

710.4

Correlating Malware and Building Rules



PLEASE READ THE TERMS AND CONDITIONS OF THIS COURSEWARE LICENSE AGREEMENT ("CLA") CAREFULLY BEFORE USING ANY OF THE COURSEWARE ASSOCIATED WITH THE SANS COURSE. THIS IS A LEGAL AND ENFORCEABLE CONTRACT BETWEEN YOU (THE "USER") AND SANS INSTITUTE FOR THE COURSEWARE. YOU AGREE THAT THIS AGREEMENT IS ENFORCEABLE LIKE ANY WRITTEN NEGOTIATED AGREEMENT SIGNED BY YOU.

With this CLA, SANS Institute hereby grants User a personal, non-exclusive license to use the Courseware subject to the terms of this agreement. Courseware includes all printed materials, including course books and lab workbooks, as well as any digital or other media, virtual machines, and/or data sets distributed by SANS Institute to User for use in the SANS class associated with the Courseware. User agrees that the CLA is the complete and exclusive statement of agreement between SANS Institute and you and that this CLA supersedes any oral or written proposal, agreement or other communication relating to the subject matter of this CLA.

BY ACCEPTING THIS COURSEWARE, USER AGREES TO BE BOUND BY THE TERMS OF THIS CLA. BY ACCEPTING THIS SOFTWARE, USER AGREES THAT ANY BREACH OF THE TERMS OF THIS CLA MAY CAUSE IRREPARABLE HARM AND SIGNIFICANT INJURY TO SANS INSTITUTE, AND THAT SANS INSTITUTE MAY ENFORCE THESE PROVISIONS BY INJUNCTION (WITHOUT THE NECESSITY OF POSTING BOND) SPECIFIC PERFORMANCE, OR OTHER EQUITABLE RELIEF.

If User does not agree, User may return the Courseware to SANS Institute for a full refund, if applicable.

User may not copy, reproduce, re-publish, distribute, display, modify or create derivative works based upon all or any portion of the Courseware, in any medium whether printed, electronic or otherwise, for any purpose, without the express prior written consent of SANS Institute. Additionally, User may not sell, rent, lease, trade, or otherwise transfer the Courseware in any way, shape, or form without the express written consent of SANS Institute.

If any provision of this CLA is declared unenforceable in any jurisdiction, then such provision shall be deemed to be severable from this CLA and shall not affect the remainder thereof. An amendment or addendum to this CLA may accompany this Courseware.

SANS acknowledges that any and all software and/or tools, graphics, images, tables, charts or graphs presented in this Courseware are the sole property of their respective trademark/registered/copyright owners, including:

AirDrop, AirPort, AirPort Time Capsule, Apple, Apple Remote Desktop, Apple TV, App Nap, Back to My Mac, Boot Camp, Cocoa, FaceTime, FileVault, Finder, FireWire, FireWire logo, iCal, iChat, iLife, iMac, iMessage, iPad, iPad Air, iPad Mini, iPhone, iPhoto, iPod, iPod classic, iPod shuffle, iPod nano, iPod touch, iTunes, iTunes logo, iWork, Keychain, Keynote, Mac, Mac Logo, MacBook, MacBook Air, MacBook Pro, Macintosh, Mac OS, Mac Pro, Numbers, OS X, Pages, Passbook, Retina, Safari, Siri, Spaces, Spotlight, There's an app for that, Time Capsule, Time Machine, Touch ID, Xcode, Xserve, App Store, and iCloud are registered trademarks of Apple Inc.

PMP® and PMBOK® are registered trademarks of PMI.

SOF-ELK® is a registered trademark of Lewes Technology Consulting, LLC. Used with permission.

SIFT® is a registered trademark of Harbingers, LLC. Used with permission.

Governing Law: This Agreement shall be governed by the laws of the State of Maryland, USA.

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



Correlating Malware and Building Rules

© 2022 Anuj Soni | All Rights Reserved | Version H02_05

Section FOR710.4, also known as Section 4 of the FOR710 course, explores approaches to correlating malware and writing rules to expedite future analysis.

FOR710.4 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>.

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 4

- ***Correlating Malware***
 - Lab 4.1: Correlating Malware
- Building YARA Rules
 - Lab 4.2: Writing YARA Rules
- Building capa Rules
 - Lab 4.3: Writing capa Rules
- Advanced Malware Analysis Tournament

This page intentionally left blank.

Correlating Malware

This page intentionally left blank.

Correlating Malware: Module Objectives

- Understand the benefits of comparing malware samples.
- Use Python scripts to identify similarities and differences in malware.
- Apply BinDiff to compare functions, identify changes to function code, and highlight added/removed functions.
- Use Ghidra's Program Diff feature to compare similar programs at the instruction level.

This slide describes the objectives of this module.

Benefits of Correlation Analysis

- Identify similarities and differences between programs to find malware variants and new or removed features.
- Pinpoint similar code and data across samples to identify relationships.
- Recognize code reuse to reduce time spent performing code analysis.
- Use identified similarities and differences to build effective YARA rules.
- Outputs from correlation analysis help build threat intelligence.

Understanding how to compare malware samples is important for several reasons. For example, you can expedite your analysis if you identify code reuse between new malware and malware you reviewed in the past. Correlation analysis might also lead to the discovery of new variants of a malware family. In addition, the successful identification of similarities and differences across malware samples can help build effective YARA rules, a topic we will discuss in the next module.

Effective Approaches to Compare Malware

- Mathematical calculations:

- File and section hashes
- Fuzzy hashes (i.e., ssdeep)
- Import Table Hash (i.e., imphash)
- Mathematical models

In general, any output of static file analysis, behavioral analysis, or code analysis can be used for comparison.

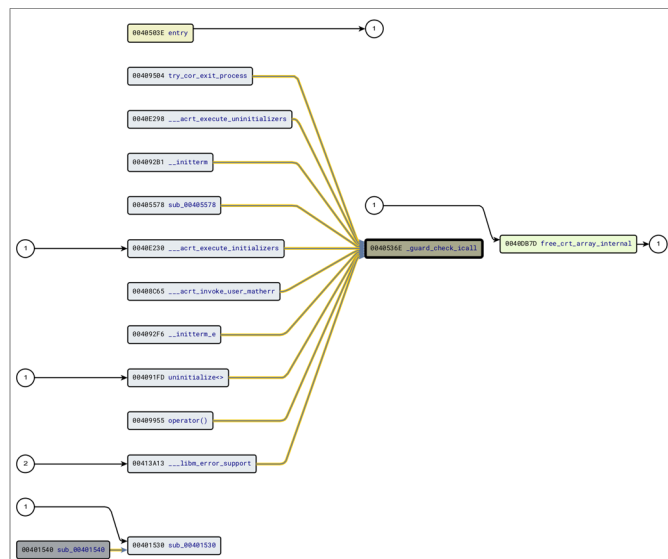
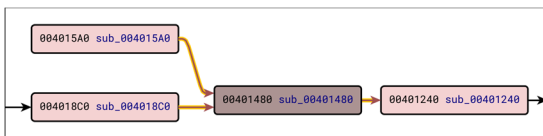
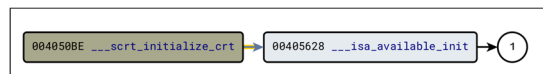
- Data: embedded strings and APIs

- Code:

- Call graphs: Relationship of all calls to one another
- Control flow graphs: Basic blocks and the links between them
- Basic block instructions

There are a variety of approaches to consider when comparing programs. In general, we could use any output from our static file analysis, behavioral analysis, or code analysis as a data point for comparison. However, this module will focus on static approaches for file comparison. For example, calculations such as file, section, and fuzzy hashes are a relatively simple and quick output to use for comparison. Another approach is to compare embedded strings and imports—these are characteristics we often discover during static file analysis. Finally, we could perform a more detailed comparison by looking at each program's code. Thankfully, there are freely available tools like BinDiff that can help with this process. BinDiff reviews call graphs, control flow graphs, and the contents of basic blocks.

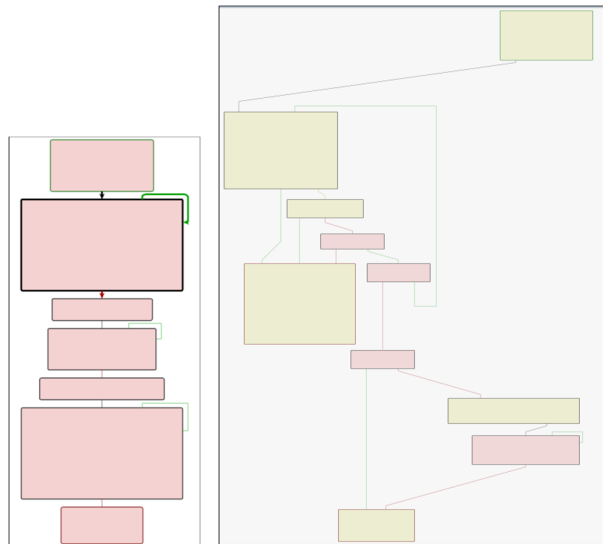
Call Graphs Describe the Relationship between Functions



Call graphs are an abstraction of a program that represents the relationship between function calls within a program. Comparing call graphs is one way to assess the similarities and differences between programs. This type of analysis is difficult to do manually, so we rely on programs like BinDiff to perform this sort of analysis.

Control Flow Graphs (CFG) Depict the Flow of Execution

- Each node is a basic block.
- A basic block is a sequence of instructions that does not include a jump.
- Each edge is a jump.
- Comparing CFGs is another way to compare programs.



A control flow graph (CFG) describes the overall flow of execution. Examples of CFGs are shown on this slide. In a CFG, each node is a basic block, and each edge represents a control transfer between blocks.

Tools for Correlation Analysis

- pestats.py: Basic static file information
- pecompare.py: Strings and imports
- BinDiff: Detailed function-level differences
- Ghidra's Program Diff feature: Instruction level differences

This slide lists the key tools we will use to perform correlation analysis. In the upcoming slides, we will discuss each capability in more detail.

Use peststats.py to Generate Information about Target Files

```
C:\Users\REM\Desktop\Malware\Section4>python peststats.py
Usage: python peststats <file|directory>
```

```
C:\Users\REM\Desktop\Malware\Section4>python peststats.py destiny
```

File name	Sha-256 Hash	Architecture	Size(bytes)	Compile Time	Imphash	Section Hashes
destiny\04f7	04F76D44DB4C3A8D810348F65E5...	32-Bit	130048	Fri Jun 18 20:16:46 2021 UTC	AA4E67D3CC88...	.text:638F5F1CFE6E68EED559F01471455B94, .rdata:768D0348CA59
destiny\124f	124F3A5CAF6EB464027F2865225...	32-Bit	129992	Sat Jun 19 20:48:52 2021 UTC	36231B97F6B1...	.text:D78EE82AF5B7217FDC00970D41012BE7, .rdata:3090CE8201FF
destiny\1941	19417C0A38A1206007A0CC82C0F...	32-Bit	121856	Sat Jun 19 20:55:23 2021 UTC	36231B97F6B1...	.text:BD1AF2DD84DB2EB5C3D913E332782877, .rdata:20A6A20DA552
destiny\ae39	A63937D94B4D0576C083398497F...	32-Bit	16896	Mon Jun 21 18:50:53 2021 UTC	716A67383034...	.text:3E886E1087137AA52B65F17CAED99339, .rdata:B974E73C6EE7
destiny\ad84	AD841882052C3F9D856AD9A3932...	32-Bit	16896	Mon Jun 21 18:54:13 2021 UTC	716A67383034...	.text:3E886E1087137AA52B65F17CAED99339, .rdata:7C18A67466F8
destiny\3462	34629751D8202BE456DCF149B51...	32-Bit	18944	Thu Jun 24 00:08:43 2021 UTC	600EF9C591D4...	.text:6F167AEAG0BDFBDFE9BA79B55ABFD91A, .rdata:B02E80DCD2A6
destiny\0d03	0D037EE0252E4F26800BCF7C750...	32-Bit	18944	Thu Jun 24 00:10:43 2021 UTC	600EF9C591D4...	.text:268D9E581BDB3B03DB3E806964020C5F, .rdata:6F32F85F5A3C
destiny\3ff1	3FF1B900B8AD5D78397FDC731C3A...	32-Bit	18944	Fri Jun 25 16:59:29 2021 UTC	600EF9C591D4...	.text:E6E40445F0275DD4859F88861DFA1051, .rdata:995526CC244D
destiny\6c98	6C98D424AB1B9BFA683EDA340F...	32-Bit	17920	Tue Sep 21 03:13:50 2021 UTC	DA687DAE3534...	.text:E5E78D7352CCBA0D501A594F96A59A3E, .rdata:B5402E3C43EE
destiny\1c41	1C41ACDC2E9D8B89522EBB51D65...	32-Bit	17920	Tue Sep 21 03:16:56 2021 UTC	DA687DAE3534...	.text:C0600A50195160CEE353FD1A3FEEA133, .rdata:A42E5A495014
destiny\84d2	84D24A16949B5A89162411AB98A...	32-Bit	17920	Tue Sep 21 03:16:56 2021 UTC	DA687DAE3534...	.text:C0600A50195160CEE353FD1A3FEEA133, .rdata:A42E5A495014

Pestats.py is a Python script developed by Anuj Soni. You can find it inside Malware\Section4\scripts.zip. It is a simple script that collects basic information about an individual file or group of files and writes them to a CSV file. Data collected includes the columns shown on this slide. The purpose of this script is to quickly compare files at a high level. This helps direct more detailed correlation analysis efforts.

To view the generated CSV file, try TimeLine Explorer. A shortcut to this program is located on your VM desktop. This tool allows you easily view, sort, and filter CSV files.

Use pecompare.py to Compare String and Imported Functions

```
C:\Users\REM\Desktop\Malware\Section4>python pecompare.py destiny\1941 destiny\639
File #1: destiny\1941
# Imports: 83
# Strings: 831

File #2: destiny\639
# Imports: 29
# Strings: 81

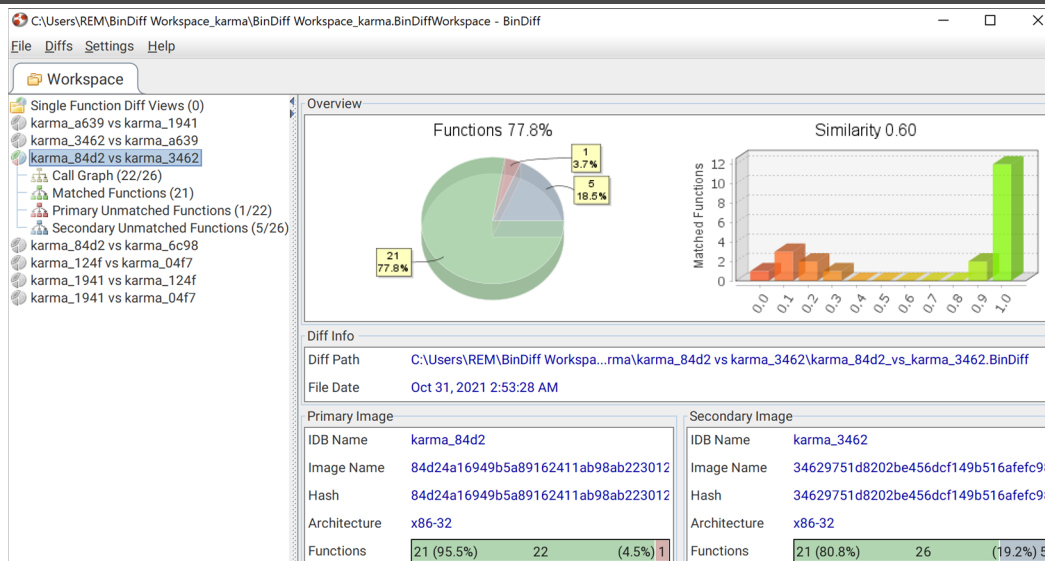
# Strings in common:40
=====
!This program cannot be run in DOS mode.
.idata$2
.idata$3
.....
.....
.....
FindFirstFileW
FindNextFileW
GetCommandLineW
GetDriveTypeW
GetFileSizeEx
GetLastError
GetProcAddress
GetProcessHeap
GetStdHandle
HeapAlloc
HeapFree
KARMA
KERNEL32.dll
LoadLibraryA
MoveFileW
```

```
Strings ONLY in File #1: 791
=====
((((((H
<requestedExecutionLevel level='asInvoker' uiAccess='false' />
</requestedPrivileges>
<requestedPrivileges>
</security>
<security>
</trustInfo>
<trustInfo xmlns="urn:schemas-microsoft-com:asm.v3">
!#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNopqrstuvwxyz[\]^_`ABCDEF
!#$%&'()*+,-./0123456789:;<=>?@abcdefghijklmnopqrstuvwxyz[\]^_`abcde
(2)\cypher_wc\Release\cypher.pdb
Base Class Array'
Base Class Descriptor at (
```

```
Strings ONLY in File #2: 41
=====
[+] Argument:
;3;!X
$recycle.bin
-ENCRYPTED.txt
11111111
<MQQ@
<MQQ@
@[+] Checking if already started...
BCryptGenRandom
C$~oq6
CreateMutexA
CryptStringToBinaryW
D$4Ph
```

Pecompare.py is a Python script that compares two files. Specifically, this script compares embedded strings and imported functions. You can find this script inside Malware\Section4\scripts.zip.

