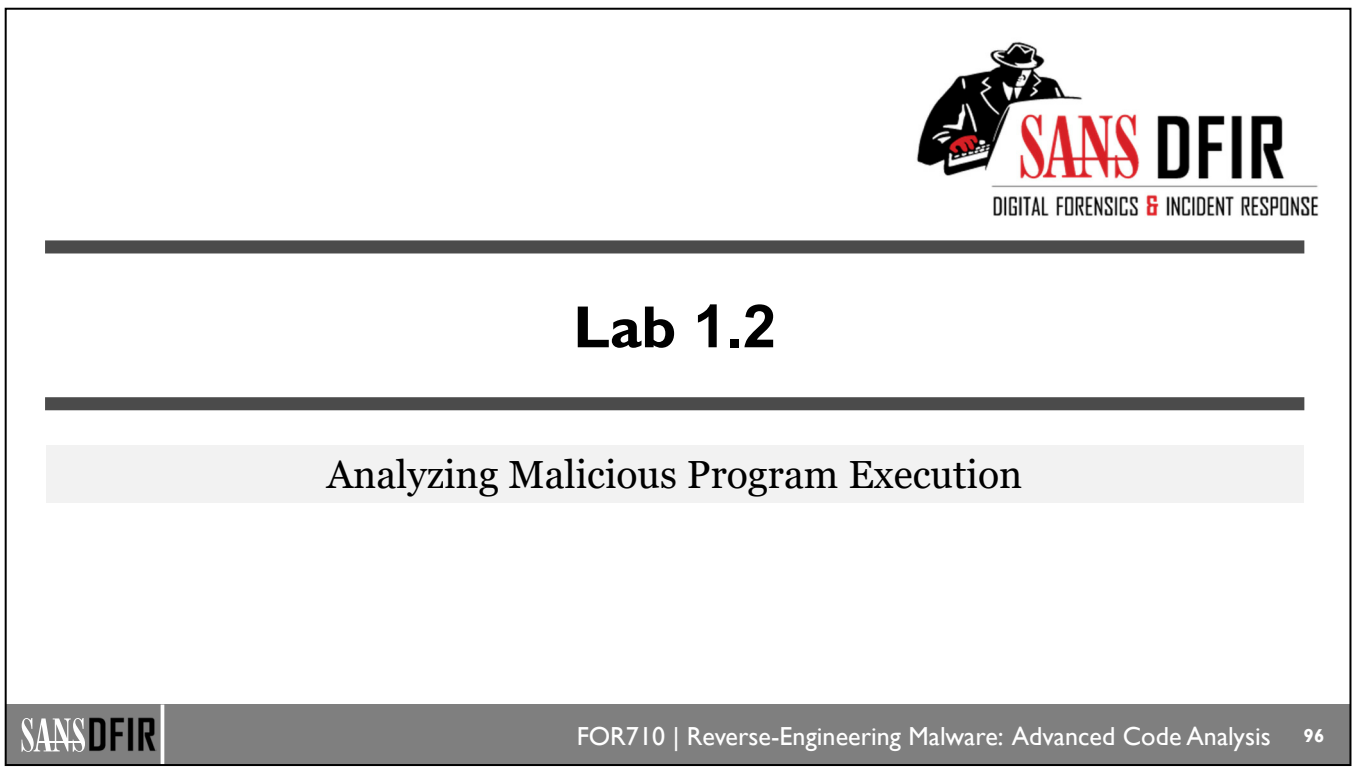


Sometimes, Ghidra does not accurately identify code or data. Ghidra uses the DAT_ label to refer to generic data. This content might be a string, code, or something else.

The best way to determine if the generic data is something meaningful is to evaluate its context. You can assess how the data is used via static code analysis or a debugger.

One quick way to evaluate if the data is actually code is to type "D" on the keyboard to disassemble the relevant bytes. Does the new representation look like legitimate code?

At the bottom of this slide is an example where generic data is converted to code. This is a helpful tip for the upcoming lab.



Please begin Lab 1.2 now.

Analyzing Program Execution: Module Objectives, Revisited

- ✓Understand the key components of a Windows Executable header.
- ✓Identify the structures and fields associated with a program's imports.
- ✓Identify the structures and fields associated with a program's exports.
- ✓Understand the steps necessary to prepare a program for execution.
- ✓Recognize code that maps an executable into memory.
- ✓Determine the code execution entry point for a second-stage binary.

This page intentionally left blank.

Course Roadmap

- FOR710.1: Code Deobfuscation and Execution
- FOR710.2: Encryption in Malware
- FOR710.3: Automating Malware Analysis
- FOR710.4: Correlating Malware and Building Rules

SECTION 1

- Analyzing Code Deobfuscation
  - Lab 1.1: Investigating Code Deobfuscation Using Steganographic Techniques
- Identifying Program Execution
  - Lab 1.2: Analyzing Malicious Program Execution
- ***Understanding Shellcode Execution***
  - Lab 1.3: Analyzing Shellcode Execution

This page intentionally left blank.

Understanding Shellcode Execution

This page intentionally left blank.

Shellcode Is Often Executed as One Component of Multi-Stage Malware

- Recall that shellcode is self-contained executable code.
- Shellcode is common because it is easy to obfuscate, harder to identify, and more challenging to analyze vs. a traditional Windows EXE.
- Offensive security tools, including Metasploit and Cobalt Strike, use shellcode to accomplish their goals.
- Like the loader code we analyzed in the last module, shellcode must load libraries and resolve function addresses to accomplish its goals.
- We will *not* use emulators or behavioral techniques, which were covered in FOR610; we will focus more on static and dynamic code analysis.

As a reminder, shellcode is self-contained executable code. It is comprised of opcodes that execute independently, without the typical structure of a Windows executable. For example, shellcode does not include a header, sections, or an import address table. Shellcode is Position Independent Code (PIC) because it does not depend on hardcoded addresses or assume it is loaded at a certain location in memory.

Shellcode is common in malware because it is easy to obfuscate and harder to identify and analyze when compared to a traditional Windows executable. Also, it is simple to spot the "MZ" ASCII characters or 4D5A hexadecimal values associated with the beginning of a Windows executable, but identifying shellcode is not as straightforward.

Shellcode may be used to perform an exploit, serve as the exploit payload, or execute as one component of multi-stage malware. Various red team tools, including Metasploit and Cobalt Strike, use shellcode to accomplish their goals. Shellcode execution can have severe impact in relatively few instructions.

Although shellcode presents many benefits to the attacker, it also presents challenges. Because there is no import address table, for example, shellcode must perform the heavy lifting associated with resolving API addresses.

We need to be prepared to extract and analyze shellcode at a code level. We'll combine debugging and static code analysis to understand the details of how shellcode operates when it is used as one component of multi-stage execution.

Note that we will *not* explore the use of shellcode emulators, behavioral analysis, or shellcode-to-EXE converters for our analysis. These techniques are discussed in FOR610, and we want to focus on code analysis.

To Identify Shellcode, Look for Common Opcodes

- **FC:** This translates to the instruction CLD (clear direction flag)
- **EB:** This is the opcode for a relative jump instruction.
- **E8:** This is the opcode for a CALL instruction.
- **55 8B EC:** This translates to the instructions push ebp and mov ebp,esp, commonly seen at the beginning of a function (i.e., the function prologue) in x86.

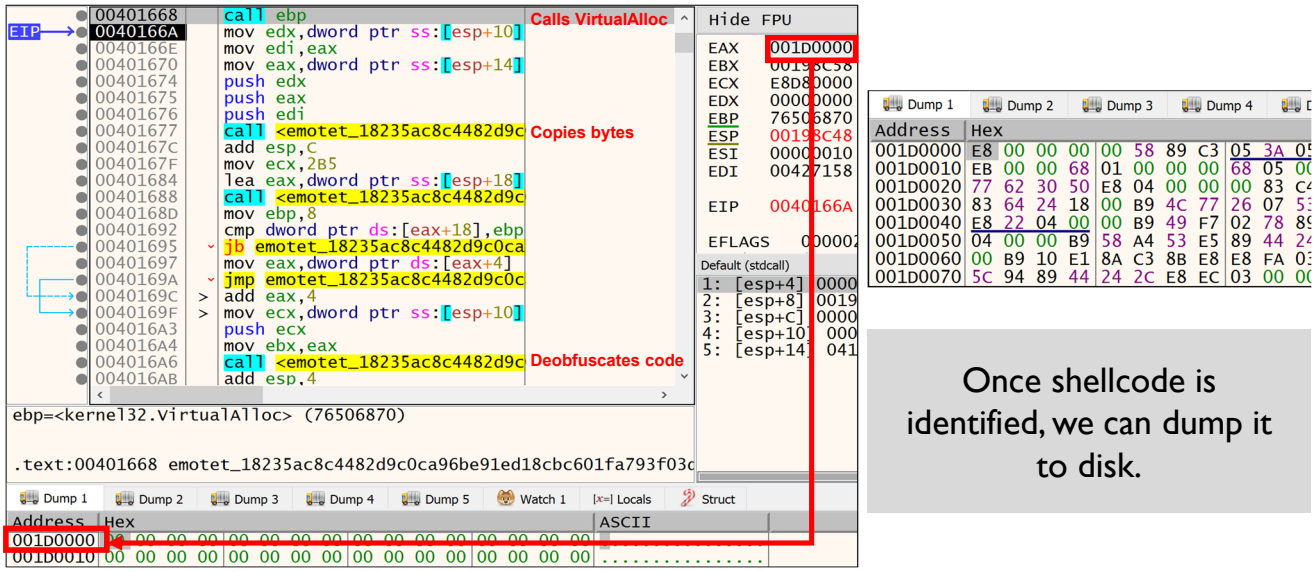
Common byte sequences we encounter when identifying shellcode include the following:

- FC: This translates to the instructions CLD (clear direction flag). It is often followed by the E8 (CALL) opcode. These bytes are typical for shellcode used by Cobalt Strike and Metasploit.
- EB: This is an opcode for a relative jump instruction.
- E8: This is an opcode for a CALL instruction.
- 55 8B EC: This translates to the instructions push ebp and mov ebp,esp, commonly seen at the beginning of a function (i.e., the function prologue) in x86. This sequence of bytes is specific to 32-bit code, while the others apply to both 32-bit and 64-bit code.

Look for these byte values at the beginning of newly allocated memory regions.

Note that this is not an exhaustive list of shellcode opcodes.