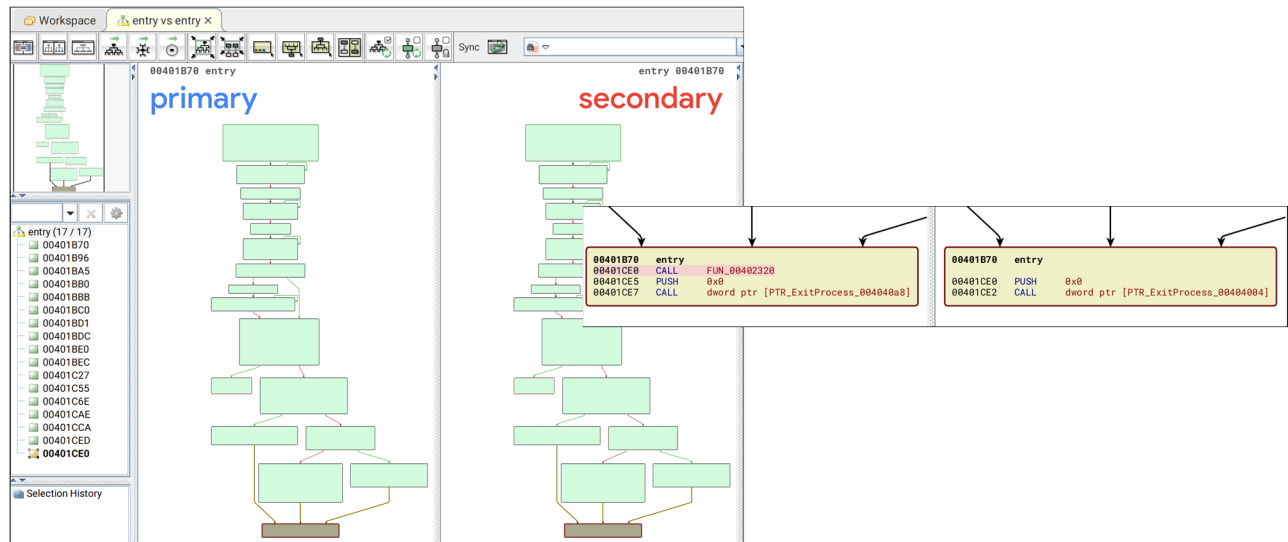
BinDiff: Function-Level Differences



BinDiff is a powerful tool for understanding function-level differences. It provides a percentage value that describes how similar functions are between two samples and allows the analyst to compare functions side-by-side. The upcoming exercise includes step-by-step instructions for generating output like what you see on this slide.

For more details on how BinDiff matches functions, browse to <https://for710.com/bindiff-matching>.

BinDiff: Matched Function Details



Clicking on a matched function provides a comparison of basic blocks, and we can zoom in for additional detail.

BinDiff: Unmatched Functions

Workspace

Single Function Diff Views (0)

karma_a639 vs karma_1941

karma_3462 vs karma_a639

karma_84d2 vs karma_3462

Call Graph (22/26)

Matched Functions (21)

Primary Unmatched Functions (1/22)

Secondary Unmatched Functions (5/26)

karma_84d2 vs karma_6c98

karma_124f vs karma_04f7

karma_1941 vs karma_124f

karma_1941 vs karma_04f7

1 / 1 Primary Unmatched Functions

Address	Name	Type	Basic Blocks	Jumps	Instructions	Callers	Callees
004038B0	__alloca_probe	Normal	4	4	4	1	0

Workspace

Single Function Diff Views (0)

karma_a639 vs karma_1941

karma_3462 vs karma_a639

karma_84d2 vs karma_3462

Call Graph (22/26)

Matched Functions (21)

Primary Unmatched Functions (1/22)

Secondary Unmatched Functions (5/26)

karma_84d2 vs karma_6c98

karma_124f vs karma_04f7

5 / 5 Secondary Unmatched Functions

Address	Name	Type	Basic Blocks	Jumps	Instructions	Callers	Callees
004021B0	sub_004021B0	Normal	15	22	15	0	0
00402760	sub_00402760	Normal	68	113	68	0	1
00402D30	sub_00402D30	Normal	140	214	140	2	1
004035A0	lpStartAddress_004035a0	Normal	1	0	1	0	1
004035D0	sub_004035D0	Normal	10	14	10	0	0

View the “Unmatched Functions” views to display functions which appear in only one program.

Correlation Analysis Challenges

- Samples that are packed or obfuscated may have very different file characteristics but include the same functionality.
- Programs may have many uninteresting differences that are byproducts of minor changes in code or compiler settings.
- Correlation analysis can take *a lot* of time—we'll discuss ways to speed up this process in upcoming modules.

Correlation analysis can yield powerful results, but there are several challenges you may encounter. This slide lists caveats to keep in mind.

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 4

- Correlating Malware
 - **Lab 4.1: Correlating Malware**
- Building YARA Rules
 - Lab 4.2: Writing YARA Rules
- Building capa Rules
 - Lab 4.3: Writing capa Rules
- Advanced Malware Analysis Tournament

This page intentionally left blank.

Lab 4.1

Correlating Malware

Please begin Lab 4.1 now.

Correlating Malware: Module Objectives, Revisited

- ✓ Understand the benefits of comparing malware samples.
- ✓ Explore Python scripts to identify similarities and differences between programs.
- ✓ Use BinDiff to compare functions, identify changes to function code, and highlight added/removed functions.
- ✓ Use Ghidra's Program Diff feature to compare similar programs at the instruction level.

This slide describes the objectives of the module we just completed.

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 4

- Correlating Malware
 - Lab 4.1: Correlating Malware
- **Building YARA Rules**
 - Lab 4.2: Writing YARA Rules
- Building capa Rules
 - Lab 4.3: Writing capa Rules
- Advanced Malware Analysis Tournament

This page intentionally left blank.

Building YARA Rules

This page intentionally left blank.

YARA Background

- YARA rules use text and hex strings to identify strings, code, and data in suspect programs.
- Incorporating YARA into your workflow can help:
 - Triage executables.
 - Classify malware.
 - Identify common malware characteristics.
 - Group malware by variant and family.
 - Hunt for related samples.
- YARA rules are a critical output of the RE process.

YARA is a tool developed to help malware analysts identify, classify, and group malware based on the presence of text or hexadecimal strings. These strings are placed into “rules”, which can be run against other files to identify matches. YARA can be applied to files on disk or content in memory, but this course will focus on developing rules for static files on disk.

YARA can help triage executables, identify common malware characteristics, and cluster malware into groups. It is considered a key output of the reverse engineering process because it often documents key functionality or unique characteristics associated with an individual file or malware family.

To learn more about YARA, browse to the official website located at <https://for710.com/yara>.

Inputs to YARA Rule Generation (I)

- Writing effective YARA rules requires that we use all the malware analysis tools and techniques we have access to.
- Outputs from static file analysis, behavioral analysis, and code analysis can all feed into the YARA rule development process.
- Key inputs for YARA rule development include:
 - Static file characteristics (pestats.py).
 - Embedded strings (PeStudio, strings64.exe, pecompare.py).
 - Code analysis results (Ghidra).
 - Binary comparison results (Ghidra, BinDiff).

Creating effective YARA rules requires that we consider the results of all phases of our analysis process: static file analysis, behavioral analysis, and code analysis.

YARA Rule Example

This rule includes metadata, a text string, a hex string with wild cards, and several conditions.

```
rule rand_loader {
  meta:
    description = "Detects code similar to the rand()-based PE and
    shellcode loader described in the referenced article."
    author = "Anuj Soni"
    reference1 = "https://for710.com/blackberryblog"
    date = "2020-07-16"
    hash1 = "DA581A5507923F5B990FE5935A00931D8CD80215BF588ABEC425114025377BB1"
    hash2 = "843CD23B0D32CB3A36B545B07787AC9DA516D20DB6504F9CDFFA806D725D57F0"
    hash3 = "7CAB7C0B3017C0830B7F518A133906E6EF7E04CE7BE83166FA6F6039474DB3F6"

  strings:
    $s1 = "D:\source\mining\wavPayloadPlayer\x64\Release\wavPayloadPlayer.pdb" ascii wide nocase
    //Loop that decodes bytes by subtracting rand() output from the encoded byte.
    $hex_rand = { E8 ?? ?? ?? ?? 25 FF 00 00 80 7D 09 FF C8 0d 00 FF FF FF FF C0 28 03 48 FF C3 48 83 EF 01 75 E0 }

  condition:
    uint16be(0) == 0x4d5a
    and filesize < 2MB
    and ($s1 or $hex_rand)
}
```

LAB_140001120			
140001120	e8 a3 1f 00 00	CALL	rand
140001125	25 ff 00 00 80	AND	EAX, 0x800000ff
14000112a	7d 09	JGE	LAB_140001135
14000112c	ff c8	DEC	EAX
14000112e	0d 00 ff ff ff	OR	EAX, 0xffffffff00
140001133	ff c0	INC	EAX
LAB_140001135			
140001135	28 03	SUB	byte ptr [RBX], AL
140001137	48 ff c3	INC	RBX
14000113a	48 83 ef 01	SUB	RDI, 0x1
14000113e	75 e0	JNZ	LAB_140001120

The best way to understand YARA rules is to review an example, as shown on this slide. Key components of a YARA rule include:

Meta data: This section includes background information about the rule author, references, and example hashes. This section is optional but highly recommended.

Comments: Use double forward slashes (“//”) for single line comments. For multiline comments, enclose the text between “/*” and “*/”.

Strings: This section includes text and hexadecimal strings. For text strings, consider these modifiers: `ascii`, `wide`, `nocase`, and `fullword`. The “fullword” modifier matches only when the specified text is delimited by non-alphanumeric characters. For example, if the fullword modifier is applied to the string “evil”, it will match “/evil/” but not “devilish”. Also, note that the text string includes the escape sequence “\” to represent a single backslash. Escape sequences are necessary if a text string includes double quotes (“”), carriage returns (\r), tabs (\t) or new lines (\n).

When crafting hexadecimal strings, consider wild cards for bytes that may vary between similar programs (“?”). The hexadecimal string used in this example is based on the code shown on the top-right of this slide. This code is from the sample with SHA-256 hash

DA581A5507923F5B990FE5935A00931D8CD80215BF588ABEC425114025377BB1. The `rand()` function called in this loop appears at different relative offsets across the three files referenced in the rule metadata. To accommodate this variation across samples, we use a hex string with wild card bytes.

Condition: This section includes the conditions to evaluate in a potential match. The first condition listed on this slide reads a 16-bit big-endian integer at the beginning of the assessed file to check for the bytes 0x4d5a (i.e., “MZ”) that indicate a Windows Executable. The second condition assesses the target file’s size. The third condition considers the presence of the text and hexadecimal strings referenced in the strings section.

For the official documentation on writing YARA rules, see <https://for710.com/yara-writing>.

Launch YARA Using yara64

yara64 <rules_file> <file or directory>

```
C:\Users\REM\Desktop>yara64 rand_loader.yar samples
rand_loader samples\843c.exe
rand_loader samples\7cab.exe
rand_loader samples\da58.exe

C:\Users\REM\Desktop>yara64.exe rand_loader.yar -s samples
rand_loader samples\843c.exe
0xf20:$hex_rand: E8 0B 44 00 00 25 FF 00 00 80 7D 09 FF C8 0D 00 FF FF FF FF C0 28 03 48 FF C3 48 83 EF 01 75 E0
rand_loader samples\7cab.exe
0x1323:$hex_rand: E8 F8 22 00 00 25 FF 00 00 80 7D 09 FF C8 0D 00 FF FF FF FF C0 28 03 48 FF C3 48 83 EF 01 75 E0
rand_loader samples\da58.exe
0x11ddc:$s1: D:\source\mining\wavPayloadPlayer\x64\Release\wavPayloadPlayer.pdb
0x520:$hex_rand: E8 A3 1F 00 00 25 FF 00 00 80 7D 09 FF C8 0D 00 FF FF FF FF C0 28 03 48 FF C3 48 83 EF 01 75 E0
```

Your Windows VM includes the official yara64.exe program. Use the command-line format at the top of this slide to run a rule against a file or directory of files.

In the examples on this slide, the rule discussed in the previous slide is run against a directory “samples” that contains the files referenced in the meta data section of the YARA rule. The first command-line output indicates all three files match the rule. In the second command-line, the “-s” argument is added to print matching strings.

Writing YARA Rules Takes Time and Practice

- Writing a rule is easy; writing an *effective* rule can be extremely difficult.
- Reducing false positives and false negatives can be challenging.
- A careful review of string output and robust code analysis looking for unique content goes a long way.
- Crafting YARA rules involves writing, testing, hunting, and iterating.
- Patience, practice, and persistence are paramount.

This slide lists key caveats associated with developing YARA rules.

YARA Modules

Modules extend YARA features, and the PE module provides access to the fields and features of a PE file:

- number_of_sections
- sections
- overlay
- number_of_resources
- version_info
- imports
- exports
- imphash

```
import "pe"
...
condition:
    uint16be(0) == 0x4D5A and
    filesize < 1MB and
    pe.imphash() == "DA687DAE353481FEBE9B9720AA9AB4BD"
```

Modules extend YARA's capabilities. One important module for PE file analysis is the PE module. This module provides access to the structure of a portable executable, including its imports, exports, and sections.

For more information on the YARA pe module, see <https://for710.com/yara-pe>.

yarGen for Rule Generation

- Sifting through hundreds or thousands of strings to find those unique to one sample or group of malware can be time-consuming and tedious.
- yarGen automates some of the work involved in generating effective YARA rules.
- Performing code analysis across multiple files looking for common opcodes can also be challenging.
- yarGen uses a database of goodware strings and opcodes (optional) to highlight values that might be unique to the target samples.
- A human analyst is still required to review the output.

yarGen is created and maintained by Florian Roth (<https://twitter.com/cyb3rops>). To visit the GitHub page for yarGen, browse to <https://for710.com/yargen>.

Run yarGen against an Individual File or Directory

```
C:\Users\REM\Desktop\Malware\Section4>python C:\Tools\yarGen-0.23.4\yarGen.py -m destiny -o destiny_yargen.yar
```

[illegible]

yarGen can be run against an individual file or group of files. In the example on this slide, yarGen is executed against a directory of files.