Example: VirtualAlloc Allocates Memory for Shellcode

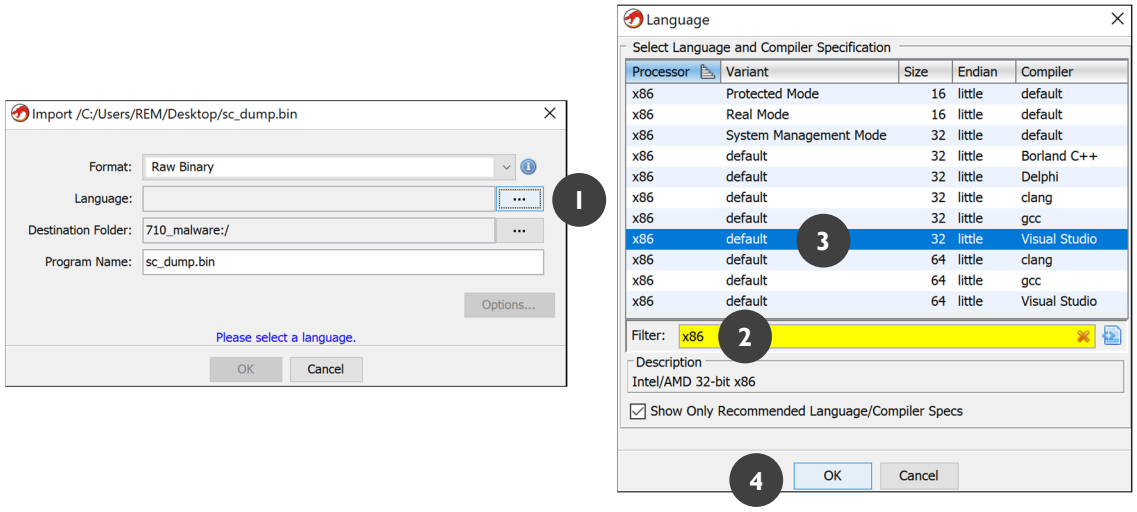


We will begin walking through an example to discuss key aspects of shellcode execution. The code on this slide is from a 32-bit Emotet loader with SHA-256 hash 18235AC8C4482D9C0CA96BE91ED18CBC601FA793F03D1820D8FFE492D6FF42EC. You can find this executable at Malware\Section1\hD53056.zip within the course VMs.

As described on this slide, this loader calls VirtualAlloc to allocate memory. In this case, the allocated memory region begins at address 1D0000. The function call at 401677 copies bytes to the allocate region, and the function call at 4016A6 deobfuscates that content to produce shellcode. A screenshot of the resulting shellcode is shown on the right of this slide. Observe the first byte E8, which translates to a CALL instruction. As discussed, on the previous slide, this is a common first byte in shellcode.

To dump the shellcode to disk, right-click on the first byte in the dump window and browse to Follow in Memory Map. Then, choose the highlighted memory region and select Dump Memory to File.

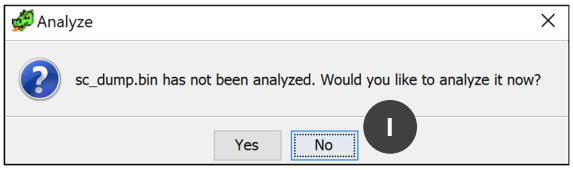
Load the Shellcode into Ghidra and Choose a Language



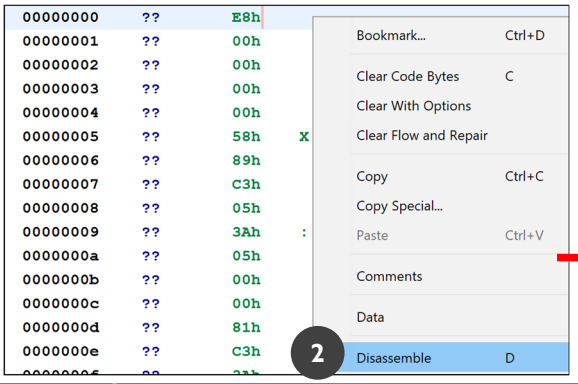
After dumping the shellcode to disk, load it into Ghidra for additional analysis. In the screenshot on this slide, the shellcode was dumped to a file called sc_dump.bin. Then, this file was drag-and-dropped into a Ghidra project. When the Import window opens, Ghidra instructs the user to select a language. As a reminder, shellcode does not include any header or other supporting information, so Ghidra needs to be told what type of code is contained within the file.

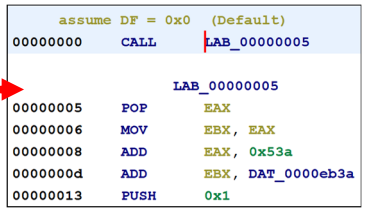
In the example of a 32-bit Emotet loader, the shellcode is also 32-bit. To select the appropriate option, first click on the dot-dot-dot (…) button. Then, from the large list of languages, filter by "x86". Next, choose the option that specifies a size of 32 and Visual Studio as the compiler—this option works for most 32-bit shellcode. Finally, click OK.

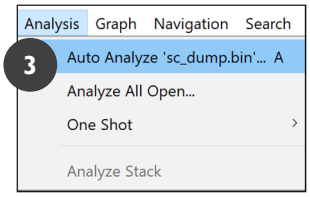
Manually Disassemble the Code Once the File Is Loaded



- Do not auto analyze the file at first.
- Use the context menu or type "D" to manually disassemble.
- Addresses are just raw offsets since the file is not a typical Windows executable.
- Finally, auto analyze the file.







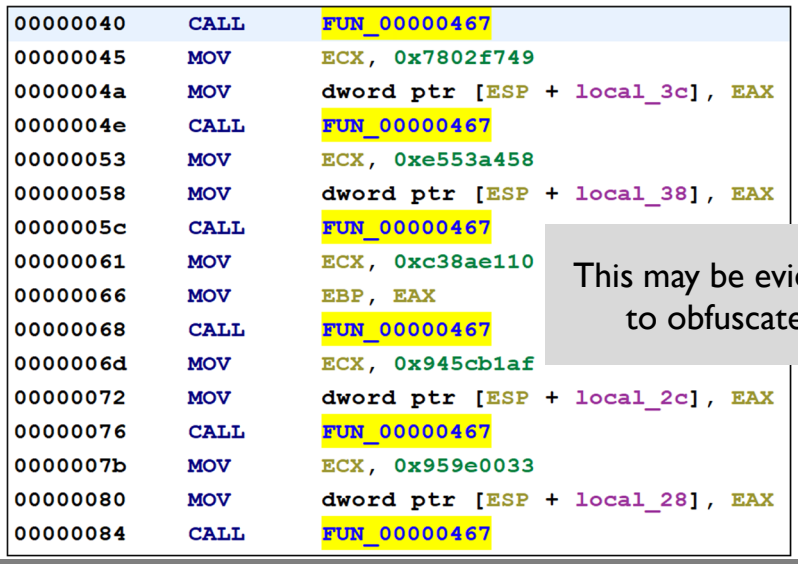
After double-clicking on the file in the Ghidra project, the Analyze popup will ask if the file should be auto-analyzed. Choose *not* to analyze the file—we will delay this step until after the code is displayed properly.

Next, we will need to manually disassemble the code. To do so, simply click on the first byte and click "D" on the keyboard. Alternatively, right-click and choose Disassemble from the context menu.

Once the code is disassembled, recognizable assembly instructions will be visible. Note that addresses in the first column are simply offsets beginning at zero. Since the file is not a traditional Windows executable, it does not specify an image base or section sizes.

Finally, browse to Analyze > Auto Analyze from the menu bar.

The Shellcode Calls a Function Repeatedly with Different Arguments



This may be evidence of API hashing to obfuscate function names.

The shellcode includes multiple CALLs to the same function, but with different hexadecimal values passed as arguments. In the screenshot on this slide, we see different hexadecimal values passed to ECX before function FUN_00000467 is called. This observation may indicate the use of API hashing to obfuscate function names. We will discuss API hashing in more detail on the next slide.

Shellcode Commonly Uses API Hashing to Resolve Functions

- Remember that shellcode needs to manually resolve APIs.
- To avoid alerting strings in the code, shellcode may resolve APIs based on a hash that includes a hashed DLL name and a hashed function name.
- ROR-13 is a popular algorithm and seen in Metasploit and Cobalt Strike code; CRC32, DJB2, FNV-1a, or a custom algorithm may also be used.
- Shellcode often resolves LoadLibrary and GetProcAddress first, so it can use these APIs to load other modules and resolve additional functions.

While this module focuses on shellcode, API hashing may appear in Windows executables too.

Recall that shellcode does not include an import address table, so it must manually load libraires and resolve function addresses to accomplish its goals. Rather than the actual strings associated with Windows APIs in the shellcode, malware developers often obfuscate the code's dependencies using API hashes. An API hash is based on performing a mathematical operation against a combination of a DLL name and function name. Based on a provided API hash, shellcode can locate the necessary DLL and resolve the required function address. Shellcode commonly resolves `LoadLibrary` and `GetProcAddress` first, so it can use these APIs to load other modules and resolve additional function addresses.

API hashing could involve any chosen algorithm, but certain algorithms are common in malicious shellcode. For example, the ROR-13 algorithm is popular in shellcode associated with Metasploit and Cobalt Strike. This algorithm moves each bit in a value 13 bits to the right.

The GuLoader uses DJB2 algorithm for API hashing (see https://for710.com/guloader).

A PlugX variant and the Matanbuchus loader both use the FNV-1a hash algorithm to resolve APIs. For more information on these examples, see https://for710.com/plugxhash and https://for710.com/matanbuchus.

Note that although this module focuses on shellcode, malicious Windows executables may include API hashing as well. For example, some variants of the VMZeus malware family use CRC32.

Shellcode Compares a Hard-Coded API Hash with a Calculated API Hash to Locate the Required Module and Function

- Shellcode commonly passes a hardcoded hash to a function to resolve APIs.
- Then, the function:
  - Iterates over all modules loaded by the process.
  - For each module, the module name (i.e., kernel32.dll) is hashed.
  - For each exported function in the module, the function name is hashed.
  - The combined hash (i.e., the addition of module name hash + function name hash) is compared against the hash passed to the function.
  - If there is a match, the code resolves the address of the function so it can be called; if there is no match, go to the next loaded module and repeat.

This slide provides an overview of how an API hash is used to resolve a function address. We will discuss the code that performs these steps shortly.

Shellcode Frequently Accesses the PEB to Resolve APIs

- The Process Environment Block (PEB) is a data structure in memory that contains information about the running process, including:
  - Loaded modules
  - If the process is being debugged
  - Process parameters (e.g., current directory or command line)
- Shellcode accesses the PEB to:
  1. Enumerate DLLs loaded in memory
  2. Access each DLL's exported functions
  3. Resolve Windows API addresses

Since shellcode does not include an import address table, it must manually resolve Windows API addresses. To accomplish this goal, it accesses the Process Environment Block (PEB). The PEB is a data structure in memory that contains information about the current process. Most importantly, it includes a list of modules loaded by the current process. By traversing this list and locating important loaded modules like kernel32.dll, shellcode can acquire the address of Windows APIs like `LoadLibrary` and `GetProcAddress`. With this information, the code can perform the tasks necessary to accomplish its goals.