Super Rules Target Multiple Files

```
rule _0d03_1c41_3462_3ff1_6c98_84d2_2 {
  meta:
    description = "destiny - from files 0d03, 1c41, 3462, 3ff1, 6c98, 84d2"
    author = "yarGen Rule Generator"
    reference = "https://github.com/Neo23x0/yarGen"
    date = "2021-11-05"
    hash1 = "0d037ee0252e4f26800bcf7c750f61d0c549b7ba0a522c75e8d96dcf4f689e27"
    hash2 = "1c41acdc2e9d8b89522ebb51d65b4c41d7fd130a14ce9d449edb05f53bbb8d59"
    hash3 = "34629751d8202be456dcf149b516afe9c980a9128dd6096fd6286fee530a0d20"
    hash4 = "3ff1b90dbad5d78397fdc731c3a3c080d91fc488ac9152793b538b74a1e2d8f3"
    hash5 = "6c98d424ab1b9bfba683eda340fef6540ffe4ec4634f4b95cf9c70fe4ab2de90"
    hash6 = "84d24a16949b5a89162411ab98ab2230128d8f01a3d3695874394733ac2a1dbd"
  strings:
    $s1 = "PLEASE, READ KARMA-ENCRYPTED" fullword wide
    $s2 = "background.jpg" fullword wide
    $s3 = "default user" fullword wide
    $s4 = "system volume information" fullword wide
    $s5 = "searches" fullword wide
    $s6 = "default" fullword wide /* Goodware String - occurred 1159 times */
  condition:
    ( uint16(0) == 0x5a4d and filesize < 60KB and ( all of them )
      ) or ( all of them )
}
```

yarGen rules that identify multiple files are referred to as “super rules”.

Note that yarGen output is helpful and often saves time in generating an appropriate YARA rule, but it is just one step in the rule development process. Analysts should always review yarGen output, add/remove strings as needed, and test the rules.

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 4

- Correlating Malware
 - Lab 4.1: Correlating Malware
- Building YARA Rules
 - **Lab 4.2: Writing YARA Rules**
- Building capa Rules
 - Lab 4.3: Writing capa Rules
- Advanced Malware Analysis Tournament

This page intentionally left blank.

Lab 4.2

Writing YARA Rules

Please begin Lab 4.2 now.

Building YARA Rules: Module Objectives, Revisited

- ✓ Describe common use cases for using YARA.
- ✓ Understand best practices for writing YARA rules.
- ✓ Learn how to write an effective YARA rule.
- ✓ Gain exposure to automation options for writing YARA rules.

This slide describes the objectives of the module we just completed.

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 4

- Correlating Malware
 - Lab 4.1: Correlating Malware
- Building YARA Rules
 - Lab 4.2: Writing YARA Rules
- **Building capa Rules**
 - Lab 4.3: Writing capa Rules
- Advanced Malware Analysis Tournament

This page intentionally left blank.

Building capa Rules

This page intentionally left blank.

Building capa Rules: Module Objectives

- Introduce capa for triage and rule development.
- Discuss key aspects of capa rule formatting.
- Explain rule development best practices.
- Write a capa rule.

This slide describes the objectives of this module.

Capa: An Open-Source Tool That Assesses Program Capabilities

- capa extracts strings and disassembles code and combines this output with a logic engine to identify key functionality.
- capa supports analysis of PE, ELF, or shellcode files.
- It is a powerful triage tool, and it also provides a robust framework to document and share common malware characteristics.
- capa uses text-based, human-readable rules to identify program features, and these rules are relatively easy to write.
- capa rules are actively maintained and shared at <https://github.com/mandiant/capa-rules>.

FOR610 discusses how to use capa for triage; in this course, we discuss how to create capa rules to structure our knowledge and automate analysis.

For more information on capa, see <https://for710.com/capa>. To review existing rules, see <https://for710.com/capa-rules>.

Capa Verbose Output

```
C:\Users\REM\Desktop\Malware\Section2>c:\Tools\capa.exe -v file.exe
loading : 100%| 639/639 [00:00<00:00, 2115.75 rules/s]
matching: 100%| 254/254 [00:05<00:00, 46.98 functions/s, skipped 6 library functions (2%)]
md5 9e758f927e897e11b40c5b0edabb9381
sha1 ae160cd1a828b2695b027eb4cd9b24cb3cba3527
sha256 6b212864731c131bd095c2537ca14e10338d3ebf997dda59465c5f1ce73d418b
```

hash data with CRC32

namespace data-manipulation/checksum/crc32

scope function

matches 0x405846

encode data using Base64

namespace data-manipulation/encoding/base64

scope function

matches 0x40A4BE

hash data with CRC32

namespace data-manipulation/checksum/crc32

author moritz.raabe@mandiant.com

scope function

mbc Data::Checksum::CRC32 [C0032.001]

examples 2D3EDC218A90F03089CC01715A9F047F:0x403CBD

87E6B3EF38AB:0x100084A6

function @ 0x405846

or:

and:

mnemonic: shr @ 0x405866

number: 0xEDB88320 @ 0x40586E

number: 0x8 @ 0x40585E

characteristic: nzxor @ 0x405860, 0x405870

```

                                FUN_00405846
00405846 55          PUSH     EBP
00405847 81 58 46 00  MOV     EAX, [EBP+00405846]
00405848 41          INC     ECX
00405849 41          INC     ECX
0040584a 25 20 83 b8... AND     EAX, 0xedb88320
0040584b 33 c8       XOR     ECX, EAX

```

When executing capa, we can specify command line options to provide more detail on matches. With the `-v` and `-vv` command line options, capa details why it detected a capability and where the code or data resides.

Capa and Obfuscated Programs

```
C:\Users\REM\Desktop\Malware\Section2>c:\Tools\capa.exe 1582.exe  
loading : 100%|██████████████████████████████████████| 639/639 [00:00<00:00, 1834.71 rules/s]  
matching: 100%|█| 402/402 [00:05<00:00, 71.17 functions/s, skipped 279 library functions (69  
WARNING:capa:-----  
WARNING:capa: This sample appears to be packed.  
WARNING:capa:  
WARNING:capa: Packed samples have often been obfuscated to hide their logic.  
WARNING:capa: capa cannot handle obfuscation well. This means the results may be misleading o  
r incomplete.  
WARNING:capa: If possible, you should try to unpack this input file before analyzing it with  
capa.  
WARNING:capa:  
WARNING:capa: Identified via rule: (internal) packer file limitation  
WARNING:capa:  
WARNING:capa: Use -v or --vv if you really want to see the capabilities identified by capa.  
WARNING:capa:-----
```

Capa runs against files on disk, and it does not execute the target. As a result, this capability is less useful if the suspect file is heavily obfuscated. However, this is expected based upon how capa works, and the output clearly indicates when this is an issue.

Capa Uses YAML Files for Its Rules Engine

```
rule:
  meta:
    name: Identify functions with ARX operations.
    namespace: test_rules
    author: Anuj Soni
    scope: basic block
    examples:
      - eecc969ba17e924093821a7c862da03f8668abe833042b6bd023fbe75fa2e0e8:0x4035C0
  features:
    - and:
      - mnemonic: add
      - mnemonic: xor
    - or:
      - mnemonic: rol
      - mnemonic: ror
```

This slide includes an example that targets techniques we discussed in class. Let's discuss key aspects of a capa rule.

First, use the meta block to name the rule, identify its scope, and provide examples. As described in the rule format documentation (<https://for710.com/capa-format>), the rule name should complete the sentence "The program/function may...". The namespace is used to organize rules and should specify the folder where the rule resides. There are three options for scope: basic block, function, and file. Finally, examples should specify file hashes and virtual addresses where an analyst can find the identified capability. Note that in addition to the hash, the rule writer should specify a function or basic block address.

The features block specifies the rule logic. Acceptable expressions include and, or, not, and optional, among others.

The logic can specify a variety of features, including:

- API
- Number
- String
- Bytes
- Offset
- Mnemonic
- Characteristic

Capa Sample Output

```
C:\Users\REM\Desktop\Malware\Section2>C:\Tools\capa.exe -v -r perform_arx_operations.yml system32.exe
loading : 100%| 1/1 [00:00<00:00, 934.14 rules/s]
matching: 100%| 281/281 [00:00<00:00, 667.64 functions/s, skipped 187 library functions (66%)]
md5      896576de0813cf79998c56fa28886a6e
sha1     b6053d4eb2c44e08905476a269e9b80ee17c526e
sha256   eecc969ba17e924093821a7c862da03f8668abe833042b6bd023fbe75fa2e0e8
path     system32.exe
timestamp 2021-11-16T19:22:59.969614
capa version v3.0.3-0-g29e61e2
os       windows
format   pe
arch     i386
extractor VivisectFeatureExtractor
base address 0x400000
rules    C:\Users\REM\Desktop\Malware\Section2\perform_arx_operations.yml
function count 94
library function count 187
total feature count 6135

Identify functions with ARX operations.
namespace test_rules
scope      basic block
matches    0x4035C0
```

```
LAB_004035c0
004035c0 ADD EAX, EDI
004035c2 MOV EDI, dword ptr [EBP + local_38]
004035c5 ROL EAX, 0x7
004035c8 XOR dword ptr [EBP + local_10], EAX
004035cb MOV EAX, dword ptr [EBP + local_10]
004035ce ADD EAX, dword ptr [EBP + local_c]
004035d1 ROL EAX, 0x9
004035d4 XOR dword ptr [EBP + local_14], EAX
004035d7 MOV EAX, dword ptr [EBP + local_14]
004035da ADD EAX, dword ptr [EBP + local_10]
004035dd ROL EAX, 0xd
004035e0 XOR EDI, EAX
004035e2 MOV EAX, dword ptr [EBP + local_14]
004035e5 ADD EAX, EDI
004035e7 MOV dword ptr [EBP + local_38], EDI
004035ea ROR EAX, 0xe
004035ed XOR dword ptr [EBP + local_c], EAX
004035f0 MOV EAX, dword ptr [EBP + local_18]
004035f3 ADD EAX, dword ptr [EBP + local_20]
004035f6 ROL EAX, 0x7
```

On this slide, we run the rule described on the previous slide against system32.exe (discussed earlier in this course). The “-r” argument specifies the path to a custom rules file or directory. If a rule path is not included, the embedded rules are used.

Capa successfully identifies a basic block with ARX operations.

This Public Rule Example Involves Constants and Windows APIs

```
rule:
  meta:
    name: encrypt data using AES via WinAPI
    namespace: data-manipulation/encryption/aes
    author: moritz.raabe@mandiant.com
    scope: function
    att&ck:
      - Defense Evasion::Obfuscated Files or Information [T1027]
  mbc:
    - Defense Evasion::Obfuscated Files or Information::Encryption-Standard Algorithm [E1027.m05]
    - Cryptography::Encrypt Data::AES [C0027.001]
  examples:
    - BC577119D1A5B7DA489E7B5817D3CC38:0x10002FAC
  features:
    - and:
      - or:
        - number: 0x6611 = CALG_AES
        - number: 0x660E = CALG_AES_128
        - number: 0x660F = CALG_AES_192
        - number: 0x6610 = CALG_AES_256
      - or:
        - api: CryptGenKey
        - api: CryptDeriveKey
        - api: CryptImportKey
    - optional:
      - or:
        - number: 1 = PROV_RSA_FULL
        - api: CryptAcquireContext
        - api: CryptEncrypt
        - api: CryptDecrypt
```

This example rule is available at <https://for710.com/capa-aes-winapi>.

Lab 4.3

Writing capa Rules

Please begin Lab 4.3 now.

Building capa Rules: Module Objectives, Revisited

- ✓ Introduce capa for triage and rule development.
- ✓ Discuss key aspects of capa rule formatting.
- ✓ Explain rule development best practices.
- ✓ Write a capa rule.

This slide describes the objectives of the module we just completed.

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 4

- Correlating Malware
 - Lab 4.1: Correlating Malware
- Building YARA Rules
 - Lab 4.2: Writing YARA Rules
- Building capa Rules
 - Lab 4.3: Writing capa Rules
- **Advanced Malware Analysis Tournament**

This page intentionally left blank.

Advanced Malware Analysis Tournament

This page intentionally left blank.

Welcome to the Advanced Malware Analysis Tournament

- The final section of this course gives you an opportunity to flex your new knowledge and skills in a more independent, competitive environment.
- You must recall key concepts and perform workflows we discussed in class to successfully navigate the tournament and accumulate points.
- This is an excellent opportunity to analyze real-world, complex malware samples and reinforce your new advanced analysis skills.
- You will log onto a CTF platform and be presented with a combination of multiple choice and short answer challenges.
- To access the game, you will need to create an account at <https://www.ranges.io/sign-up> or use an existing account.

This page intentionally left blank.

Playing the Game in a Live or Hybrid Class

- The game will begin soon and end in two weeks (no extensions/exceptions).
- The highest scorer wins a FOR710 Challenge Coin.
- To access the game, log in and enter the “Event code” provided by the instructor.
- If you have questions or encounter issues while playing, e-mail for710@sans.org.
 - Be sure to specify the relevant question.
 - Allow up to 12 hours for an initial response.
- After play, submit an eval at <https://for710.com/ctfeval>.



This page intentionally left blank.

Playing the Game via OnDemand

- You will have extended access to the game for four months.
- There is no challenge coin awarded when playing via OnDemand.
- For the event code, access MyLabs through your SANS Portal account.
- If you have any questions while playing, contact OnDemand support.

This page intentionally left blank.

Tournament Notes (I)

- Analysis environment:
 - Keep your virtual machines configured using “host-only” networking.
 - In addition to your Static and Dynamic VMs, use REMnux as needed.
- Challenge questions:
 - You will find a combination of multiple choice and short answer questions.
 - Incorrect answers will cost you.
- If you believe a question is poorly written, or an answer is incorrect, please let the instructor know.

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 4

- Correlating Malware
 - Lab 4.1: Correlating Malware
- Building YARA Rules
 - Lab 4.2: Writing YARA Rules
- Building capa Rules
 - Lab 4.3: Writing capa Rules
- Advanced Malware Analysis Tournament

This page intentionally left blank.

Workbook



PLEASE READ THE TERMS AND CONDITIONS OF THIS COURSEWARE LICENSE AGREEMENT ("CLA") CAREFULLY BEFORE USING ANY OF THE COURSEWARE ASSOCIATED WITH THE SANS COURSE. THIS IS A LEGAL AND ENFORCEABLE CONTRACT BETWEEN YOU (THE "USER") AND SANS INSTITUTE FOR THE COURSEWARE. YOU AGREE THAT THIS AGREEMENT IS ENFORCEABLE LIKE ANY WRITTEN NEGOTIATED AGREEMENT SIGNED BY YOU.

With this CLA, SANS Institute hereby grants User a personal, non-exclusive license to use the Courseware subject to the terms of this agreement. Courseware includes all printed materials, including course books and lab workbooks, as well as any digital or other media, virtual machines, and/or data sets distributed by SANS Institute to User for use in the SANS class associated with the Courseware. User agrees that the CLA is the complete and exclusive statement of agreement between SANS Institute and you and that this CLA supersedes any oral or written proposal, agreement or other communication relating to the subject matter of this CLA.

BY ACCEPTING THIS COURSEWARE, USER AGREES TO BE BOUND BY THE TERMS OF THIS CLA. BY ACCEPTING THIS SOFTWARE, USER AGREES THAT ANY BREACH OF THE TERMS OF THIS CLA MAY CAUSE IRREPARABLE HARM AND SIGNIFICANT INJURY TO SANS INSTITUTE, AND THAT SANS INSTITUTE MAY ENFORCE THESE PROVISIONS BY INJUNCTION (WITHOUT THE NECESSITY OF POSTING BOND) SPECIFIC PERFORMANCE, OR OTHER EQUITABLE RELIEF.

If User does not agree, User may return the Courseware to SANS Institute for a full refund, if applicable.

User may not copy, reproduce, re-publish, distribute, display, modify or create derivative works based upon all or any portion of the Courseware, in any medium whether printed, electronic or otherwise, for any purpose, without the express prior written consent of SANS Institute. Additionally, User may not sell, rent, lease, trade, or otherwise transfer the Courseware in any way, shape, or form without the express written consent of SANS Institute.

If any provision of this CLA is declared unenforceable in any jurisdiction, then such provision shall be deemed to be severable from this CLA and shall not affect the remainder thereof. An amendment or addendum to this CLA may accompany this Courseware.

SANS acknowledges that any and all software and/or tools, graphics, images, tables, charts or graphs presented in this Courseware are the sole property of their respective trademark/registered/copyright owners, including:

AirDrop, AirPort, AirPort Time Capsule, Apple, Apple Remote Desktop, Apple TV, App Nap, Back to My Mac, Boot Camp, Cocoa, FaceTime, FileVault, Finder, FireWire, FireWire logo, iCal, iChat, iLife, iMac, iMessage, iPad, iPad Air, iPad Mini, iPhone, iPhoto, iPod, iPod classic, iPod shuffle, iPod nano, iPod touch, iTunes, iTunes logo, iWork, Keychain, Keynote, Mac, Mac Logo, MacBook, MacBook Air, MacBook Pro, Macintosh, Mac OS, Mac Pro, Numbers, OS X, Pages, Passbook, Retina, Safari, Siri, Spaces, Spotlight, There's an app for that, Time Capsule, Time Machine, Touch ID, Xcode, Xserve, App Store, and iCloud are registered trademarks of Apple Inc.

PMP® and PMBOK® are registered trademarks of PMI.

SOF-ELK® is a registered trademark of Lewes Technology Consulting, LLC. Used with permission.

SIFT® is a registered trademark of Harbingers, LLC. Used with permission.

Governing Law: This Agreement shall be governed by the laws of the State of Maryland, USA.

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

Welcome to the FOR710 Electronic Workbook

E-Workbook Overview

This electronic workbook contains all lab materials for SANS FOR710, Reverse-Engineering Malware: Advanced Code Analysis. Each lab is designed to address a hands-on application of concepts covered in the corresponding courseware and help students achieve the learning objectives the course and lab authors have established.

Some of the key features of this electronic workbook include the following:

- Inline drop-down solutions, command lines, and results for easy validation and reference
- Integrated keyword searching across the entire site at the top of each page
- Full-workbook navigation is displayed on the left and per-page navigation is on the right of each page
- Many images can be clicked to enlarge when necessary

Updating the E-Workbook

Tip

We recommend performing the update process at the start of the first day of class to ensure you have the latest content.

The electronic workbook site is stored locally in the VM so that it is always available. However, course authors may update the source content with minor fixes, such as correcting typos or clarifying explanations, or add new content such as updated bonus labs. You can pull down any available updates into the VM by temporarily connecting the VM to the internet (i.e., update the VM's network configuration from host-only to Bridged or NAT) and running the following command in a bash window:

```
workbook-update
```

In a Windows VM, open an Ubuntu bash window using the shortcut in the taskbar. Then, type `workbook-update` and press **Enter**. The script will indicate whether there were available updates. If so, be sure to refresh any pages you are currently viewing (or restart the browser) to make sure you are seeing the latest content.

After completing the e-workbook update, be sure to change the VM's network configuration back to host-only.

Using the E-Workbook

The FOR710 electronic workbook should be the home page for the browsers inside all virtual machines where it is maintained. Simply open a browser or click the home page button to immediately access it in the VMs.

You can also access the workbook from your host system by connecting to the IP address of your VM. Run `ip a` in Linux or in the Ubuntu bash shell in Windows to get the IP address of your VM. Next, in a browser on your host machine, connect to the URL using that IP address (i.e. `http://<%VM-IP-ADDRESS%>`). You should see this main page appear on your host. This method could be especially helpful when using multiple screens.

We hope you enjoy the FOR710 class and workbook!

Lab 0: Completing Lab Setup

Background

The purpose of this lab is to set up your VM environment for the FOR710 course.

Lab Objectives

- Unzip `FOR710-Windows.7z` , which contains the primary analysis virtual machine (VM) for this course.
- Clone the 710 VM.
- Rename one VM to specify it will only be used for static analysis (i.e., static file analysis, static code analysis).
- Rename the cloned VM to specify it will only be used for dynamic analysis (i.e., behavioral analysis, debugging).
- Take baseline snapshots for both VMs.

Note

A REMnux VM (<https://remnux.org/>) is provided in the class materials for convenience, but its use is not required for this course. For this reason, unzipping this VM is not explicitly covered in this lab.

Lab Preparation

Log in to your SANS portal account and download the class materials for FOR710. This should include an ISO that contains the file `FOR710-Windows.7z` .

Lab Steps

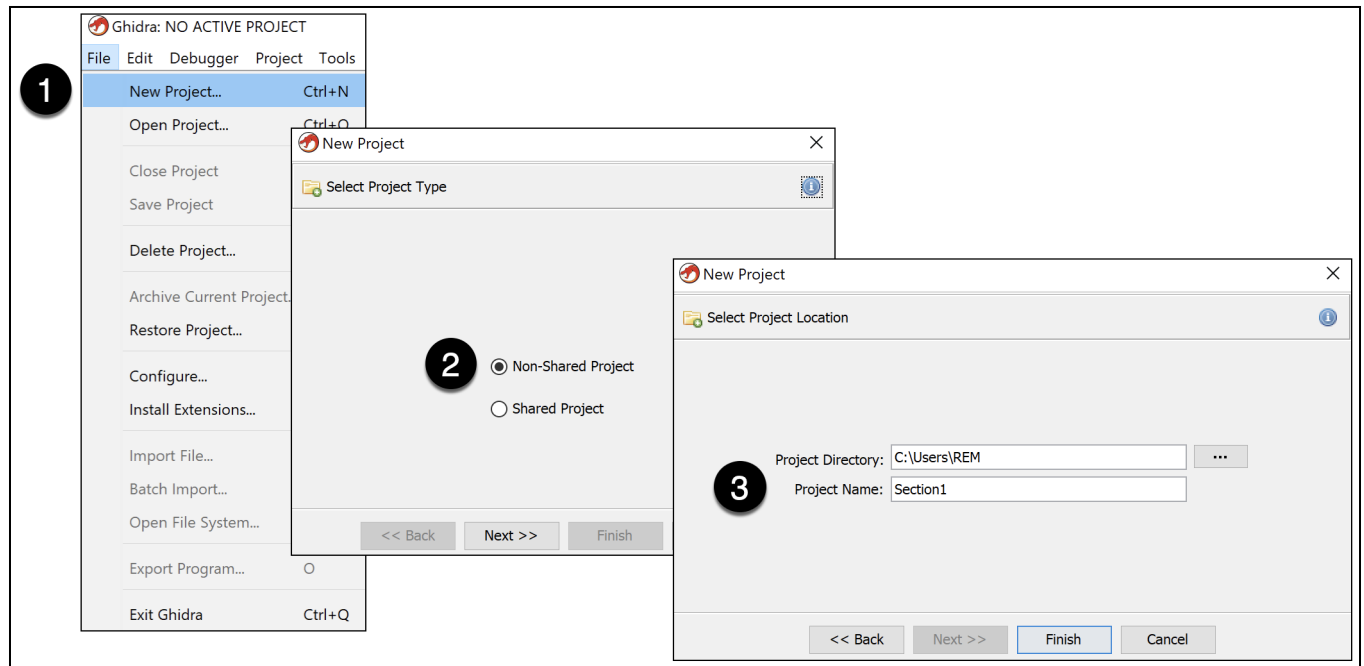
1. Unzip `FOR710-Windows.7z` to a location of your choosing on your host.
2. Within the unzipped folder, double-click the `.vmx` file to load the VM in VMware Workstation or Fusion. If prompted to upgrade the virtual machine, choose **Upgrade**.
3. After the VM boots, perform a log off/log on as needed until the resolution is acceptable. You may also modify the VMware Display settings for the virtual machine as needed and rearrange desktop icons as desired.
4. Next, we will update the electronic workbook associated with this course. The electronic workbook site is stored locally and it is accessible via the Firefox homepage (see Desktop shortcut). However, course authors may update the source content with minor fixes, such as correcting typos or clarifying explanations. To download updates, first change the VM's network configuration from host-only to Bridged or NAT. Then, launch a bash shell from the taskbar and run the command `workbook-update` . The command output will indicate whether there were available updates. If so, refresh any pages you are currently viewing (or restart the browser) to make sure you are seeing the latest content. After completing the e-workbook update, **be sure to change the VM's network configuration back to host-only**.
5. Gracefully shut down Windows (**right-click** on the Windows logo on the bottom left, and browse to **Shut down or sign out > Shut down**).
6. Within VMware Workstation or Fusion, view the list of VMs. Find the VM with name `FOR710-Windows` and choose to rename the VM. Use the new name `FOR710-Windows-Static` .

7. Right-click on the recently renamed VM and choose to create a full clone. Place the cloned VM at a location of your choosing. If prompted for a file name, use the name **FOR710-Windows-Dynamic** (do not modify the default extension).
8. After the VM is cloned, launch the new VM in VMware.
9. Within the list of available VMs listed in VMware Workstation or Fusion, find the VM clone (i.e., the new VM). Confirm that it is named **FOR710-Windows-Dynamic** . If this is not the current name, right-click on the clone and rename the VM **FOR710-Windows-Dynamic** .
10. Within the Windows OS of **FOR710-Windows-Dynamic** , right-click on the desktop and choose **Personalize**. You should arrive at the **Background** settings page. Under **Choose your background color**, choose a background **other** than black. The purpose is to visually differentiate this Dynamic VM from the Static VM. For consistency with the instructor's screen, click on **Custom color**. Then, click on **More** and insert the follow values: **Red: 225** , **Green: 115** , **Blue: 30** . If you use these color values, the background should change to an orange color.
11. Launch the FOR710 Static VM again. Once this is complete, both VMs should be up and running. You should now have two FOR710 VMs running. One VM should be named **FOR710-Windows-Static** , and a second VM should be named **FOR710-Windows-Dynamic** . Keep both VMs in host-only networking mode.
12. Take a baseline snapshot of the Static VM. Name this initial snapshot **Static Baseline** .
13. Take a baseline snapshot of the Dynamic VM. Name this initial snapshot **Dynamic Baseline** .

Lab Objectives, Revisited

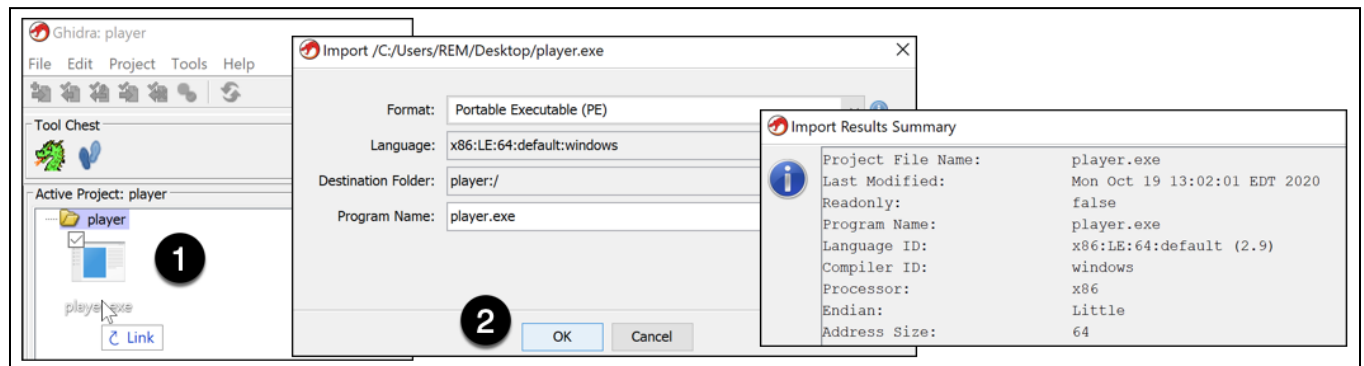
After completing this lab, you now have the following analysis environment set up:

- One VM named **FOR710-Windows-Static** for static analysis.
- A second VM named **FOR710-Windows-Dynamic** for dynamic analysis.
- The **FOR710-Windows-Dynamic** VM has a different color background when compared to the **FOR710-Windows-Static** VM.
- Both VMs are configured to use host-only networking.
- Both VMs have an initial snapshot representing a baseline state.



4. Add `player.exe` to the new project.

- Drag and drop `player.exe` into the project window so Ghidra can begin processing the file.
- An **Import** window will soon appear, prompting you to confirm various import settings. We can accept the defaults and click **OK**. The remainder of the import process will take some time to complete.
- After the import is finished, a **Import Results Summary** window will appear and provide an overview of the file and brief log of the loading process. Click **OK** or hit **Enter** to close the window.



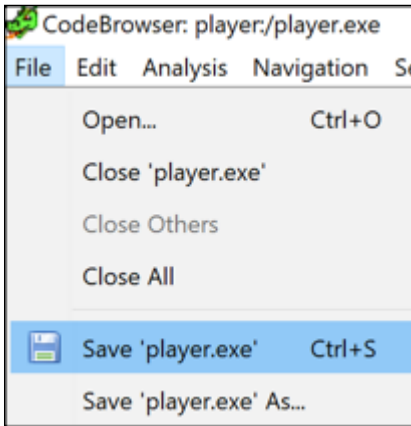
5. Launch the CodeBrowser and begin the auto-analysis.

- Return to the project window and double-click `player.exe`.
- Ghidra will generate a prompt asking if it should analyze the file; click **Yes** to configure the analysis options.
- In the next window, we need to make one change: uncheck **Decompiler Switch Analysis**. For unknown reasons, enabling this option negatively impacts Ghidra's ability to identify functions.
- Finally, click **Analyze** to kick off the analysis. You will see a progress bar on the bottom right of the CodeBrowser. If this bar is not visible, it means the window is too low on the screen—simply drag the window higher for better visibility or maximize the window to full screen.

Note

You can begin analyzing the code while the auto-analysis continues, but Ghidra's performance may lag until the file is processed. Once the analysis is complete, Ghidra may warn you that the file does not contain debug information. This is a common message and is not indicative of a serious issue.

6. Once Ghidra's auto-analysis is complete, it is a good idea to save our state to ensure we do not need to perform this time-intensive processing again. Save the project from the menu bar via **File > Save `player.exe`**.



7. Within the Dynamic VM, load `player.exe` into a debugger so we are prepared to use both dynamic and static code analysis skills to investigate this sample. Simply drag and drop the executable to the `x64dbg` shortcut on your Dynamic VM desktop.

Note

We disabled ASLR for `player.exe` so the virtual addresses in the solutions will match those in your environment.

Lab Questions

If you performed some behavioral analysis with Process Monitor or debugging with x64dbg, you would discover that `player.exe` accesses `film.wav`. Let's investigate how and why `player.exe` interacts with a WAV audio file.

1. **As stated above, we know `player.exe` accesses `film.wav`, but we need to understand how the WAV file's content is used. A review of the `player.exe` Import Address Table shows it imports `ReadFile`. Can you investigate if `player.exe` calls this Windows API to read in the contents of `film.wav`? If this is the case, what is the name and address of the function that calls `ReadFile` (i.e., the function name given within Ghidra)?**

Notes

- The question asks about the first `ReadFile` call that accesses `film.wav`.
- The question is asking for the function that calls `ReadFile`-not the instruction that calls `ReadFile`.
- Consider using a debugger (within the Dynamic VM) first to find the `CALL` instruction and then use Ghidra (within the Static VM) to identify the function name and address.

6.

? Let's now focus on the multiple CALLs to `fread` within `FUN_1400011f0`-you will notice a total of five CALLs to `fread`. What is the content of the data read by the *fourth* CALL to `fread`, and what is its relevance in the context of the WAV file format?

Notes

- Consider viewing `FUN_1400011f0` in the decompile window.
- Although the focus of this question is the fourth `fread` CALL, you will need to consider the first, and third `fread` CALLs to answer this question. You can ignore the second CALL to `fread` (it is not called for reasons that are out of scope of this lab).
- As described in the [Microsoft documentation](#), each call to `fread` will increment the file pointer by the number of bytes read. The file pointer is simply a pointer to a location within the file.
- Consult this table for more information on the WAV file format, derived from [this resource](#):

Offset (Decimal)	Size	Name	Description
0	4	ChunkID	"RIFF" header in ASCII form
4	4	ChunkSize	Size of the rest of the file
8	4	Format	Contains the letters "WAVE"
12	4	Subchunk1ID	Contains the letters "fmt". This is the beginning of the "fmt" subchunk, which describes the sound data's format.
16	4	Subchunk1Size	Size of the rest of the subchunk
20	2	AudioFormat	Values other than 1 indicate some form of compression
22	2	NumChannels	Mono = 1, Stereo = 2, etc.
24	4	SampleRate	8000, 44100, etc.
28	4	ByteRate	$== \text{SampleRate} * \text{NumChannels} * \text{BitsPerSample} / 8$
32	2	BlockAlign	Number of bytes for one sample including all channels
34	2	BitsPerSample	8 bits = 8, 16 bits = 16, etc.
36	4	Subchunk2ID	Contains the letters "data". This is the beginning of the "data" subchunk.
40	4	Subchunk2Size	Number of bytes in the data
44	*	Data	Actual sound data

7.