You can read more about the PEB at https://for710.com/peb.

The Shellcode We Loaded into Ghidra References FS:[0x30]

```
00000040    CALL    FUN_00000467
00000045    MOV     ECX, 0x7802f749
0000004a    MOV     dword ptr [ESP + local_3c], EAX
0000004e    CALL    FUN_00000467
00000053    MOV     ECX, 0xe553a458
00000058    MOV     dword ptr [ESP + local_38], EAX
0000005c    CALL    FUN_00000467
00000061    MOV     ECX, 0xc38ae110
00000066    MOV     EBP, EAX
00000068    CALL    FUN_00000467
0000006d    MOV     ECX, 0x945cb1af
00000072    MOV     dword ptr [ESP + local_2c], EAX
00000076    CALL    FUN_00000467
0000007b    MOV     ECX, 0x959e0033
00000080    MOV     dword ptr [ESP + local_28], EAX
00000084    CALL    FUN_00000467
```

```
                    FUN_00000467

00000467    SUB     ESP, 0x10
0000046a    MOV     EAX, FS:[0x30]
00000470    PUSH    EBX
00000471    PUSH    EBP
00000472    PUSH    ESI
00000473    MOV     EAX, dword ptr [EAX + 0xc]
00000476    PUSH    EDI
00000477    MOV     dword ptr [ESP + local_8], ECX
0000047b    MOV     ESI, dword ptr [EAX + 0xc]
0000047e    JMP     LAB_0000050d
```

- The first MOV instruction places the PEB address into EAX.
- The next MOV references offset oxC within the PEB—why?

As a reminder, we started analyzing shellcode from a 32-bit Emotet loader. The screenshot on the left mirrors one from an earlier slide. Recall that hexadecimal values passed to a function in shellcode may represent API hashes used to obfuscate DLL and function names.

If we jump to FUN_00000467, we immediately observe a reference to FS:[0x30]. As discussed on the previous slide, this retrieves the address of the PEB in 32-bit code. This observation suggests FUN_00000467 will access the PEB and supports our theory that this function is used to resolve Windows APIs.

The MOV instruction at address 0000046a places the address of the PEB into EAX.

At 00000473, the MOV instruction accesses an offset within the PEB. What resides at that location? We can use WinDbg to investigate this further.

WinDbg Provides Insight into Process Structures in Memory

- WinDbg is a Microsoft debugger often used to analyze kernel-mode, user-mode code, and crash dumps.
- WinDbg is a powerful debugger but has a steep learning curve and is not considered user-friendly.
- WinDbg *Preview* provides an updated, more modern debugging experience with a refreshed interface; it also has a dark theme.
- We will only use WinDbg to debug the user-mode loader and review process structures in memory, including the PEB.

For simplicity, we may refer to WinDbg Preview as WinDbg in upcoming slides.

`WinDbg` is a powerful debugger that can be used to analyze kernel-mode code, user-mode programs, and crash dumps. Historically, it has had a steep learning curve and is considered to provide an unfriendly user experience by most analysts. However, in 2017 Microsoft released `WinDbg` *Preview* (also referred to as WinDbgX). This new version provides a more modern GUI and, in general, a more pleasant user experience. For example, `WinDbg` Preview includes a ribbon for easy access to a variety of views and windows. Users of Visual Studio will find it approximates that interface. Oh, and it includes a dark theme ☺. WinDbg Preview is available through the Microsoft Store.

For those who used the older `WinDbg`, all prior commands and extensions are still relevant. At the time of this writing, both `WinDbg` and `WinDbg` Preview are supported by Microsoft.

As a kernel-mode debugger, `WinDbg` can be used to analyze kernel-mode malware by configuring the application to perform remote debugging. This requires two systems or virtual machines.

`WinDbg` is also helpful in cases where malware employs the Heaven's Gate technique. This approach allows malicious code to execute 64-bit code from 32-bit code. However, with the roll out of Control Flow Guard in Windows 10, this technique is largely mitigated.

For more background information on WinDbg, see:
https://for710.com/windbg1
https://for710.com/windbg2

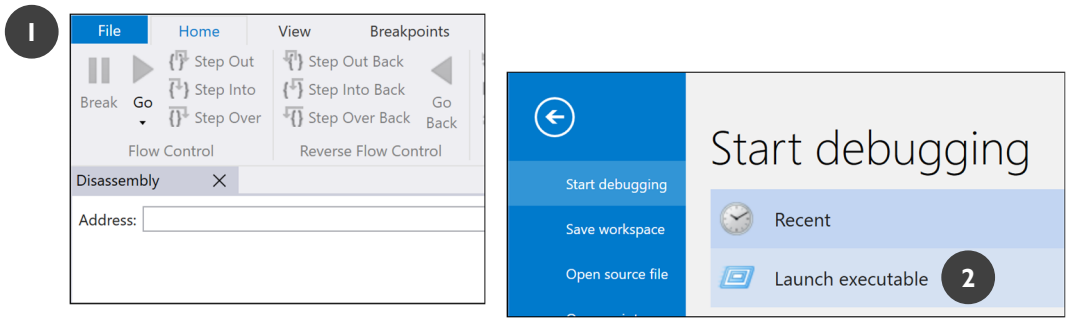
The Windows VM Has Additional Symbol Information Included

- WindDbg can automatically download symbols from a Microsoft server during analysis, but this requires an internet connection.
- To include symbols in an offline instance:
  - Provide an initial internet connection.
  - Download and install the Windows 10 SDK.
  - Install “Debugging Tools for Windows” only.
  - Use symchk to download symbols for kernel32.dll and ntdll.dll at a minimum.

- symchk "C:\Windows\system32\kernel32.dll" /s SRV*c:\symbols*http://msdl.microsoft.com/download/symbols
- symchk "C:\Windows\system32\ntdll.dll" /s SRV*c:\symbols*http://msdl.microsoft.com/download/symbols
- symchk "C:\Windows\system32\KernelBase.dll" /s SRV*c:\symbols*http://msdl.microsoft.com/download/symbols
- symchk "C:\Windows\SysWOW64\kernel32.dll" /s SRV*c:\symbols*http://msdl.microsoft.com/download/symbols
- symchk "C:\Windows\SysWOW64\ntdll.dll" /s SRV*c:\symbols*http://msdl.microsoft.com/download/symbols
- symchk "C:\Windows\SysWOW64\KernelBase.dll" /s SRV*c:\symbols*http://msdl.microsoft.com/download/symbols

First, Load the 32-Bit Executable into WinDbg Preview

- Launch WinDbg Preview from the desktop shortcut
- From the menu bar, click **File**
- Then, choose **Launch executable** and choose `hD53056.exe`

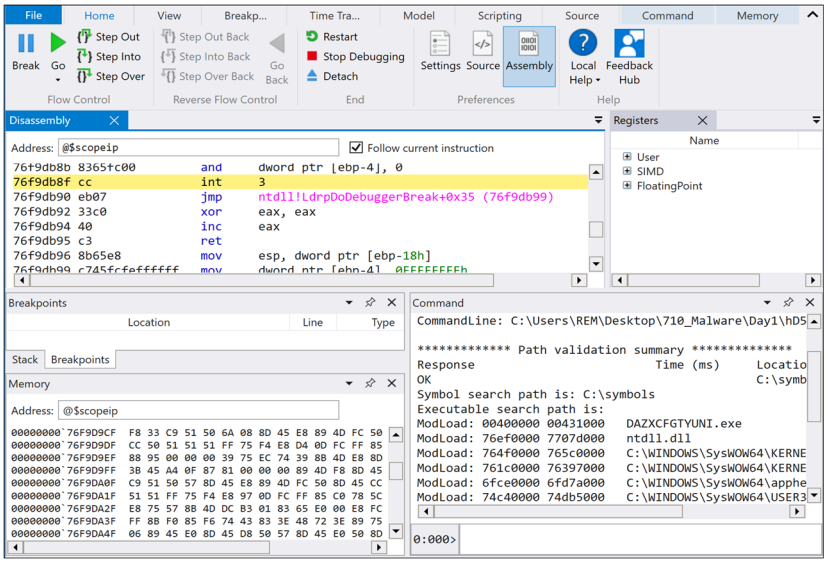


Let's load the 32-bit Emotet loader into `WinDbg` to better understand how the shellcode uses the PEB. Note that this slide describes loading the executable, not the shellcode.

First, launch `WinDbg` Preview from the 710 VM desktop. Then, click on File in the menu bar and choose to Launch executable. When the File Open prompt appears, choose the file of interest. After taking these steps, you will not see any code displayed in WinDbg (assuming you are opening it for the first time). See the upcoming slide for next steps.

Update the WinDbg Window Layout to View Code

- Within the View tab click Layouts > Disassembly.
- Within the Home tab choose Restart.
- Rearrange windows as desired.
- Layout changes are saved after closing WinDbg.



After completing the steps on the previous slide, there is no useful content displayed by default. To begin reviewing code, we must change the window layout within WinDbg.

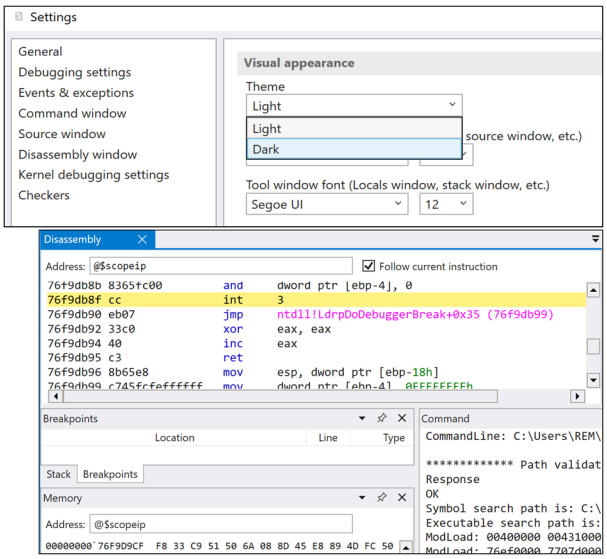
Click on the View tab and choose Layouts > Disassembly. Then, return to the Home tab and choose Restart. You should now see some content, including disassembly.