? How is the value read by the fourth CALL to `fread` used?

8. ? In the context of the WAV file format, what content appears at the starting address of the allocated memory discussed in the previous question? ^

 Note

Attempt to answer this question via static code analysis only. Then, you may confirm your answer via debugging.

 Important

You've reached **Checkpoint #1** in this lab. In a live class, the instructor will use checkpoints to gauge progress with this lab.

Soon after the multiple calls to `fread`, we encounter a loop that begins at 1400012e6 and ends at 1400012fd. Let's investigate this loop and how it processes WAV audio data within `film.wav`.

 Notes

- Attempt to answer the following questions based on static code analysis of the disassembly only, unless stated otherwise (i.e., debugging is not required). You may use the decompiler output and debugging to confirm your conclusions. Consider closing your decompile window for now to avoid the temptation.
- Consider commenting each line of the disassembly with your findings to assist with your analysis.

9. ? At 1400012e9, observe the TEST and JZ instructions. What is evaluated by these instructions? Be as specific as possible. ^

 Note

- `MOVSXD` ([Microsoft reference](#)) moves the 32-bit value in the source operand to the 64-bit value in the destination operand. It signs extends the value during the move, which means that the negative/positive characteristic of the source operand will remain.
- `DIL` refers to the lower 8 bits of `EDI`.

10. ? Under what conditions is the BTS instruction at 1400012ef executed? ^

11. ? If the first byte evaluated by the TEST instruction at 1400012e9 is 0x3, is the BTS instruction at 1400012ef executed? If so, how would you describe the result of executing the BTS instruction? ^

12. ? How does R8D change with each iteration of the loop, and how does this impact the BTS instruction? ^

13. ? What does the ADD instruction at 1400012f5 contribute to the overall functioning of the loop? ^

 Note

You may need to debug the executable to identify the value stored in ESI.

14. ? How many bytes of WAV audio data does the loop evaluate? ^

 Notes

As described in [this resource](#), The JC (Jump if Carry) instruction is identical to the JB (Jump if Below) instruction. View the link for a good description of conditional jump instructions, including which ones are identical in functionality.

15. ? Please summarize the functionality and purpose of the loop beginning at 1400012e6. ^

16. ? What is the final 32-bit value that includes all extracted LSBs from the loop just analyzed? Provide the hexadecimal representation. ^

 Note

Debug the program to answer this question.

 Important

You've reached **Checkpoint #2** in this lab.

17.  **How is the 32-bit value from the previous question used?** 

18.  **What type of content later appears in the allocated memory?** 

 Note

- Perform brief static analysis of the disassembly, but rely primarily on debugging to answer this question.
- This question may involve some trial and error as you continue executing code to see what content appears in the newly allocated memory.
- Allocated memory may be freed later. Note the function `j_j_free` called at 14000146e. This function may free memory so consider running the program until this call.

19.  **Dump the memory region identified in the previous question to disk and load it into the HxD hex editor (see the HxD Desktop shortcut). Carve the file using HxD to create a valid file format of the appropriate size with no overlay (i.e., no data after the end of the file). Save the modified file to disk.** 

 Note

- Note that the content within the x64dbg dump window is not located at the beginning of the memory region, so the "MZ" bytes will not appear at the start of the dumped file.
- To search for the "MZ" bytes within **HxD**, browse to **Search > Find** from the menu bar and search for the appropriate text.
- To identify overlay content, recall that the call to `operator_new` at 14000132d allocated B3200 bytes of data.
- To carve a file within **HxD** browse to **Edit > Select block...** and choose the appropriate start offset and end offset or length.

20.  **After you modify the dumped content, perform a few minutes of static file properties analysis (i.e., do not execute anything) and document a theory about the file's functionality.** 

 Important

You've reached **Checkpoint #3** in this lab.

We extracted the underlying content, but we want to understand the specific technique used to embed a program in a WAV file. The following questions will help you investigate this level of detail.

Recall that after the instruction `CALL operator_new` at 14000132d, the starting address of the allocated memory is stored in RAX. The value in RAX is then referenced at 140001335 with the instruction `MOV R14, RAX` and at 140001344 with the instruction `MOV RCX, RAX`. By looking at upcoming references to R14 and RCX, we can observe when content is placed in the newly allocated memory. The R14 register is referenced a few times near the end of the function, but not as a pointer (i.e., data is not read from or written to the location specified in R14). RCX, however, is referenced many times as a pointer in the destination operand, indicating that the current function does contain code to place content in the allocated memory.

21. ? As discussed above, the instruction `MOV RCX, RAX` at 140001344 places the starting address of the allocated region into RCX. Continue reviewing the code and identify the loop that modifies content at the address stored in RCX. Specifically, what addresses encompass the loop? ^

22. ? How many times will this loop be executed, and how is that number related to the analysis we've performed thus far? ^

 Note

Answer this question based on static code analysis-debugging is not necessary.

23. ? Let's begin understanding the purpose of the loop. At 1400013d3, we see the instruction. `TEST byte ptr [RDX + R9 * 0x1], DIL`. Debugging this code would reveal the first operand points to a byte of WAV audio data. With this in mind, what is the likely purpose of this TEST instruction and the other TEST instructions in this loop? ^

24. ? In the first loop we analyzed at 1400012e6, the code extracted LSBs from every other byte of WAV audio data (i.e., it skipped one byte in between LSB evaluations). Does this loop operate similarly? ^

 Note

Performing static code analysis to answer this question is time consuming. Try a debugger to observe what happens when the TEST instructions in this loop are executed.

25. ? How many bytes of WAV audio data does each iteration of the loop traverse (this includes "skipped" bytes, not just the ones assessed by the conditional statements)? ^

Handle value (you may have to right-click and choose **Refresh** to display Handles). Return to user code via the menu option **Debug > Run to user code**. You should arrive at `000000014000E3F9`, and you'll see the CALL to **ReadFile** is the previous instruction at `000000014000E3F3`.

Locate the above address in Ghidra (type `g` to jump to an address). Scroll up to the beginning of the function, where you'll see its name is `_read_nolock` and it begins at `14000e104`.

2. ☒ Is the function you identified in the previous question likely one written by the developer, or is it library code? ▼

Answer: `_read_nolock` at `14000e104` is library code.

Explanation: `_read_nolock` at `14000e104` is library code identified by Ghidra's Function ID (FID) analyzer. This feature can identify statically linked libraries, and it runs as part of the initial auto-analysis. Ghidra clearly identifies `_read_nolock` as a library function in the metadata provided above the function's starting address.

```
*****
* Library Function - Single Match                                     *
* Name: _read_nolock                                                *
* Library: Visual Studio 2015 Release                               *
*****

int __fastcall _read_nolock(int _FileHandle, v...
int          EAX:4          <RETURN>
int          ECX:4          _FileHandle
void *       RDX:8          _DstBuf
uint         R8D:4          _MaxCharCount
```

3. ☒ We do not want to spend time analyzing library code, so it is important that we locate the user-defined function (i.e., not library code) that makes calls to read the contents of `film.wav`. What is the name of the user-defined function that calls `ReadFile`? ▼

 Note

To answer this question, you may want to open the Function Call Tree window and view Incoming Calls.

Answer `FUN_1400011f0`

Explanation: To identify which user-defined function reads `film.wav`, we need to review functions that call `_read_nolock`. Using static code analysis, there are two potential approaches to consider: we can access the **Function Call Trees** or the **Function Call Graph**. The latter approach turns out to be challenging when reviewing a large number of function calls, so we will view the **Function Call Trees**.

Browse to **Window > Function Call Trees**, turn your attention to the **Incoming Calls** on the left side, and expand all incoming references. You will see sequences of function calls from the entry point to `_read_nolock`; in total, there are four sequences, where the entry point is located at the bottom of each expanded tree. The function we seek is between the entry point and calls to library functions identified by Ghidra. In the first two cases, if we work our way backward from the `_read_nolock` reference and skip over library function calls, we encounter `FUN_1400011f0`. In the bottom two cases, we also encounter `FUN_140006f34` along the way, but these paths seem to include more library code used to open a file (i.e., calls to `_wopenfile`), which is not our area of interest at this time (we'll come back to that function shortly).

We can confirm `FUN_1400011f0` is the user-defined function that performs reads by jumping to it to view the disassembly. Scrolling down just a bit reveals multiple calls to `fread`, the C/C++ function for reading data from a file stream (<https://for710.com/fread>).

FUN_1400011f0		
1400011f0	PUSH	RBX
1400011f2	PUSH	RDI
1400011f3	SUB	RSP, 0x18
140001229	MOV	R9, RBX
14000123c	MOV	R8D, EDI
14000123f	LEA	EDX, [RDI + 0x23]
140001242	CALL	fread
140001247	CMP	word ptr [RSP + local_4c], DI
14000124c	JZ	LAB_140001261
14000124e	MOV	R9, RBX
140001251	LEA	EDX, [RDI + 0x1]
140001254	MOV	R8D, EDI
140001257	LEA	RCX=>local_84, [RSP + 0x44]
14000125c	CALL	fread
LAB_140001261		
140001261	MOV	R9, RBX
140001264	LEA	RCX=>local_80, [RSP + 0x48]
140001269	MOV	R8, RDI
14000126c	MOV	EDX, 0x4
140001271	CALL	fread

When compiled for the Windows operating system, calls to `fread` will call the Windows API `ReadFile`. To confirm this using x64dbg, run the program until the first call to `fread`, set a breakpoint on `ReadFile`, and then step over the call to `fread`; you will hit the `ReadFile` breakpoint.

We just pivoted into code based on a Windows API. A complimentary approach, discussed in the following two questions, is to begin our analysis at the user-defined entry point.

4. ☒ Is the user-defined entry point for this program called `main` or `WinMain`?

Notes

- Use PeStudio to determine the type of program.
- Code analysis is not required to answer this question.

Answer: `WinMain`

Explanation: You can use PeStudio or ExeInfo to determine if the program is a GUI or console application (see below). Both tools indicate this is a GUI application, which means the user-defined entry point is called `WinMain`.

Program Tree

Symbol T...

Imports

Exports

entry

Functions

Labels

Classes

FUNCTION

- ulonglong __fastcall entry(void)
- ulonglong RAX:8 <RETURN>
- undefined8 Stack[0x8]:8local_res8
- undefined1 Stack[-0x1... local_18
- entry XREF[2]:

- 140002e88 48 83 e... SUB RSP, 0x28
- 140002e8c e8 07 0... CALL __security_init_cookie
- 140002e91 48 83 c... ADD RSP, 0x28
- 140002e95 e9 82 f... JMP LAB_140002d1c

You'll notice there are only a few instructions at the entry point, including a call to `__security_init_cookie` (see <https://for710.com/security-init-cookie> for more information). This function is associated with buffer overrun protection, but more importantly, this function is called by the C Runtime Library (CRT) during program initialization; this is not likely to be user-defined code.

Note

The function labelled `__security_init_cookie` shows Ghidra's Function ID (FID) feature at work. This built-in analyzer can identify statically linked libraries, including those used by Microsoft Visual Studio. Ghidra also has a Function ID Plug-in that allows users to create their own databases to identify functions of their choice.

Let's keep following code execution to locate `WinMain`; double-click on `LAB_140002d1c` to take the jump. When you arrive at address `140002d1c` and scroll down, you'll see many CALLs to functions that start with `__scrt_`, which all refer to CRT library functions. Scrolling down some more we eventually arrive at a CALL to `_get_wide_winmain_command_line`. While this is not a call to `WinMain`, it appears related. Notice that immediately after the CALL, the value in `RAX` (the return register) is placed into `R8` in preparation for the upcoming `CALL FUN_140001120`. Could this be a CALL to `WinMain`?

140002e12	CALL	__scrt_get_show_window_mode
140002e17	MOVZX	EBX, AX
140002e1a	CALL	_get_wide_winmain_command_line
140002e1f	MOV	R8, RAX
140002e22	MOV	R9D, EBX
140002e25	XOR	EDX, EDX
140002e27	LEA	RCX, [IMAGE_DOS_HEADER_140000000]
140002e2e	CALL	FUN_140001120

We can research `WinMain` on microsoft.com, to learn more about it.

WinMain function (winbase.h)

The user-provided entry point for a graphical Windows-based application.

WinMain is the conventional name used for the application entry point. For more information, see Remarks.

Syntax

C++

Copy

```
int __cdecl WinMain(  
    HINSTANCE hInstance,  
    HINSTANCE hPrevInstance,  
    LPSTR      lpCmdLine,  
    int        nShowCmd  
);
```

WinMain's first parameter is a pointer to the executable in memory. Looking at our example, the LEA instruction at 140002e27 populates ECX with a pointer to the binary's MZ header. WinMain's third parameter is a pointer to the command line, which corresponds to `RS` in our case. Both these observations allow us to conclude `FUN_140001120` (located at 140001120) is likely `WinMain`. To rename this function accordingly, click on `FUN_140001120` and type `L` on the keyboard. Then, enter the new name `WinMain` and click **OK**.

Note

Identifying `main` and `WinMain` may require some trial and error and may not always involve a CALL to `_get_wide_winmain_command_line` prior to the user-defined entry point.

View WinMain code and observe the CALL instructions. Notice that the first two CALLs reference Windows APIs, and the last CALL references the same function we identified in Question #3. In this case, beginning our analysis at the user-defined entry point quickly brings us to the same function that reads in content from `film.wav`. Let's take a closer look at `FUN_1400011f0`.

The first CALL within `FUN_1400011f0` executes `FUN_140006f34`. Additional code analysis or debugging reveals this function opens `film.wav` and returns a pointer to the file (we're skipping this basic analysis to focus our attention on code deobfuscation). The file pointer is then passed as the fourth argument to `fread` in upcoming function calls.

6. ✓ Let's now focus on the multiple CALLs to `fread` within `FUN_1400011f0`-you will notice a total of five CALLs to `fread`. What is the content of the data read by the *fourth* CALL to `fread`, and what is its relevance in the context of the WAV file format?

Notes

- Consider viewing FUN_1400011f0 in the decompile window.
- Although the focus of this question is the fourth `fread` CALL, you will need to consider the first, and third `fread` CALLs to answer this question. You can ignore the second CALL to `fread` (it is not called for reasons that are out of scope of this lab).
- As described in the [Microsoft documentation](#), each call to `fread` will increment the file pointer by the number of bytes read. The file pointer is simply a pointer to a location within the file.
- Consult this table for more information on the WAV file format, derived from [this resource](#):

Offset (Decimal)	Size	Name	Description
0	4	ChunkID	"RIFF" header in ASCII form
4	4	ChunkSize	Size of the rest of the file
8	4	Format	Contains the letters "WAVE"
12	4	Subchunk1ID	Contains the letters "fmt". This is the beginning of the "fmt" subchunk, which describes the sound data's format.
16	4	Subchunk1Size	Size of the rest of the subchunk
20	2	AudioFormat	Values other than 1 indicate some form of compression
22	2	NumChannels	Mono = 1, Stereo = 2, etc.
24	4	SampleRate	8000, 44100, etc.
28	4	ByteRate	$== \text{SampleRate} * \text{NumChannels} * \text{BitsPerSample} / 8$
32	2	BlockAlign	Number of bytes for one sample including all channels
34	2	BitsPerSample	8 bits = 8, 16 bits = 16, etc.
36	4	Subchunk2ID	Contains the letters "data". This is the beginning of the "data" subchunk.
40	4	Subchunk2Size	Number of bytes in the data
44	*	Data	Actual sound data

Answer: The fourth CALL to `fread` reads 4 bytes at offset 40 (decimal) within `film.wav`. A description of the WAV file format indicates that the value at this offset represents the size of sound data within the file. In this case, the data size is E79F20.