To re-arrange windows within the interface, drag and drop each title bar as desired. The screenshot on this slide shows a layout that resembles what we see in x64dbg and includes (starting from the top and going clockwise):

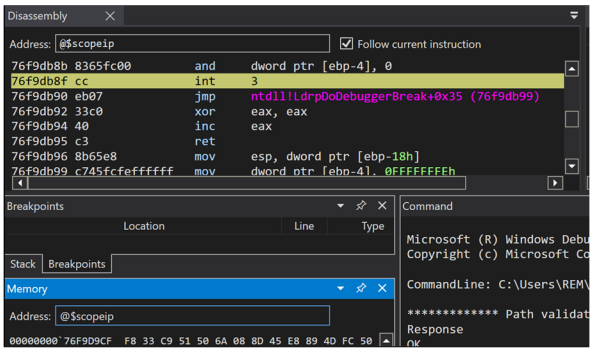
- **Disassembly:** Instructions to be executed.
- **Registers:** The various registers, which can be expanded to view values.
- **Command:** A command input field, with the output shown directly above it. Initially, this output describes the DLL loaded based on the import address table of the target executable.
- **Memory:** An area to view the content at a specified address in memory.
- **Breakpoints:** A view to track any breakpoints set. To access this window, browse to the View tab and choose Breakpoints.

You can modify the Window layout based on your preferences. Any changes to the Window layout are saved globally once `WinDbg` Preview is closed.

To Enable a Dark Theme, Browse to File > Settings > General



The slides will use the Light theme, but feel free to modify your interface.



Including a dark mode is popular in most applications, and `WinDbg` delivers on this front. As shown on this slide, feel free to modify your GUI and enable the Dark theme.

WinDbg Command Types

Built-in commands query and control the debugged executable

- **`g:`** go / continue executing (add **`u`** to run until after return)
- **`bp <address>:`** Set a breakpoint at the specified address
- **`ba <access><size> <address>:`** Set access (R/W/E) breakpoint at specified address for number of bytes
- **`bl:`** List breakpoints
- **`lm:`** List loaded modules (add **`f`** for full path information)
- **`dt <structure> <address>:`** Display type information (add **`-r`** to recursively dump subtype fields)
- **`?:`** Evaluate expression
- **`r:`** Print register information

The purpose of these commands will be clearer when we apply them to our example.

To interact with `WinDbg`, we need to understand its command types. This slide lists important built-in commands. These notes contain a few additional notes on certain commands.

The command structure for dt on this slide is in the format we will typically use. However, there are additional optional parameters. For more information on this command, see https://for610.com/dt.

When using a break on access breakpoint (ba), if the access is *e* for execute, the size must be one (see https://for710.com/ba).

If this information seems too abstract right now, don't worry—the value of these commands will be clearer when we apply them to an example.

Other Command Types Include Meta and Extension Commands

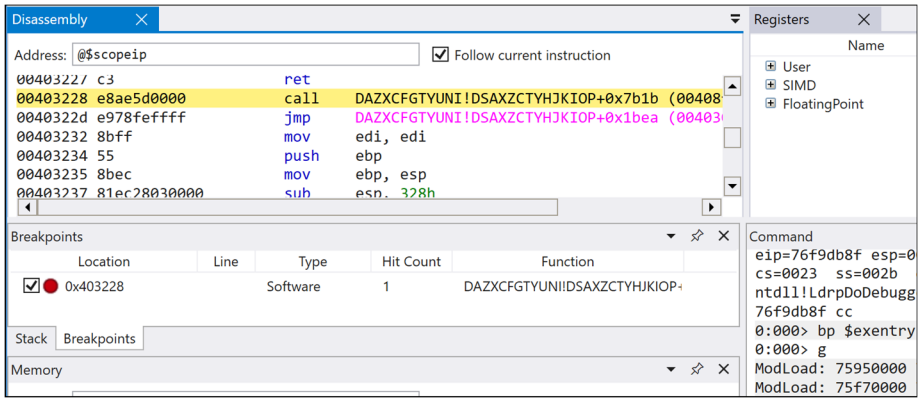
- Meta commands (a.k.a. dot commands) control the debugger
  - **`.help`:** Display list of meta commands
  - **`.restart`:** Restart debugged executable
  - **`.writemem`:** Dump memory to disk
- Extension commands (a.k.a. bang commands) from WinDbg extensions
  - **`!dh <address>:`** Displays header information for a file at the address
  - **`!address:`** Displays memory map. Add an address to get module details
- Use the keyboard up arrow for command history

This page intentionally left blank.

WinDbg Initially Pauses in ntdll.dll

Set a breakpoint and run the program to arrive at the entry point

- **`bp $exentry:`** Set a breakpoint at the target's entry point
- **`g:`** Run the program



When we load a program into a debugger, we expect it to pause the target executable at the entry point. Unfortunately, `WinDbg` does not do this. Instead, it pauses in ntdll.dll during the loading process. To arrive at the program's entry point, we must set a breakpoint and run the executable.

To set a breakpoint at the entry point, we will use a pseudo-register. Type the command: bp $exentry. Next, run the program with the g command.

After executing these two commands, the debugger should pause at the entry point at address 403228 (as shown on the slide). In the Breakpoints view, observe the breakpoint that was set. In the Command view, note a history of the commands run.

View Loaded Modules and Use the Base Address to Display Headers

```
0:000> lm
start    end        module name
00400000 00431000   DAZXCFGTYUNI   (export symbols)
73810000 7381a000   CRYPTBASE      (deferred)
74c40000 74db5000   USER32         (deferred)
74fc0000 7507e000   RPCRT4         (deferred)
755a0000 755c0000   SspiCli        (deferred)
755e0000 755f6000   win32u         (deferred)
75610000 75632000   GDI32          (deferred)
75640000 7579e000   gdi32full      (deferred)
75940000 7594e000   MSASN1         (deferred)
75950000 75975000   IMM32          (deferred)
759e0000 75a23000   sechost        (deferred)
75c50000 75ccc000   msvcp_win      (deferred)
75de0000 75e37000   bcryptPrimitives   (deferred)
```

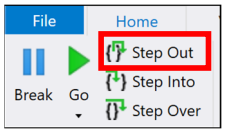
```
0:000> !dh -a 00400000

File Type: EXECUTABLE IMAGE
FILE HEADER VALUES
     14C machine (i386)
[redacted]

OPTIONAL HEADER VALUES
     10B magic #
    9.00 linker version
   11800 size of code
   1BA00 size of initialized data
       0 size of uninitialized data
    3228 address of entry point
    1000 base of code
         ----- new -----
00400000 image base
```


While not necessary for our current analysis, we can view loaded modules using the `lm` command. The first module listed represents the target executable. Note the module name DAZXCFGTYUNI—this is the hardcoded internal name of the program. Also note the reference to "exported symbols", which indicates that this EXE exports at least one function. This is unusual for an EXE, but we will not explore this observation in class.

With the base address specified in that output, we can use the `!dh` (display header) extension to display header details for the target executable.

Set a Breakpoint on the VirtualAlloc API to Locate Shellcode (1)

- We could set a breakpoint of the instruction *after* VirtualAlloc is called:
  - **`bp 0040166A`**
- Alternatively, we can set a breakpoint on the API code in the format `bp <module>!<API name>` and run until the function returns
  - **`bp KERNEL32!VirtualAllocStub`** OR **`bp KERNELBASE!VirtualAlloc`**
  - Kernel32.dll forwards VirtualAlloc calls to KernelBase.dll
  - **`gu`** OR click **Step Out** in the WinDbg Preview GUI to return



- After specifying the module name, you can use tab autocomplete

Our next step is to arrive at the shellcode within `WinDbg`. We know `VirtualAlloc` allocates space for the shellcode, so there are two approaches to consider:

- Set a breakpoint on the `VirtualAlloc` API and return from the function call.
- Set a breakpoint on the address of the instruction after `VirtualAlloc` is called (based on our earlier analysis).

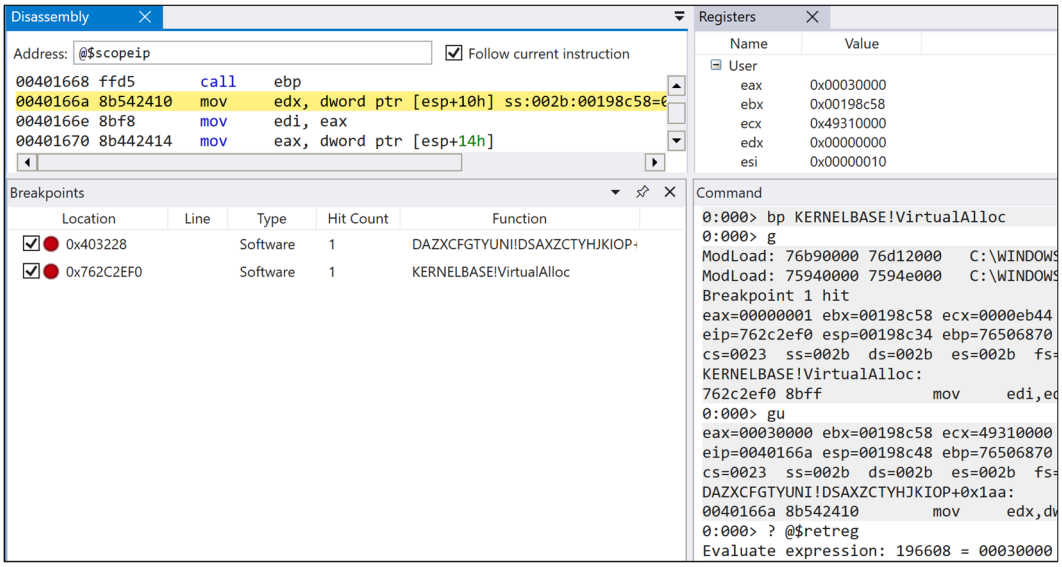
To set a breakpoint on `VirtualAlloc`, type the command bp `KERNEL32!VirtualAllocStub` or bp `KERNELBASE!VirtualAlloc`.

The above commands are appropriate because kernel32.dll actually forwards `VirtualAlloc` calls to kernelbase.dll.

When specifying the API name, tab autocomplete is helpful.

To set a breakpoint on the address after `VirtualAlloc` is called, type the command bp 0040166A.