Explanation: The easiest way to identify the content of the fourth CALL to `fread` is to debug the program. The fourth call to `fread` occurs at `140001286`, so we can set a breakpoint at this address and run the program to identify the buffer used to store the read data.

According to [Microsoft documentation](#) for `fread`, the buffer address is specified in the first argument passed to the function. Above, we look at the contents of `RCX` to identify the address of the buffer- `14FE60`.

Next, we right-click `RCX` and choose **Follow in Dump** to keep an eye on what data gets populated at this address (note that the address within `RCX` may be different in your debugging session).

Then, from the **Debug** menu, we choose **Step over** and review the dump window to find four bytes read.

Address	Hex
000000000014FE60	20 9F E7 00 00 00 00 00 64 61 74 61 FF 7F 00 00
000000000014FE70	90 29 47 02 00 00 00 00 D0 FE 14 00 00 00 00 00
000000000014FE80	00 00 00 00 00 00 00 00 52 49 46 46 20 9F E7 00
000000000014FE90	57 41 56 45 66 6D 74 20 10 00 00 00 01 00 02 00

The bytes are displayed in little-endian, so the value read is `E79F20`. What is the significance of these bytes within the context of the WAV file format? First, let's determine the offset of this data within `file.wav`. Microsoft documentation indicates that the second argument of `fread` is the number of bytes to read. Reviewing this argument for the prior calls to `fread` can help us determine the offset. We'll briefly consult the decompiler output since it is easier to read in this case:

```
fread(local_60,0x24,1,_File);
if (local_4c != 1) {
    fread(local_84,2,1,_File);
}
fread(local_80,4,1,_File);
fread(&local_88,4,1,_File);
if (0 < local_88) {
    _DstBuf = operator_new(local_88);
    fread(_DstBuf,local_88,1,_File);
}
```

The prior three calls read in `0x24` (decimal 36), 2, and 4 bytes, respectively. However, the second call to `fread` is only executed if a condition is met, and additional debugging reveals this condition is not met (more on this shortly). As a result, the fourth `fread` call begins reading at offset $36 + 4 = 40$. Researching the [WAVE file format](#) reveals that the 4 bytes at offset 40 specifies the number of bytes of data within the file (i.e., the size of the actual sound data).

For additional detail on the condition for the third reference to `fread`, see [this additional resource](#). You'll find that the condition evaluates if the WAVE file stores compressed data, and if so, accommodates the slightly different header.

7. ✓ **How is the value read by the fourth CALL to `fread` used?**

Answer: The value is used to allocate memory with the `operator_new` function, called at 1400012ae.

Explanation: The first argument passed to `fread` is the buffer address, and we expect this address will be moved into `RCX` before `fread` is executed. At address 140001279 in the screenshot below, the address of a local variable with the label `local_88` is placed into `RCX` before `fread` is called.

At 14000128b, the value stored in `local_88` is moved into `RAX`.

At 1400012a3, this value in `RAX` is moved to `RCX` before `operator_new` is called. This is the only argument passed to `operator_new`, and it specifies the size in bytes to allocate.

140001279	LEA	RCX=>local_88, [RSP + 0x40]
14000127e	MOV	R8, RDI
140001281	MOV	EDX, 0x4
140001286	CALL	fread
14000128b	MOVSXD	RAX, dword ptr [RSP + local_88]
140001290	TEST	EAX, EAX
140001292	JG	LAB_14000129b
140001294	MOV	EAX, EDI
140001296	JMP	LAB_1400014c7
LAB_14000129b		
14000129b	MOV	qword ptr [RSP + local_res8], RBP
1400012a3	MOV	RCX, RAX
1400012a6	MOV	qword ptr [RSP + local_res10], RSI
1400012ae	CALL	operator_new

8. ✓ **In the context of the WAV file format, what content appears at the starting address of the allocated memory discussed in the previous question?**

 Note

Attempt to answer this question via static code analysis only. Then, you may confirm your answer via debugging.

Answer: The sound data contained within the WAV file.

Explanation: The `operator_new` function call referenced in this question is at 1400012ae. `operator_new` (see [Microsoft documentation](#)) returns the starting address of the allocated region in memory. At 1400012be, the value stored in `RAX` is moved into `RCX`. This occurs shortly before `fread` is executed at 1400012c4, indicating read data will be placed at the recently allocated location.

instruction places the contents of RAX into RBP. At this address, RAX contains the return value of `operator_new`. This same value in RAX is moved into RCX at 1400012be, shortly before the CALL to fread at 1400012c4. We reviewed this fread reference earlier and concluded it reads in the WAV audio data and stores it at the address specified in RCX (i.e., the first argument):

1400012ae	CALL	operator_new
1400012b3	MOVSXD	RDX, dword ptr [RSP + local_88]
1400012b8	MOV	R9, RBX
1400012bb	MOV	R8, RDI
1400012be	MOV	RCX, RAX
1400012c1	MOV	RBP, RAX
1400012c4	CALL	fread

Therefore, RBP in the first operand of the TEST instruction points to the beginning of `film.wav` audio data when the loop executes for the first time. The second operand of the TEST instruction is DIL, the lower 8 bits of `EDI`. Highlight `DIL` with a single mouse click and scroll up to locate an instruction where the `EDI` register is in the destination operand. At 14000122f, we see `MOV EDI, 0x1`. Therefore, `DIL` contains the value 1. A `TEST` performs a logical AND. Performing an AND operation between a value and 0x1 (00000001 in binary) is an approach to evaluating the least significant bit of the value.

10. ✓ Under what conditions is the `BTS` instruction at 1400012ef executed?

Answer: The `BTS` instruction is executed if the LSB evaluated in the `TEST` instruction at 1400012e9 is 1.

Explanation: If the LSB evaluated in the `TEST` instruction at 1400012e9 is 1, the zero flag is not set (i.e., it is zero). Otherwise, the zero flag is set (i.e., it is 1). The `BTS` instruction is only executed if the `JZ` at 1400012ed is not taken. The jump is not taken if the zero flag is not set, which means the LSB evaluated is 1.

11. ✓ If the first byte evaluated by the `TEST` instruction at 1400012e9 is 0x3, is the `BTS` instruction at 1400012ef executed? If so, how would you describe the result of executing the `BTS` instruction?

Answer: If the first byte evaluated is 0x3, the `BTS` instruction is executed. It sets the 31st (left-most) bit in `EBX` to 1.

Explanation: The `BTS` (Bit Test and Set) instruction sets a specified bit in a bit string to 1. For more information on this instruction, see [this resource](#). In the instruction `BTS EBX, R8D`, the bit string is located in `EBX` and the position set to 1 is located in `R8D` (the lower 32 bits of `R8`). The position number uses an index where the right-most (i.e., least significant) bit is 0, and the left-most (i.e., most significant) bit is 31.

Let's consider the instruction `TEST byte ptr [RAX + RBP * 0x1], DIL` at 1400012e9. The question assumes the first operand points to 0x3, which is represented as 00000011 in binary. We know `DIL` contains the number 1, represented as 00000001 in binary. The `TEST` instruction performs an AND operation of 00000011 and 00000001, resulting in 00000001, a non-zero value. This means the conditional jump at 1400012ed is not taken, and the `BTS` instruction is executed.

Now's let's consider the `BTS` instruction. In the first operand, `EBX` is zero. As previously discussed, the register is zeroed out at 1400012dc with the instruction `XOR EBX, EBX`. In the second operand, `R8D` contains 0x1f (decimal 31). Highlight the operand to identify content placed into this register via the instruction `LEA R8D, [RBX + 0x1f]` at 1400012e2. In that instruction, `RBX` is zero, so 0x1f is placed into `R8D`. This means the `BTS` instruction sets the bit at position 31 (i.e., the left-most bit) in `EBX` to 1.

12. ✓ How does `R8D` change with each iteration of the loop, and how does this impact the `BTS` instruction?

Answer: At 1400012f7, we see the instruction `DEC R8D`, which decrements this register by 1. This changes the bit position set by the `BTS` instruction. It begins with bit position 31 and eventually decrements to 0.

13.

✓ What does the ADD instruction at 1400012f5 contribute to the overall functioning of the loop?

 Note

You may need to debug the executable to identify the value stored in ESI.

Answer: ESI contains 2. Incrementing ECX by 2 with each run of the loop shifts the pointer to the audio data in the TEST instruction by 2 as well. Since the TEST instruction evaluates the LSB of a byte of audio data, the loop evaluates the LSB of every other byte (i.e., it skips a byte).

Explanation: The second operand of the ADD instruction is ESI. To identify the value of ESI, we highlight the register with a single mouse click and locate earlier instructions where this register is in the destination operand. At 1400012d9, we see `SHR ESI, 0x3`.

1400012d9	SHR	ESI, 0x3
1400012dc	XOR	EBX, EBX
1400012de	MOV	EDX, EBX
1400012e0	MOV	ECX, EBX
1400012e2	LEA	R8D, [RBX + 0x1f]
LAB_1400012e6		
1400012e6	MOVSXD	RAX, ECX
1400012e9	TEST	byte ptr [RAX + RBP*0x1], DIL
1400012ed	JZ	LAB_1400012f3
1400012ef	BTS	EBX, R8D
LAB_1400012f3		
1400012f3	INC	EDX
1400012f5	ADD	ECX, ESI
1400012f7	DEC	R8D

This shift right operation shifts the bits in ESI to the right by the value specified in the second operand (3). However, it's not clear what value ESI holds at this instruction, and brief static code analysis will not reveal this information. If we set a breakpoint at 1400012d9 within x64dbg and run the program, we see ESI contains the value 0x10, which is `00010000` in binary. Shifting these bits to the right by 3 results in `00000010`, or decimal 2.

The ADD instruction at 1400012f5 adds 2 to ECX with each iteration of the loop. The only other place ECX is referenced is in the MOVSXD instruction at 1400012e6, where ECX is placed into RAX. RAX is then used in the first operand of the TEST instruction as an offset from RBP, which we determined is the starting address where the `film.wav` audio data is stored. Incrementing ECX by 2 with each iteration of the loop increments the pointer to the audio data in the TEST instruction by 2. Since the TEST instruction evaluates the LSB of an individual byte of audio data, the loop assesses every other byte (i.e., it skips a byte).

14.

✓ How many bytes of WAV audio data does the loop evaluate?

LAB_1400012e6		
1400012e6	MOVSD	RAX, ECX
1400012e9	TEST	byte ptr [RAX + RBP*0x1], DIL
1400012ed	JZ	LAB_1400012f3
1400012ef	BTS	EBX, R8D
LAB_1400012f3		
1400012f3	INC	EDX
1400012f5	ADD	ECX, ESI
1400012f7	DEC	R8D
1400012fa	CMP	EDX, 0x20
1400012fd	JC	LAB_1400012e6

15. ✓ Please summarize the functionality and purpose of the loop beginning at 1400012e6.

Answer: The loop iterates over every other byte of audio data to extract LSBs. Each LSB is assigned to a bit position within a 32-bit value, starting with the left-most bit (position 31). In total, the loop spans 64 bytes of audio data and extracts 32 LSBs since it skips a byte with each run of the loop.

16. ✓ What is the final 32-bit value that includes all extracted LSBs from the loop just analyzed? Provide the hexadecimal representation.

Note

Debug the program to answer this question.

Answer: B3200

Explanation: EBX is the destination operand of the BTS instruction, so it will contain all LSBs once the loop is complete. Set a breakpoint on the instruction after the loop at 1400012ff. Then, run the program and observe the value of RBX. It is B3200.

00000001400012DC	. xor ebx,ebx	RAX	000000000000003F
00000001400012DE	. mov edx,ebx	RBX	000000000000B3200
00000001400012E0	. mov ecx,ebx	RCX	0000000000000040
00000001400012E2	. lea r8d,qword ptr ds:[rbx+1F]	RDX	0000000000000020
00000001400012E6	> movsxd rax,ecx	RBP	000000000261B040
00000001400012E9	. test byte ptr ds:[rax+rbp],dil	RSP	000000000014FE20
00000001400012ED	✓ je player.1400012F3	RSI	0000000000000002
00000001400012EF	. bts ebx,r8d	RDI	0000000000000001
00000001400012F3	> inc edx	R8	00000000FFFFFFFF
00000001400012F5	. add ecx,esi	R9	000000000261B040
00000001400012F7	. dec r8d	R10	0000000000000000
00000001400012FA	. cmp edx,20	R11	0000000000014FDB8
00000001400012FD	✓ jb player.1400012E6		
RIP → 00000001400012FF	. test ebx,ebx		

Important

You've reached **Checkpoint #2** in this lab.

17. ✓ How is the 32-bit value from the previous question used?

Answer: At 14000132d, a CALL to operator_new allocates memory of size B3200.

Explanation: Within Ghidra, highlight EBX at 1400012ff with a single left-click. As discussed in the previous question, this register contains B3200 after the loop is complete. At 140001322, the value in EBX is moved into R15, and the next instruction moves the value in R15 into RCX. This serves as the single argument passed to operator_new at 14000132d, and the argument specifies the size of memory to allocate.

1400012fa	CMP	EDX, 0x20
1400012fd	JC	LAB_1400012e6
1400012ff	TEST	EBX, EBX
140001301	JG	LAB_140001312
140001303	MOV	RCX, RBP
140001306	CALL	thunk_FUN_140006f40
14000130b	MOV	EAX, EDI
14000130d	JMP	LAB_1400014b7
LAB_140001312		
140001312	MOV	qword ptr [RSP + local_20], R14
14000131a	MOV	qword ptr [RSP + local_28], R15
140001322	MOVSXD	R15, EBX
140001325	MOV	RCX, R15
140001328	MOV	qword ptr [RSP + local_68], R15
14000132d	CALL	operator_new

18. ✓ What type of content later appears in the allocated memory?

 Note

- Perform brief static analysis of the disassembly, but rely primarily on debugging to answer this question.
- This question may involve some trial and error as you continue executing code to see what content appears in the newly allocated memory.
- Allocated memory may be freed later. Note the function j_j_free called at 14000146e. This function may free memory so consider running the program until this call.

Answer: The content begins with the ascii bytes mZ. The recently allocated memory likely contains a Windows executable.

Explanation: After the CALL to operator_new at 14000132d, the starting address of the allocated memory is stored in RAX. The value in RAX is then referenced at 140001335 with the instruction MOV R14, RAX and 140001344 with the instruction MOV RCX, RAX. By looking at upcoming references to R14 and RCX, we can observe when content is placed in the newly allocated memory. The R14 register is referenced a few times near the end of the function, but not has a pointer (i.e., data is not read from or written to the location specified in R14). RCX, however, is referenced many times as a pointer in the destination operand, indicating this function does contain code to place content in the allocated memory.

To identify the content placed in the allocated memory, we can keep an eye on its starting address in a dump window and continue executing the function FUN_1400011f0. Within x64dbg, set a breakpoint at 140001332, immediately after operator_new is called. Run the program to arrive at the breakpoint and dump the address within RAX to a dump window.

Note

The virtual addresses shown below may differ from your environment.

Assembly code snippet:

```
.jmp player.1400014B7
mov qword ptr ss:[rsp+A8],r14
mov qword ptr ss:[rsp+A0],r15
movsxd r15,ebx
mov rcx,r15
mov qword ptr ss:[rsp+60],r15
call <player.sub_140002C00>
mov r10d,esi
mov r14,rax
shl r10d,5
test ebx,ebx
jle player.14000146B
mov rcx,rax
mov qword ptr ss:[rsp+E0],r12
```

Registers (x64 fastcall):

- 1: rcx 000000007FFE0380
- 2: rdx 000000000002CC01
- 3: r8 00000000024EA0F0
- 4: r9 000000000259D301

Memory dump (Address 00000000024EA0F0):

Address	Hex	ASCII
00000000024EA0F0	0D F0 AD BA 0D F0 AD BA 0D F0 AD BA 0D F0 AD BA	.D.D.D.D.D.D.D.D
00000000024EA100	0D F0 AD BA 0D F0 AD BA 0D F0 AD BA 0D F0 AD BA	.D.D.D.D.D.D.D.D

Then, continue executing the program by setting a breakpoint later in the function. This part might involve some trial and error, so the Notes provided a hint-set a breakpoint on the `CALL` to `j_j_free` at `14000146e`. Alternatively, you may have noticed the loop that writes content to the allocated memory and decided to set a breakpoint after the loop is complete at `140001456`. When you hit either breakpoint and observe the dump window, you should see content beginning with an `mz` header. This appears to be a Windows executable. In the next step, we'll confirm this theory.