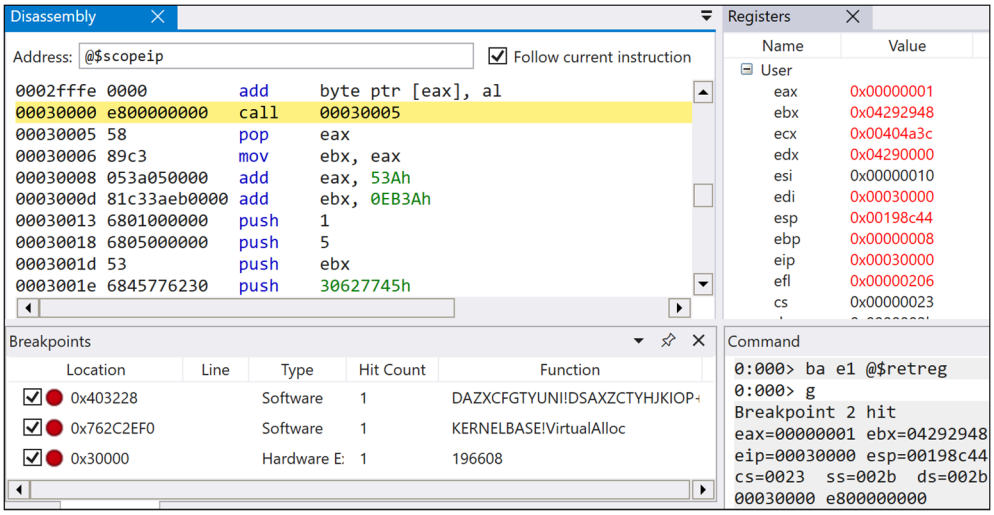
Set a Breakpoint on the VirtualAlloc API to Locate Shellcode (2)



This screenshot shows the results of running the commands on the previous slide. It includes the "User" section in the Registers view (expanded), and we can see the return value contained within EAX. This is the starting address of the allocated region.

We can also view the return address by typing the commands `? @$retreg` or `r eax`.

Set an Access Breakpoint to Pause at the Start of the Shellcode



To set an access breakpoint, type the command `ba e1 @$retreg`. This command sets an access breakpoint on the address stored in the return register (EAX). The "e" in `e1` specifies that the breakpoint should trigger on execution, and the numerical value specifies the size, in bytes, to monitor for access. As described in the Microsoft documentation for access breakpoints (https://for710.com/ba), the size must be 1 when the access breakpoint is set to trigger on execution.

Then, type `` `g` `` in the command window to continue executing the program and arrive at the breakpoint.

The debugger should pause at the beginning of the shellcode.

Recall That the Shellcode Passes Different Hexadecimal Values to the Same Function

- The function at offset 467 may use a hash to resolve API addresses
- Set a breakpoint with the command **`bp <function address>`**
- Type **`g`** to arrive in the function the first time it is called

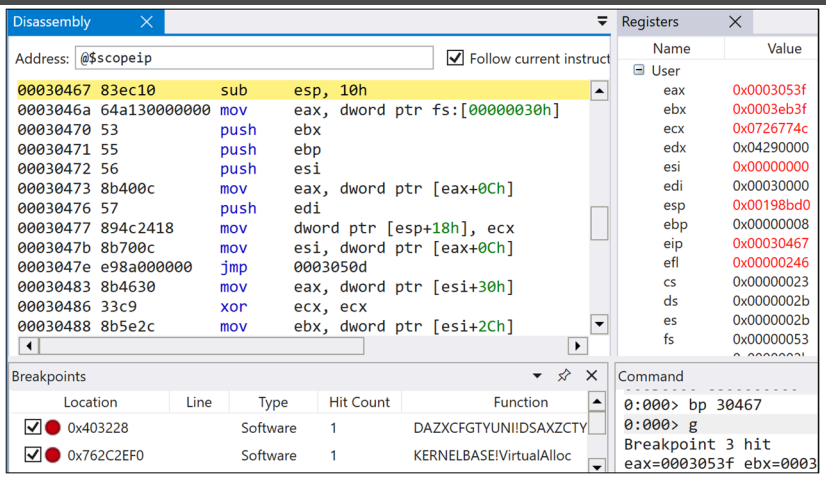
```
00030045 b949f70278   mov     ecx, 7802F749h
0003004a 8944241c     mov     dword ptr [esp+1Ch], eax
0003004e e814040000   call    00030467
00030053 b958a453e5   mov     ecx, 0E553A458h
00030058 89442420     mov     dword ptr [esp+20h], eax
0003005c e806040000   call    00030467
00030061 b910e18ac3   mov     ecx, 0C38AE110h
00030066 8be8         mov     ebp, eax
00030068 e8fa030000   call    00030467
```

Your virtual addresses may differ from those in the screenshots based on the starting address of the region allocated, so the slides will refer to *offsets* of specific instructions.

As a reminder, when we reviewed the shellcode in Ghidra, we observed different hex values passed to the same function. This is the function that contained a reference to the PEB. To arrive at the beginning of the function the first time it is called, we can set a breakpoint at the function address with the command `bp <function address>`. Note that the specific address to use will vary depending upon the memory region allocated in your debug session.

Then, type `g` to resume execution.

The Function Beginning at Offset 467 References FS:[30]

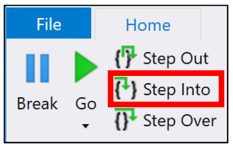


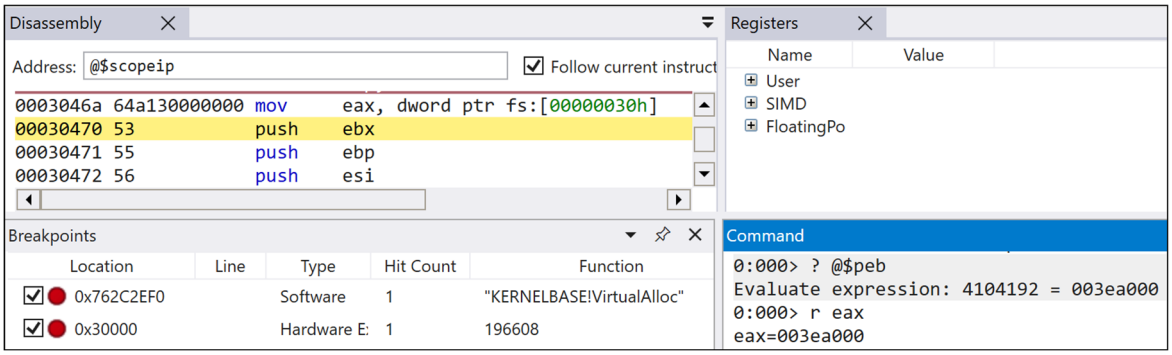
We can now view the Process Environment Block (PEB) and investigate other offsets within this structure.

This page intentionally left blank.

Execute the MOV Instruction at 46a and Confirm the Address of the PEB Matches the Value in EAX

- Click **Step Into** to execute individual instructions
- Execute the MOV at offset 46a to place fs:[30h] into eax
- The value in EAX matches the PEB's address

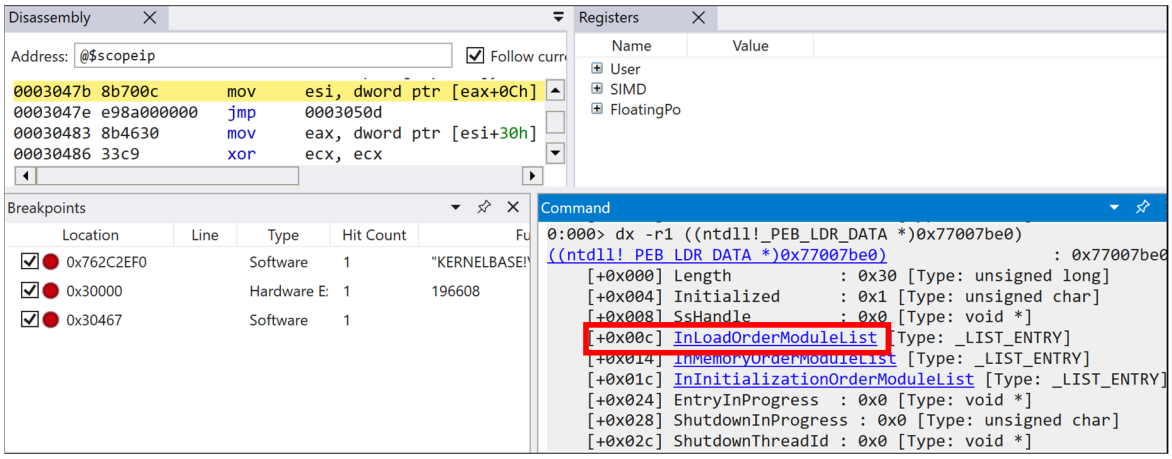




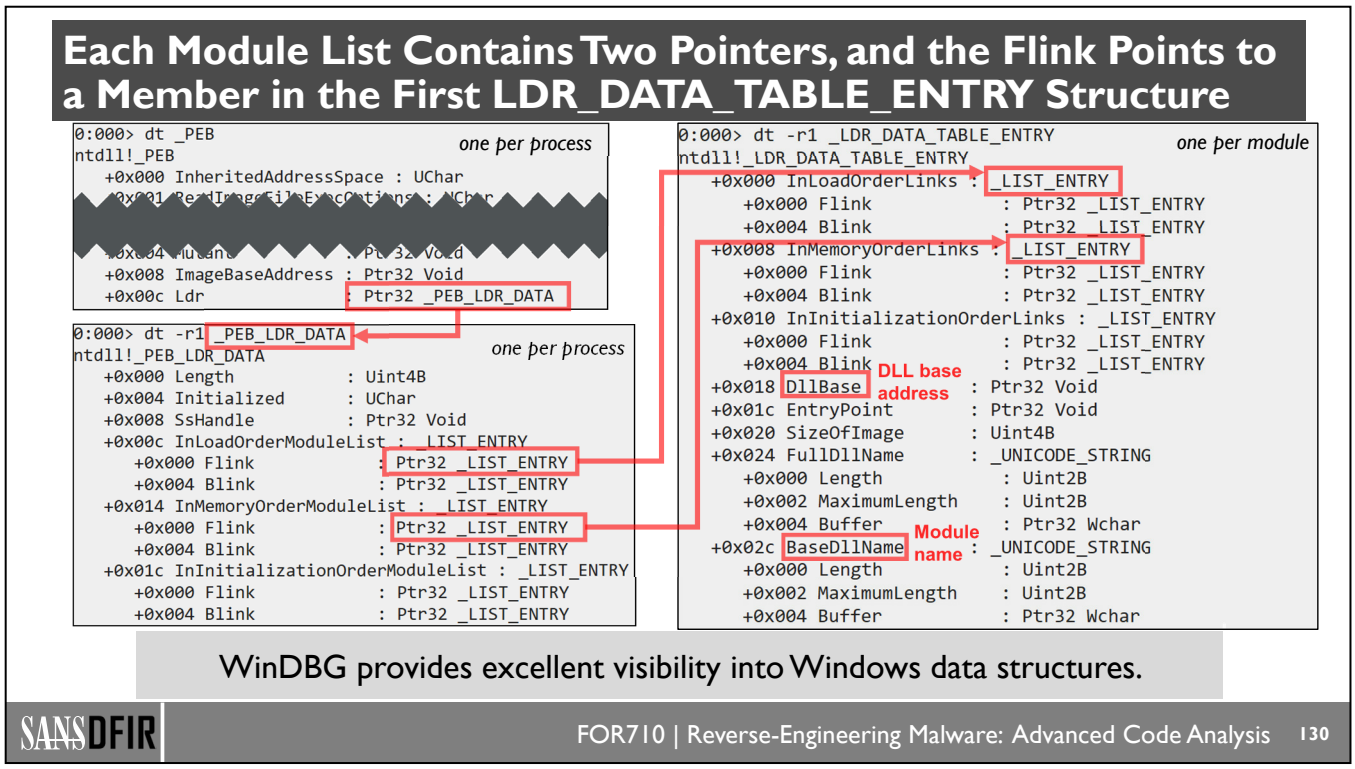
To confirm FS:[30] references the PEB, we can compare the contents of the `$peb` pseudo-register and the value of EAX after the instruction at offset 46a is executed.

At Offset 47b, the Second Operand Dereferences the InLoadOrderModuleList Member within the PEB_LDR_DATA

The InLoadOrderModuleList member is the head (i.e., start) of a doubly-linked list that describes the loaded modules for the process.



On this slide, the content in the command window was generated by clicking the "`Ldr`" link in the PEB structure shown on the previous slide.



The WinDbg screenshots on this slide show generic process structures that are not tied to a target process. The goal is to explain the relationship between a module list (e.g., InLoadOrderModuleList) and the LDR_DATA_TABLE_ENTRY structure, which contains information about a single module loaded into a process.

All module list members in the PEB contain the head (i.e., start) of a doubly linked list that tracks loaded modules:

InLoadOrderModuleList: List of modules in the order in which they were loaded.
InMemoryOrderModuleList: List of modules in the order in which they were placed in memory.
InInitializationOrderModuleList: List of modules in the order in which they were initialized.

The differences between these lists are irrelevant to our analysis. We must simply understand that each list tracks loaded modules. Shellcode will typically choose *one* list to traverse and identify the file names of loaded modules. These file names are then hashed as one part of an API hashing algorithm.

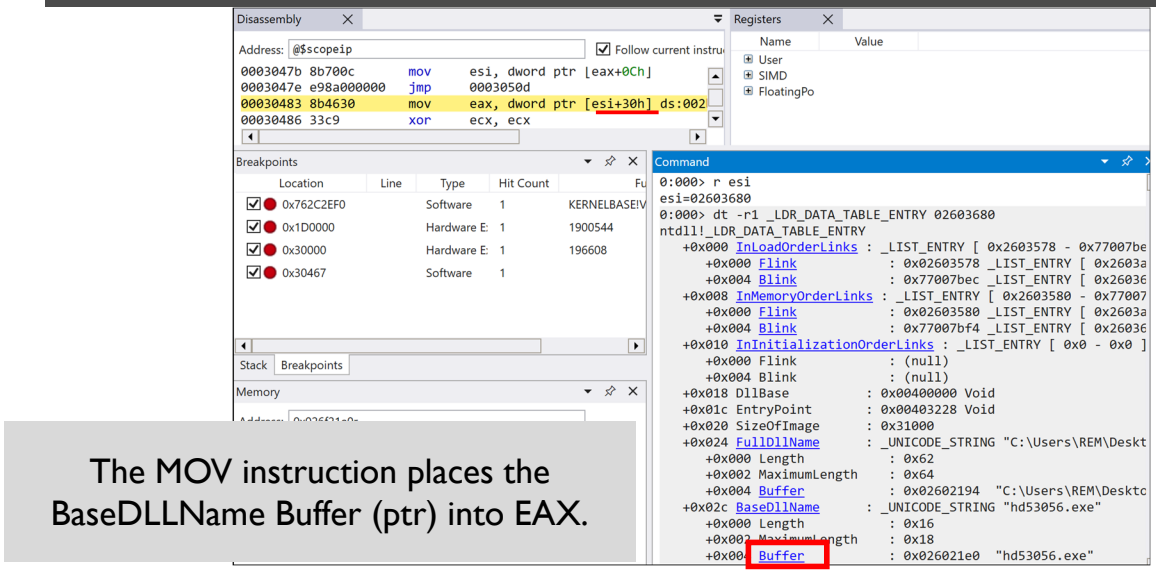
The example on this slide focuses on the InLoadOrderModuleList member, but the details that follow also apply to the other two double-linked lists.

InLoadOrderModuleList is a structure of type LIST_ENTRY. In the graphic on the bottom-left, observe that the LIST_ENTRY structure contains two members—a forward pointer to the *next* LIST_ENTRY structure (i.e., Flink) and a backward pointer to the *previous* LIST_ENTRY structure (i.e., Blink). As an example, we will follow the Flink. The Flink points to another LIST_ENTRY structure contained within an LDR_DATA_TABLE_ENTRY structure. The LDR_DATA_TABLE_ENTRY structure contains information about one loaded DLL, and each DLL tracked by InLoadOrderModuleList has an LDR_DATA_TABLE_ENTRY structure associated with it. Among its many members is the DllBase, which specifies the base address in memory of the module. The base address is often used by shellcode as a starting point to find the export directory. Also worth noting is the BaseDllName, which specifies the name of the loaded DLL. This member is typically accessed by shellcode to acquire the address of the Buffer that stores a DLL name. The DLL name is often combined with a function name to perform API hashing.

As we discussed earlier, the dt command displays information about the specified data type. If you query information about a specific process's PEB, the format of the command is dt _PEB <PEB address>. If we want to view generic information about the PEB (i.e., not specific to a process) we simply omit the last parameter. In fact, we can obtain generic information about many Windows structures using the command format dt <structure>. On this slide, we use the generic form of this command to query information about the PEB and its members.

Note that slide describes 32-bit structures, but the same commands will produce the 64-bit structures when a 64-bit process is loaded into `WinDbg` Preview. Also, all structures on this slide have additional members not listed. For a full list of structure members, type the specified commands in `WinDbg` Preview. You can also retrieve more information for each structure by searching for the structure name at https://www.aldeid.com/wiki/.

At Offset 483, ESI Contains the Address of the InLoadOrderLinks Member Within an LDR_DATA_TABLE_ENTRY Structure

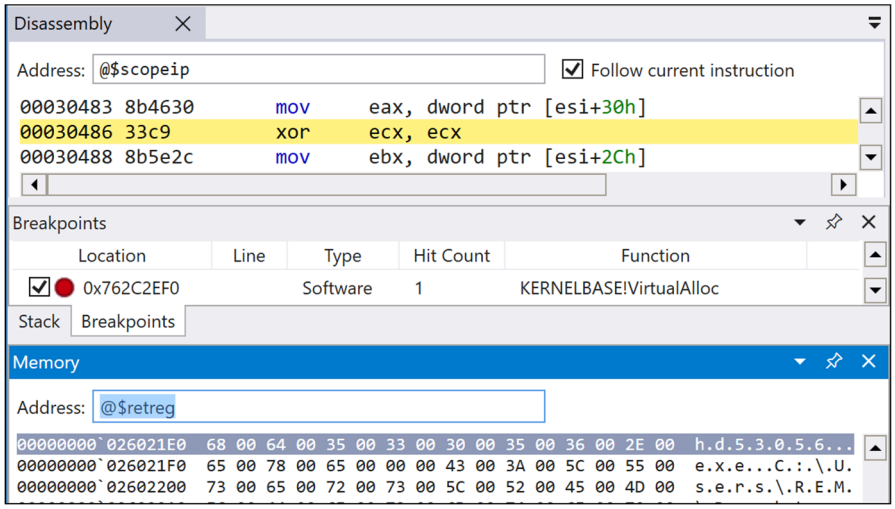


The MOV instruction places the BaseDLLName Buffer (ptr) into EAX.

Executing the MOV instruction at offset 483 places a pointer to the module name into EAX.

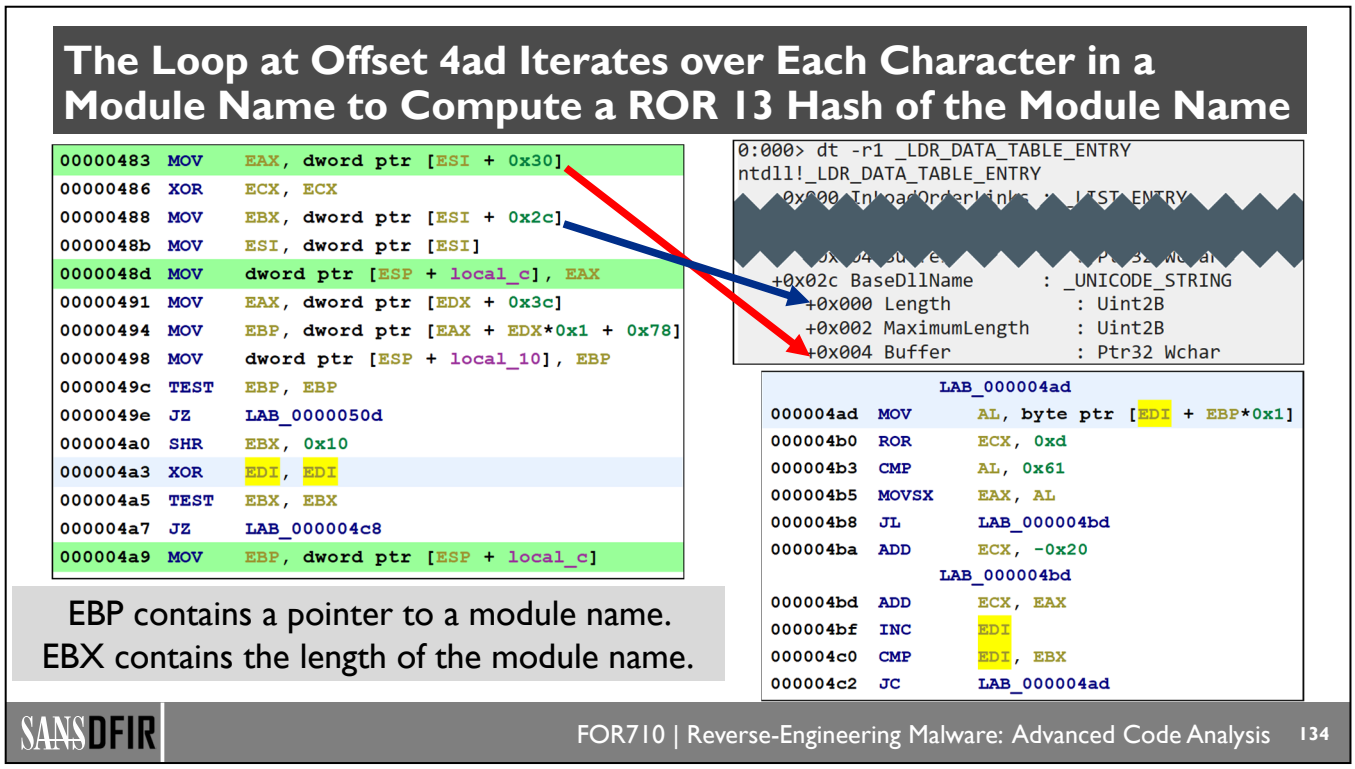
Observe that offset 0x30 from the InLoadOrderLinks member is within the BaseDllName member—specifically, it brings us to the Buffer component of the BaseDllName, which specifies the pointer to a module name.