Memory dump (Address 00000000024EA0F0):

Address	Hex	ASCII
00000000024EA0F0	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZ.....ÿÿ..
00000000024EA100	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00	@.....
00000000024EA110	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00000000024EA120	00 00 00 00 00 00 00 00 00 00 00 00 20 01 00 00	20 01 00 00
00000000024EA130	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	..°..!..L!Th
00000000024EA140	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
00000000024EA150	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
00000000024EA160	6D 6F 64 65 2E 0D 0A 24 00 00 00 00 00 00 00 00	mode...\$......
00000000024EA170	5C FB AC 0A 18 9A C2 59 18 9A C2 59 18 9A C2 59	\û-..ÄY..ÄY..ÄY
00000000024EA180	AC 06 33 59 12 9A C2 59 AC 06 31 59 8D 9A C2 59	-..3Y..ÄY-..1Y..ÄY
00000000024EA190	AC 06 30 59 0D 9A C2 59 86 3A 05 59 1E 9A C2 59	-..0Y..ÄY.:..Y..ÄY
00000000024EA1A0	0B FC C1 58 11 9A C2 59 0B FC C7 58 7A 9A C2 59	.üÄX..ÄY.üCXz..ÄY
00000000024EA1B0	0B FC C6 58 0E 9A C2 59 24 FD C1 58 19 9A C2 59	.üÄX..ÄY\$yÄX..ÄY
00000000024EA1C0	C5 65 09 59 17 9A C2 59 18 9A C3 59 1F 9B C2 59	Äe.Y..ÄY..ÄY..ÄY
00000000024EA1D0	24 FD CB 58 51 9A C2 59 24 FD C2 58 19 9A C2 59	\$yëXQ..ÄY\$yÄX..ÄY
00000000024EA1E0	24 FD 3D 59 19 9A C2 59 18 9A 55 59 19 9A C2 59	\$y=Y..ÄY..UY..ÄY
00000000024EA1F0	24 FD C0 58 19 9A C2 59 52 69 63 68 18 9A C2 59	\$yÄX..ÄYRich..ÄY
00000000024EA200	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00000000024EA210	50 45 00 00 64 86 06 00 71 5C 0C 5B 00 00 00 00	PE..d...q\.[....
00000000024EA220	00 00 00 00 F0 00 22 20 0B 02 0E 0E 00 98 08 00	ð..".....

19. ✓ Dump the memory region identified in the previous question to disk and load it into the HxD hex editor (see the HxD Desktop shortcut). Carve the file using HxD to create a valid file format of the appropriate size with no overlay (i.e., no data after the end of the file). Save the modified file to disk.

Note

- Note that the content within the x64dbg dump window is not located at the beginning of the memory region, so the "MZ" bytes will not appear at the start of the dumped file.
- To search for the "MZ" bytes within **HxD**, browse to **Search > Find** from the menu bar and search for the appropriate text.
- To identify overlay content, recall that the call to `operator_new` at 14000132d allocated B3200 bytes of data.
- To carve a file within **HxD** browse to **Edit > Select block...** and choose the appropriate start offset and end offset or length.

Answer: To dump the content from the dump window, right-click in the dump window and choose **Follow in Memory Map**. Then, right-click on the appropriate memory region and choose **Dump Memory To File**. Open the dumped content in a hex editor. The first bytes of this data do not contain `MZ` because that content appears later in the dumped content. As mentioned in the Notes for this question, the dump window content was not located at the beginning of the memory region.

We must carve out the Windows executable from the dumped memory region. Using the **HxD** hex editor, you can manually delete bytes before the MZ signature as well as the overlay.

To carve out the executable from the dumped region in memory, first locate the MZ signature. Search for `MZ` (**Search > Find**), and the first hit will arrive at the appropriate offset.

To remove any overlay, create a file of size `B3200`. One way to do this within **HxD** is to first browse to **Edit > Select block...**. The **Start-offset** field should already be populated with the offset of the MZ signature. Recall that the size of memory allocated was `B3200`. Type this value into the **Length** field and click **OK**.

Then, browse to **File > Save selection...** and save the carved file to disk.

20. ✓ After you modify the dumped content, perform a few minutes of static file properties analysis (i.e., do not execute anything) and document a theory about the file's functionality. ✓

Answer: Based on the available strings and version information, this DLL may be an XMRig Monero CPU miner.

Explanation: If we load the binary into PeStudio, the available information suggests this DLL is a XMRig Monero CPU miner.

optional-header (console)	language	neutral
directories (time-stamp)	code-page	Unicode UTF-16, little endian
sections (wait...)	CompanyName	www.xmrig.com
libraries (wait...)	FileDescription	XMRig CPU miner
imports (wait...)	FileVersion	2.6.2
exports (Start)	LegalCopyright	Copyright (C) 2016-2018 xmrig.com
exceptions (1570)	OriginalFilename	xmrig.exe
tls-callbacks (n/a)	ProductName	XMRig
relocations (1326)	ProductVersion	2.6.2
resources (7)		
strings (wait...)		
debug (PGO)		
manifest (asInvoker)		
version (xmrig.exe)		
certificate (n/a)		

type (2)	size (bytes)	file-offset	blacklist (102)	hint (89)	value (10242)
ascii	14	0x00095D98	-	file	.minergate.com
ascii	19	0x00096018	-	file	miner.fee.xmrig.com
ascii	23	0x00096030	-	file	emergency.fee.xmrig.com

type (2)	size (bytes)	file-offset	blacklist (102)	hint (89)	value (10242)
ascii	22	0x00094C60	-	-	Usage: xmrig [OPTIONS]
ascii	8	0x00094C77	-	-	Options:
ascii	55	0x00094C80	-	-	-a, --algo=ALGO specify the algorithm to use
ascii	40	0x00094CB8	-	-	cryptonight
ascii	45	0x00094CE1	-	-	cryptonight-lite
ascii	46	0x00094D0F	-	-	cryptonight-heavy
ascii	47	0x00094D3E	-	-	-o, --url=URL URL of mining server
ascii	67	0x00094D6E	-	-	-O, --userpass=U:P username:password pair for mining server
ascii	53	0x00094DB2	-	-	-u, --user=USERNAME username for mining server
ascii	53	0x00094DE8	-	-	-p, --pass=PASSWORD password for mining server
ascii	87	0x00094E1E	-	-	--rig-id=ID rig identifier for pool-side statistics (needs pool support)
ascii	50	0x00094E76	-	-	-t, --threads=N number of miner threads
ascii	61	0x00094EA9	-	-	-v, --av=N algorithm variation, 0 auto select
ascii	82	0x00094EE7	-	-	-k, --keepalive send keepalive for prevent timeout (need pool support)
ascii	95	0x00094F3A	-	-	-r, --retries=N number of times to retry before switch to backup server (default: 5)
ascii	69	0x00094F9A	-	-	-R, --retry-pause=N time to pause between retries (default: 5)
ascii	90	0x00094FE0	-	-	--cpu-affinity set process affinity to CPU core(s), mask 0x3 for cores 0 and 1
ascii	79	0x0009503B	-	-	--cpu-priority set process priority (0 idle, 2 normal to 5 highest)
ascii	53	0x0009508B	-	-	--no-huge-pages disable huge pages support
ascii	49	0x000950C1	-	-	--no-color disable colored output
ascii	48	0x000950F3	-	-	--variant algorithm PoW variant
ascii	79	0x00095124	-	-	--donate-level=N donate level, default 5% (5 minutes in 100 minutes)
ascii	58	0x000951B5	-	-	-B, --background run the miner in the background

Important

You've reached **Checkpoint #3** in this lab.

We extracted the underlying content, but we want to understand the specific technique used to embed a program in a WAV file. The following questions will help you investigate this level of detail.

Recall that after the instruction `CALL operator_new` at 14000132d, the starting address of the allocated memory is stored in RAX. The value in RAX is then referenced at 140001335 with the instruction `MOV R14, RAX` and at 140001344 with the instruction `MOV RCX, RAX`. By looking at upcoming references to R14 and RCX, we can observe when content is placed in the newly allocated memory. The R14 register is referenced a few times near the end of the function, but not as a pointer (i.e., data is not read from or written to the location specified in R14). RCX, however, is referenced many times as a pointer in the destination operand, indicating that the current function does contain code to place content in the allocated memory.

21. ☒ As discussed above, the instruction `MOV RCX, RAX` at 140001344 places the starting address of the allocated region into RCX. Continue reviewing the code and identify the loop that modifies content at the address stored in RCX. Specifically, what addresses encompass the loop?

Answer: The loop begins at 1400013d0 and ends at 140001450.

Explanation: This question refers to the loop that places content into the address pointed to by RCX - the location of the recently allocated memory. The loop begins with the following code at 1400013d0:

Note

Answer this question based on static code analysis-debugging is not necessary.

Answer: `B3200` (733,696 decimal). This is the size, in bytes, of the memory allocated by the instruction `CALL operator_new` at `14000132d`.

Explanation: The stopping condition occurs at `14000144d` with the following two instructions:

			LAB_140001447
140001447	ADD	R9 ,	R10
14000144a	INC	RCX	
14000144d	SUB	R8 ,	RDI
140001450	JNZ		LAB_1400013d0
140001456	MOV	R15 ,	qword ptr [RSP + local_68]

We know `RDI` is 1 from earlier analysis.

To identify the value stored in `R8`, we highlight the register and scroll up to find instructions that contain `R8` in the destination register.

At `1400013c4`, we find the instruction `MOV, R8D, EBX`. We then highlight `EBX` to identify references to this register. This brings us to the following code:

140001322	MOVSXD	R15 ,	EBX
140001325	MOV	RCX ,	R15
140001328	MOV	qword ptr [RSP + local_68] ,	R15
14000132d	CALL		operator_new
140001332	MOV	R10D ,	ESI
140001335	MOV	R14 ,	RAX
140001338	SHL	R10D ,	0x5
14000133c	TEST	EBX ,	EBX

At `140001322`, the value in `EBX` is moved into `R15`, and that value is then moved into `RCX` in the very next instruction. Only a couple instructions later at `14000132d`, we see the instruction `CALL operator_new`. We already reviewed this instruction and determined it is passed the value `B3200`. Therefore, `R8` contains this same value when the loop at `1400013d0` executes for the first time.

With the above information in mind, we can assess that the loop under evaluation will iterate `B3200` or 733,696 times. `R8` will store this value and decrement by one with each iteration until it reaches zero. The loop executes once for each byte allocated.

23. ✓ Let's begin understanding the purpose of the loop. At `1400013d3`, we see the instruction. `TEST byte ptr [RDX + R9 * 0x1], DIL`. Debugging this code would reveal the first operand points to a byte of WAV audio data. With this in mind, what is the likely purpose of this `TEST` instruction and the other `TEST` instructions in this loop? ✓

Answer: The eight `TEST` instructions in this loop assesses the LSBs of eight bytes of WAV audio data.

Explanation: We saw a `TEST` instruction similar to this one when evaluating the smaller loop at `1400012e6`. In that case, the instruction `TEST byte ptr [RAX + RBP*0x1], DIL` at `1400012e9` assessed the LSB of one byte of WAV audio data.

24. ✓ In the first loop we analyzed at `1400012e6`, the code extracted LSBs from every other byte of WAV audio data (i.e., it skipped one byte in between LSB evaluations). Does this loop operate similarly? ✓

 Note

Performing static code analysis to answer this question is time consuming. Try a debugger to observe what happens when the TEST instructions in this loop are executed.

Answer: Yes, this loop evaluates the LSB of every other byte, skipping one byte in between.

Explanation: Within x64dbg, set a breakpoint on several TEST instructions within the larger loop. Run the program and observe that the first operand in each successive TEST instruction increments by two. This means a byte is skipped in between each LSB evaluation.

25.  **How many bytes of WAV audio data does each iteration of the loop traverse (this includes "skipped" bytes, not just the ones assessed by the conditional statements)?** 

Answer: 16 bytes.

Explanation: There are eight TEST instructions in the loop, and each one assesses one byte, skipping a byte in between. $8 \times 2 = 16$ bytes per loop iteration.

26.  **In the first loop we analyzed at 1400012e6, each loop iteration set one bit at the appropriate bit position within a 32-bit decoded value beginning with the left-most bit (position 31). In this larger loop beginning at 1400013d0 1) How many bits of decoded content does each iteration of the loop create? and 2) In what order does it set the appropriate bits in the decoded value?** 

 Note

At 1400013da, observe the instruction `CMOVNZ EAX, EDI`. A CMOVNZ instruction means "conditional move if not zero". Since EDI in this instruction contains 1, executing this code places 1 in EAX *if the recent TEST instruction did not result in zero*. If the TEST instruction result is zero, EAX is untouched (in this case, it maintains a zero value). In this way, the CMOVNZ instruction achieves a similar result as using the BTS instruction to set a value to 1.

Answer: The loop beginning at 1400013d0 creates 8 bits (1 byte) of decoded data with each iteration. Each iteration also begins work at bit position 0 until it reaches bit position 7.

Explanation: To identify the number of decoded bits created by this loop, observe the single CMOVNZ instruction and seven BTS instructions. These eight instructions have the same destination operand, EAX, and each instruction works on one bit of data. 8 bits of data equal one byte of decoded content.

Also observe the instruction `MOV byte ptr [RCX], AL` at 140001445. This instruction occurs near the end of the loop, and it places the lower 8 bits of data in EAX (the destination operand for the CMOVNZ and BTS instructions) at the address contained within RCX, where the decoded executable eventually resides. At 14000144a, RCX is incremented by 1 before each loop iteration in preparation for the next decoded byte. This further supports our theory that each loop iteration decodes one byte.

The BTS instructions set bits at lower bit positions up to bit position 7. As suggested in the Note for this question, the CMOVNZ instruction at 1400013da can set EAX (which contains zero when the loop first executes) to 1. This is equivalent to setting the bit at bit position 0. The last BTS instruction in the loop, `BTS EAX, 0x7`, sets a bit at bit position 7.

27.  **Summarize the purpose of this loop and how it decodes content.** 

This loop extracts the LSB of every other byte of WAV data to produce a Windows executable in memory. Each iteration of the loop extracts 8 bits of decoded content (1 byte) from 16 bytes of encoded data, including skipped bytes. Each iteration also begins work at bit position 0 until it reaches bit position 7. The loop produces a Windows executable of size 0xB3200 (733,696 bytes).

Lab 1.2: Analyzing Malicious Program Execution

Background

In the previous lab, we began analyzing malware that extracted code from a WAV audio file using steganography techniques. We identified the decoding algorithm, described its inner workings, and extracted the underlying binary. Next, the program must prepare this decoded binary for execution and then launch the program. We'll explore the key steps necessary to accomplish this task.

Lab Objectives

- Identify code that checks for a valid Windows Executable.
- Identify code that maps an executable into memory in preparation for execution.
- Identify code that applies relocations, if needed.
- Identify code that loads dependent DLLs and resolves APIs.
- Identify code that updates section permissions in memory.
- Identify code that locates the entry point for execution.

Lab Preparation

Complete all steps described in the Lab Preparation for lab 1.1. `player.exe` should still be loaded within Ghidra in the Static VM, and both `player.exe` and `film.wav` should be unzipped and located in the same directory within both the Static and Dynamic VMs.

Within the Static VM, load the dumped executable into CFF explorer so we can review its structure as needed. Simply drag-and-drop the dumped executable to the CFF Explorer shortcut on the 710 VM desktop. Lastly, load the dumped DLL into the `Section1` project within Ghidra and initiate the auto-analysis with the same configuration we used in Lab 1.1: uncheck the box for `Decompiler Switch Analysis`.