After Executing the MOV at Offset 483, Confirm EAX Contains a Pointer to a Module Name Using the Memory View



Print the contents of EAX with the command `r eax`. Then, copy and paste the resulting address to the Address input field in the Memory window and press Enter on the keyboard. Alternatively, you can just type `@$retreg` in the Address input field and press Enter on the keyboard.

The hexdump should show a loaded module name in the ascii representation of the displayed content. The first time the MOV instruction is executed, EAX will contain a pointer to the target executable.



`WinDbg` is great for probing the PEB and related structures, but for strict static code analysis, Ghidra is our primary interface. Ghidra excels at showing the flow of execution (e.g., visual arrows to identify jumps and loops) and is optimal for entering comments and renaming functions.

As described on the slide, the loop beginning at offset 4ad generates a ROR13 hash of a module name.

After the MOV Instruction at Offset 494 is Executed, EBP Contains the VA of the Module's Export Directory

```
0000047b    MOV     ESI, dword ptr [EAX + 0xc]
0000047e    JMP     LAB_0000050d
```

The address of InLoadOrderLinks within LDR_DATA_TABLE_ENTRY is placed into ESI.

```
0000050d    MOV     EDX, dword ptr [ESI + 0x18]
00000510    TEST    EDX, EDX
00000512    JNZ     LAB_00000483
```

The module's base address (DllBase member within LDR_DATA_TABLE_ENTRY) is placed into EDX.

```
                    LAB_00000483                                XRE
00000483    MOV     EAX, dword ptr [ESI + 0x30]
                                  ...
0000048d    MOV     dword ptr [ESP + local_c], EAX
00000491    MOV     EAX, dword ptr [EDX + 0x3c]
00000494    MOV     EBP, dword ptr [EAX + EDX*0x1 + 0x78]
```

At offset 0x3c from the base address is the e_lfanew field, which specifies the RVA of the PE header.

At offset 0x78 from the PE header is the Export Directory RVA (for 32-bit executables).

As described in the text on this slide, the instructions shown obtain access to the address of the module's export directory. Recall that as part of the API hashing process, shellcode typically iterates through a module's exported functions and hashes each function name.

The Instruction at Offset 4e0 Places the Virtual Address (VA) of an Exported Function Name (i.e., ptr to a string) into EBP

```
000004c8  MOV   EAX, dword ptr [EDX + EBP*0x1 + 0x20]
000004cc  XOR   EBX, EBX
000004ce  MOV   EDI, dword ptr [EDX + EBP*0x1 + 0x18]
000004d2  ADD   EAX, EDX
000004d4  MOV   dword ptr [ESP + local_c], EDI
000004d8  TEST  EDI, EDI
000004da  JZ    LAB_0000050d

              LAB_000004dc                                  XR
000004dc  MOV   EBP, dword ptr [EAX]
000004de  XOR   EDI, EDI
000004e0  ADD   EBP, EDX
```

```
IMAGE_EXPORT_DIRECTORY
+0x000 Characteristics
+0x004 TimeDateStamp
+0x008 MajorVersion
+0x00a MinorVersion
+0x00c Name
+0x010 Base
+0x014 NumberOfFunctions
+0x018 NumberOfNames
+0x01c AddressOfFunctions
+0x020 AddressOfNames
+0x024 AddressOfNameOrdinals
```

Export Directory Table → Name Pointer Table → Export Name Table → Export Ordinal Table → Export Address Table

At offset 4c8, EDX contains the VA of a module and EBP contains the RVA of the module's export directory—adding these values yields the VA of the export directory. As a reminder, the structure of a 32-bit export directory is shown on this slide. At offset 0x20 within the export directory is the `AddressOfNames` member, which contains the RVA of the name pointer table. The name pointer table contains RVAs into the export name table, which contains the string names of exported functions. Therefore, the instruction at offset 4c8 places the Name Pointer Table RVA into EAX.

At offset 4d2, the image base (EDX) is added to the RVA of the name pointer table to place the VA of the name pointer table into EAX.

At offset 4dc, the first RVA within the name pointer table is placed into EBP. At offset 4e0, the VA of a string within the export name table is placed into EBP.

The text on the very bottom of this slide serves as a reminder of the process for resolving a function via the export directory table.

If the Calculated API Hash Matches the Argument, the Address is Resolved and Placed into EAX

```
                    LAB_00000522                                    XRE
00000522    MOV     ESI, dword ptr [ESP + local_10]
00000526    MOV     EAX, dword ptr [ESI + EDX*0x1 + 0x24]
0000052a    LEA     EAX, [EAX + EBX*0x2]
0000052d    MOVZX   ECX, word ptr [EAX + EDX*0x1]
00000531    MOV     EAX, dword ptr [ESI + EDX*0x1 + 0x1c]
00000535    LEA     EAX, [EAX + ECX*0x4]
00000538    MOV     EAX, dword ptr [EAX + EDX*0x1]
0000053b    ADD     EAX, EDX
0000053d    JMP     LAB_0000051a
```

```
0000051a    POP     EDI
0000051b    POP     ESI
0000051c    POP     EBP
0000051d    POP     EBX
0000051e    ADD     ESP, 0x10
00000521    RET
```

```
IMAGE_EXPORT_DIRECTORY
+0x000 Characteristics
+0x004 TimeDateStamp
+0x008 MajorVersion
+0x00a MinorVersion
+0x00c Name
+0x010 Base
+0x014 NumberOfFunctions
+0x018 NumberOfNames
+0x01c AddressOfFunctions
+0x020 AddressOfNames
+0x024 AddressOfNameOrdinals
```

```
00000084    CALL    resolve_api
00000089    MOV     EBX, EAX
```

```
000000eb    CALL    EBX
```

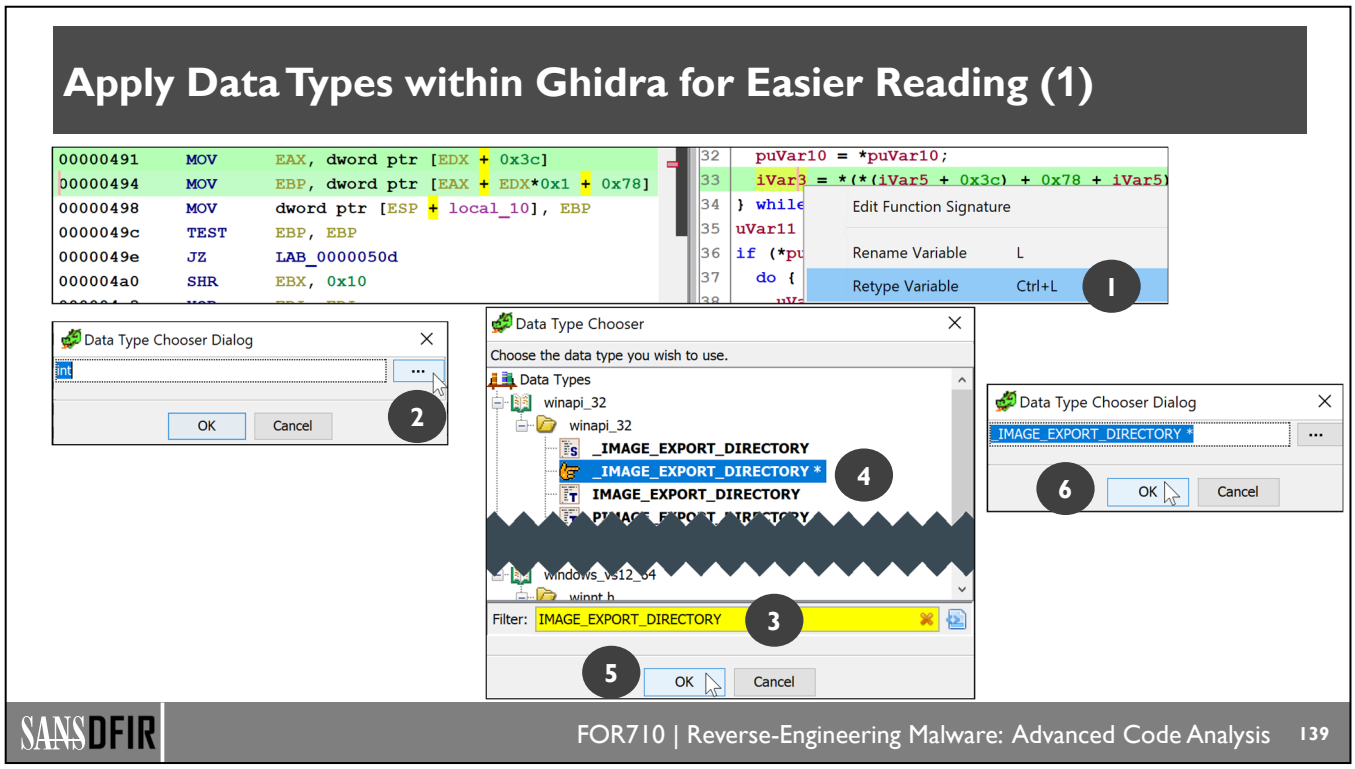
Export Directory Table → Name Pointer Table → Export Name Table → Export Ordinal Table → Export Address Table

If the calculated API hash matches the one passed as an argument, then the shellcode resolves the appropriate function address in preparation for executing the function. While we will not discuss each remaining instruction in detail, observe that the instruction at offset 526 accesses the `AddressOfNameOrdinals` member, located at offset 0x24 within the export directory. Dereferencing this field provides access to the export ordinal table. As we discussed earlier in Section 1, this is one of the final steps to acquire a function's address.

Before the function exits, the code places a resolved function's address in EAX.

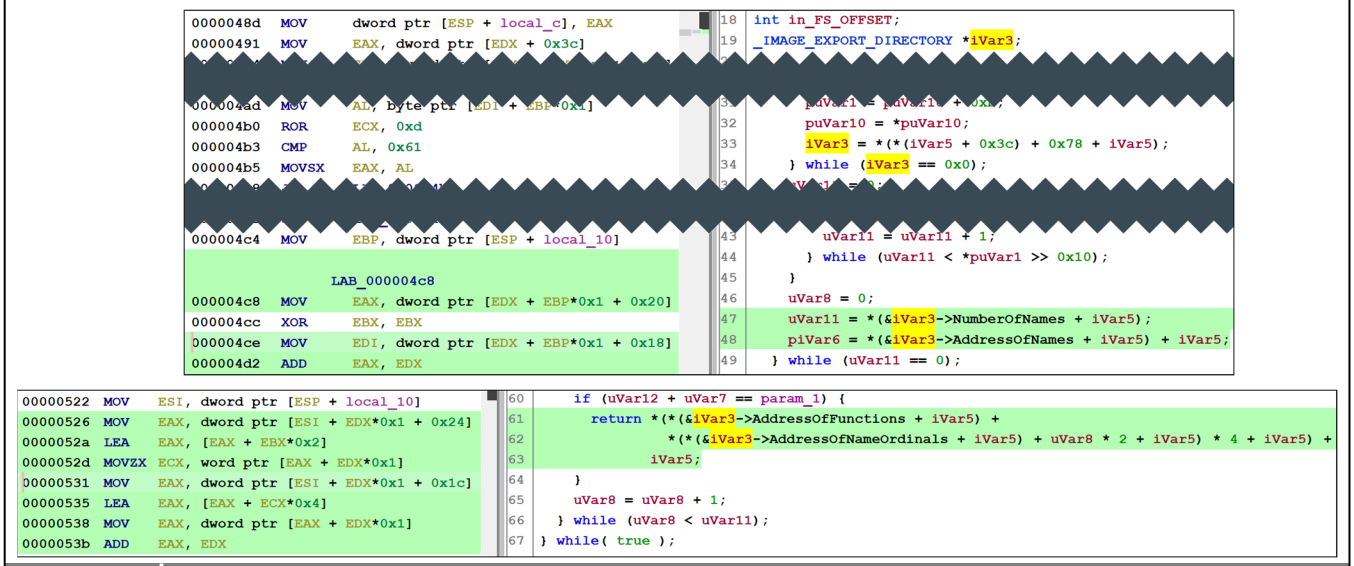
As shown in the code excepts on the bottom-right of this slide, we can rename the function that resolves APIs to `resolve_api`. When a function address is returned, the shellcode eventually executes the function as needed.

The text on the very bottom of this slide serves as a reminder of the process for resolving a function via the export directory table.



With an understanding of the Windows and PE file data structures referenced in this code, we can apply the relevant data types in the Decompiler output. Apply data types by following the steps on this slide. In this example, we apply a pointer to the IMAGE_EXPORT_DIRECTORY structure to a variable in our decompiler output. Note that if the structure of interest is not an argument or assigned to a variable, we may not be able to assign the relevant data type.

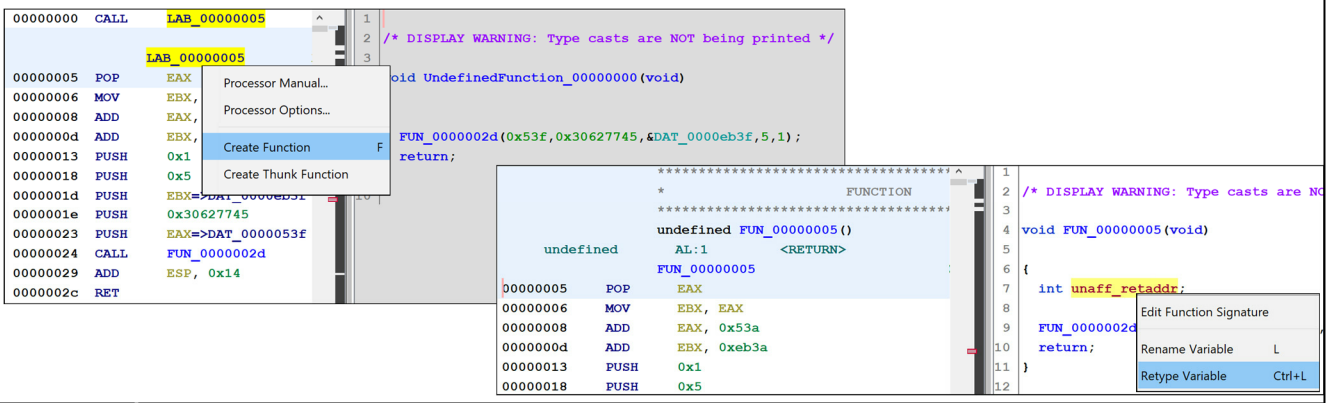
Apply Data Types within Ghidra for Easier Reading (2)



After applying the IMAGE_EXPORT_DIRECTORY pointer data type, the decompiler output updates to reflect this structure's members.

Apply Data Types within Ghidra for Easier Reading (3)

- Functions must be fully defined to rename and retype arguments and variables.
- A gray Decompiler background indicates a function is not fully defined.



If you encounter a scenario where Ghidra does not give you an option to retype or rename an argument or variable, it may be because the function in question is not fully defined. Another indication of this is a grayed out Decompiler background. To define a function, right click at the beginning of the function in the Listing view and choose Create Function (or type the ”F” key).

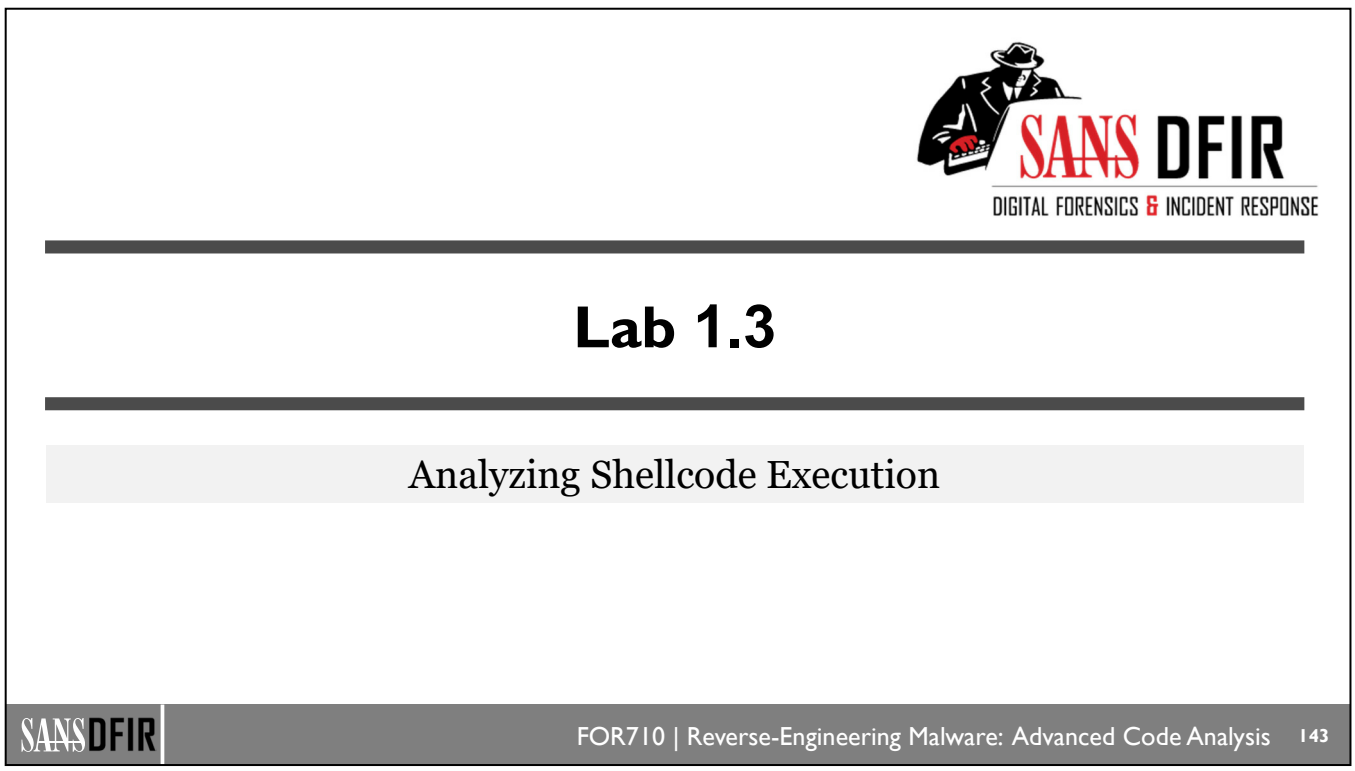
Course Roadmap

- FOR710.1: Code Deobfuscation and Execution
- FOR710.2: Encryption in Malware
- FOR710.3: Automating Malware Analysis
- FOR710.4: Correlating Malware and Building Rules

SECTION I

- Analyzing Code Deobfuscation
  - Lab 1.1: Investigating Code Deobfuscation Using Steganographic Techniques
- Identifying Program Execution
  - Lab 1.2: Analyzing Malicious Program Execution
- Understanding Shellcode Execution
  - ***Lab 1.3: Analyzing Shellcode Execution***

This page intentionally left blank.



Please begin Lab 1.3 now.

Analyzing Shellcode Execution: Module Objectives, Revisited

- ✓Identify and extract shellcode during program execution.
- ✓Understand how shellcode uses hashing algorithms to resolve APIs.
- ✓Understand the Process Environment Block (PEB) and its components.
- ✓Gain familiarity with WinDBG for debugging.
- ✓Use WinDbg to explore the PEB and related data structures.
- ✓Apply an analysis workflow that involves Ghidra, x32dbg, and WinDbg.

This page intentionally left blank.

Course Roadmap

- FOR710.1: Code Deobfuscation and Execution
- FOR710.2: Encryption in Malware
- FOR710.3: Automating Malware Analysis
- FOR710.4: Correlating Malware and Building Rules

SECTION I

- Analyzing Code Deobfuscation
  - Lab 1.1: Investigating Code Deobfuscation Using Steganographic Techniques
- Identifying Program Execution
  - Lab 1.2: Analyzing Malicious Program Execution
- Understanding Shellcode Execution
  - Lab 1.3: Analyzing Shellcode Execution

This page intentionally left blank.