If you do not have access to the dumped DLL from the last lab for some reason, you can unzip the `dumped_dll.zip` file in the `Malware\Section1` folder (password: `malware`).

Lab Questions

In the previous lab, we identified a deobfuscated malicious DLL. Now, let's review what happens next to execute this DLL in memory. The first function call that occurs after the deobfuscation loop is at 14000146e-however, it is a call to a library function associated with deallocating memory and is not worthy of further investigation. At 14000147e, we see a CALL to FUN_140001b10.

1.  **Examine the arguments passed to FUN_140001b10? How many arguments are passed to the function, and what is their significance? Rename the arguments in the Decompile window.** 

Note

Static code analysis is sufficient to answer this question, but use a debugger if this becomes too time consuming.

Let's examine FUN_140001b10 to assess its purpose. Enter this function within Ghidra.

2.  **What does the check at 140001b44 evaluate?** 

Note

Be sure to add comments to your disassembly and/or decompiler output as you answer these questions.

3.  **What check is performed at 140001b70?** 

Note

When answering this question, you will encounter the `MOVSXD` instruction. This instruction performs a move but also preserves the positive/negative (i.e., signed) nature of the source. For this lab, you can interpret this instruction as an ordinary move.

4.  **What check is performed at 140001b88?** 

The code includes additional checks for a valid PE format. This includes going through the tedious process of validating each section's size to make sure the PE header accurately describes the file. We won't explore every attempt to validate the file format. Instead, let's review what happens next.

5.

? After the various file format checks are complete, we see a call to VirtualAlloc at 140001c59. Shortly before the CALL, RCX is populated with `qword ptr [RSI + 0x30]`. What field in the decoded PE header is this referring to, and what does this tell you about the purpose of this call to VirtualAlloc?

 Note

View the dumped DLL within CFF Explorer to identify the appropriate field.

6.

? What is the difference between the call to VirtualAlloc at 140001c59 and the one at 140001c76? Under what conditions does the second call to VirtualAlloc get executed?

 Important

You've reached **Checkpoint #1** in this lab.

7.

? After VirtualAlloc successfully executes, the program encounters a CALL to HeapAlloc at 140001c96. How many bytes does this function attempt to allocate?

 Notes

- Debugging is not necessary to answer this question.
- Consult <https://for710.com/heapalloc> as needed.

In the listing view, highlight the CALL to HeapAlloc and identify the corresponding pseudocode in the decompile window. In the decompile window, click on the variable that contains the return value of HeapAlloc. You will see multiple references to elements within the allocated memory as various values are assigned. This variable might be better characterized as a structure. Let's take advantage of Ghidra's feature to automatically create a structure. Right-click on the variable that contains the return value of HeapAlloc and choose **Auto Create Structure**. Right-click again on the same variable and choose **Rename Variable**. Rename the variable to `info_struct`. Your decompile window should look similar to the excerpt pictured below:

```

info_struct = HeapAlloc(hHeap, 8, 0x68);
if (info_struct == 0x0) {
    VirtualFree(lpAddress, 0, 0x8000);
    goto LAB_140001cb5;
}
info_struct->field_0x8 = lpAddress;
uVar3 = *(lVar2 + 0x16);
info_struct->field_0x50 = 0;
info_struct->field_0x20 = uVar3 >> 0xd & 1;
info_struct->field_0x28 = &DAT_140001ac0;
info_struct->field_0x30 = &DAT_140001ad0;
info_struct->field_0x38 = &DAT_140001ae0;
info_struct->field_0x40 = &DAT_140001af0;
info_struct->field_0x48 = &DAT_140001b00;
info_struct->field_0x60 = local_48.dwPageSize;

```

Within the Data Type Manager, locate the target executable, expand it (i.e., click the +) and browse to **auto_structs > astruct**. This is the default name of the structure we just created. We will keep this default name, but you could right-click and choose **Edit** to modify the name.

8. ? For now, let's focus on the structure member assignments that reference a label name beginning with "DAT_". Ghidra did not interpret the `DAT_` locations during its initial pre-processing, but it's possible there is meaningful code or data at this location. 1) Can you determine what content resides at each `DAT_` location? 2) Rename each `DAT_` location in the decompiler output with a more meaningful label after you reinterpret the bytes at these locations. 3) Lastly, rename the corresponding `info_struct` members using the same names you chose for the `DAT_` locations. ^

Note

Try answering this question using static code analysis, but if this becomes too time consuming, use a debugger.

9. ? Find the reference to `info_struct->field_0x8` in the Decompile window. What information is stored in that structure member? Describe the content; you do not need to provide the specific value. Then, rename the structure member. ^

10. ? At 140001d61, what is the purpose of the CALL to VirtualAlloc? ^

 Note

Within the decompile window, remember that an asterisk (*) means the address stored in the specified variable is dereferenced.

What is the name of the control variable?:

What is the initial value of the control variable?:

How is the control variable updated?:

What does the while condition assess, and what does this tell you about the loop?:

16.  We are still reviewing FUN_1400014f0. Within Ghidra's Listing view, identify the two CALLs to VirtualAlloc. Using x64dbg, set a breakpoint on those two CALLs (consider disabling other breakpoints to reduce confusion) and run the program. You'll find that the program only reaches one of the CALLs during execution. Based on your evaluation of that function call and the nearby call to memmove, how would you characterize the purpose of the do-while loop? Rename the function based on your analysis. 

 Note

To assist with your analysis, compare the size of each requested memory allocation with the section header content for the decoded DLL.

 Important

You've reached **Checkpoint #2** in this lab.

17.  Let's return to FUN_140001b10. At 140001dac, we see a call to FUN_140001870. Under what conditions is this function executed (review the nearby conditional jump at 140001da7)? 

18.  What is the purpose of FUN_140001870? Using static code analysis only, review the function's arguments and the beginning of the function's disassembly. Rename the function. 

Notes

- Consider the answer to the previous question when performing your analysis.
- Recall that the first member of `info_struct` is the virtual address of the mapped DLL's PE header.
- An investigation of the function's first seven instructions should be sufficient to determine what this function is likely responsible for.
- Remember, this program has work left to do before it can execute the decoded DLL. We are trying to identify what code is responsible for each step in this preparation process, but this does not require us to analyze every line of code.

19.  At 140001dc0, we see a CALL to FUN_140001930. Based on performing static code analysis of the first 10 instructions of this function, what is it likely responsible for? 

While we will not perform a comprehensive analysis of FUN_140001930, we will explore a few additional aspects of this code.

20.  At 1400019a4, what function is called? 

Note

Attempt to answer this question without debugging the program.

21.  At 140001a30, we see another CALL instruction. What function is called? How does the presence of this CALL and the one in the previous question support our theory about the purpose of FUN_140001930? Rename the function. 

Let's confirm our knowledge of Imports-related terminology and its associated structure. Within Ghidra, open the dumped DLL from the project view. You should already have the dumped DLL loaded into CFF Explorer.

22.  First, what is the virtual address (*not* relative virtual address) of the Import Directory table for this sample (assuming it is loaded at its preferred image base)? 

23.  Access the Listing view for the dumped DLL within Ghidra and jump to the address identified in the previous question. This is the location of the Import Directory Table, which includes an IMAGE_IMPORT_DESCRIPTOR structure for each imported DLL. Ghidra did not

correctly interpret the 32-bit relative virtual addresses within this structure. First, modify the data types for the non-zero elements in the first IMAGE_IMPORT_DESCRIPTOR structure. Then, answer the following questions about this first structure. ^

VA of Import Name Table:

Name of imported DLL (not the address, the actual name):

VA of Import Address Table section that corresponds to the specified DLL:

Important

You've reached **Checkpoint #3** in this lab.

The next function call within `player.exe` occurs at 140001dd0, where we see a CALL to FUN_140001620. Let's explore the body of this function.

24.  Within `player.exe`, what two Windows APIs are called inside FUN_140001620? Static code analysis should be sufficient to answer this question. ^

25.  Using x64dbg, set breakpoints on the CALLs to VirtualProtect discussed in the previous question and evaluate these calls. Based on your analysis, what is the likely purpose of FUN_140001620 overall? Rename the function based on your analysis. ^

26.  The CALL at 140001e37 within `player.exe` executes a function in the decoded DLL. What function within the decoded DLL is called? ^

Note

Debug `player.exe` to assess the contents of `RAX` at 140001e37.

We've completed our review of FUN_140001b10. We can rename it to `check_prep_dll`, or something similar. Now, let's return to the parent function, which we renamed `desteg`.

At 14000148e we have another `CALL`, which executes FUN_140001e80. Let's jump to this function and investigate its purpose.

1. ✓ **Examine the arguments passed to FUN_140001b10? How many arguments are passed to the function, and what is their significance? Rename the arguments in the Decompile window.**

 Note

Static code analysis is sufficient to answer this question, but use a debugger if this becomes too time consuming.

Answer: `FUN_140001b10` takes two arguments. The first argument is a pointer to the deobfuscated DLL (the specific address will vary). The second argument is the size of the decoded DLL: `0xB3200`. We can rename the first argument to `addr_decoded_dll` and the second argument to `size_decoded_dll`.

Explanation: To determine the number of arguments passed to `FUN_140001b10`, view the function metadata that Ghidra provides. It specifies two arguments. Alternatively, view the instructions leading up to the function call, and you will notice only `RCX` and `RDX` are modified nearby (i.e., `R8` and `R9` are not updated in close proximity to the function call).

To assess the values and significance with a debugger, set a breakpoint within x64 at `14000147e` and run the program. Observe `RCX` and `RDX` to answer this question.

Static analysis reveals the same information with just a bit more work. First, observe how `RCX` and `RDX` are populated before `FUN_140001b10` is called:

140001478	MOV	RDX, R15
14000147b	MOV	RCX, R14
14000147e	CALL	FUN_140001b10

`RCX` is populated with the value stored in `R14` and `RDX` is populated with the value stored in `R15`. If we highlight `R14` with a single click and scroll up, we see it contains the return value of the `CALL` to `operator_new` at `14000132d`:

14000132d	CALL	operator_new
140001332	MOV	R10D, ESI
140001335	MOV	R14, RAX

Based on our work in lab 1.1, this means `R14` will contain the address of the decoded DLL.

To investigate the second argument passed to `FUN_140001b10`, we must identify what is stored in `R15`. Just a few instructions earlier, we see an instruction where `R15` is populated:

140001450	JNZ	LAB_1400013d0
140001456	MOV	R15, qword ptr [RSP + local_68]
14000145b	MOV	R13, qword ptr [RSP + local_18]

To identify the value stored in the second operand, we highlight it and scroll up to see when it is referenced. This brings us once again to the `CALL` to `operator_new` at `14000132d`:

140001325	MOV	RCX, R15
140001328	MOV	qword ptr [RSP + local_68], R15
14000132d	CALL	operator_new

We see that the value in `R15` is moved into both the variable of interest *and* `RCX`, the first argument passed to `operator_new`. The first argument to `operator_new` specifies the size of memory to allocate, and we already reviewed this function call in lab 1.1. Our earlier analysis indicated the specified size was `0xB3200`, which is the second argument passed to `FUN_140001b10`.

To rename the arguments in the Decompile window, right-click each one and choose **Rename Variable**. We can rename the first argument to `addr_decoded_dll` and the second argument to `size_decoded_dll`.

Let's examine `FUN_140001b10` to assess its purpose. Enter this function within Ghidra.

2. ✓ What does the check at `140001b44` evaluate?

 Note

Be sure to add comments to your disassembly and/or decompiler output as you answer these questions.

Answer: The `CMP` instruction at `140001b44` checks if the first argument passed to `FUN_140001b10` points to `0x4D5A` to help determine if the decoded DLL is a valid Windows executable.

Explanation: To answer this question, we need to evaluate two instructions:

<code>140001b3f</code>	<code>MOV</code>	<code>EAX, 0x5a4d</code>
<code>140001b44</code>	<code>CMP</code>	<code>word ptr [RCX], AX</code>

The `CMP` instruction evaluates `AX` (the lower 2 bytes of `EAX`) against the 2 bytes pointed to by the first argument passed to `FUN_140001b10`. As discussed in the previous question, the function's first argument points to the decoded DLL. The code checks for the `ascii mZ` bytes located at the beginning of a valid Windows executable. Note that the value moved into `EAX` at `140001b3f` is `0x5a4d` and not `0x4d5a` because the two bytes read from the decoded DLL are read as little-endian data.

3. ✓ What check is performed at `140001b70`?

 Note

When answering this question, you will encounter the `MOVSXD` instruction. This instruction performs a move but also preserves the positive/negative (i.e., signed) nature of the source. For this lab, you can interpret this instruction as an ordinary move.

Answer: The check at `140001b70` checks if the decoded DLL has a valid PE header (i.e., it checks if the DLL has the `ascii PE` characters at the expected offset).

Explanation: The relevant instructions for the check at `140001b70` include:

<code>140001b60</code>	<code>MOVSXD</code>	<code>RCX, dword ptr [RCX + 0x3c]</code>
<code>140001b64</code>	<code>LEA</code>	<code>RAX, [RCX + 0x108]</code>
<code>140001b6b</code>	<code>CMP</code>	<code>R15, RAX</code>
<code>140001b6e</code>	<code>JC</code>	<code>LAB_140001b28</code>
<code>140001b70</code>	<code>CMP</code>	<code>dword ptr [RCX + R14*0x1], 0x4550</code>

The `CMP` instruction at `140001b70` performs the comparison of interest. The right operand is `0x4550` with an `ascii` representation of `PE`. Similar to the `mZ` check earlier, this value likely refers to the little endian representation of the characters `PE` that occur at the beginning of the PE header for a valid Windows executable.

Let's assess the left operand `dword ptr [RCX + R14*0x1]`. If we highlight `R14` and scroll up to identify references, we see the instruction `MOV R14, RCX` at `140001b1f`. Since `RCX` contains the first argument to the function when this instruction is executed (i.e., the pointer to the decoded DLL), `R14` will store this same address when the move is completed.

`RCX` in the `CMP` instruction at `140001b70` is most recently populated at `140001b60` with a `MOVSD` instruction. This operation moves the value at the `0x3c` offset from the beginning of the decoded DLL into `RCX`. This is the offset of the `e_lfanew` field which specifies the offset to the PE header. Therefore, the `CMP` instruction at `140001b70` checks if the PE header of the decoded DLL begins with the expected bytes `0x4550` (little endian).

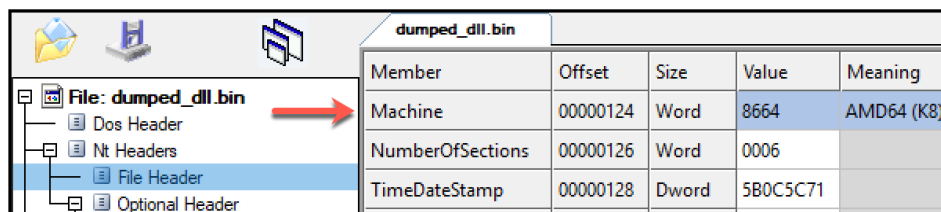
4. ✔ What check is performed at `140001b88`?

Answer: The check at `140001b88` evaluates if the decoded DLL is a 64-bit binary based on the appropriate field in the Windows executable header.

Explanation: The relevant instructions for the check at `140001b88` include:

<code>140001b7d</code>	<code>LEA</code>	<code>RSI, [RCX + R14*0x1]</code>
<code>140001b81</code>	<code>JNZ</code>	<code>LAB_140001b96</code>
<code>140001b83</code>	<code>MOV</code>	<code>EAX, 0x8664</code>
<code>140001b88</code>	<code>CMP</code>	<code>word ptr [RSI + 0x4], AX</code>

Let's review the `CMP` instruction at `140001b88`. The right operand, `AX`, is populated in the previous instruction with the value `0x8664`. To understand the left operand, we need to determine the value stored in `RSI`. This register is in the destination operand at `140001b7d` in the instruction `LEA RSI, [RCX + R14 * 0x1]`. Based on the previous question in this lab, we know the source operand is the virtual address of the decoded DLL's PE header. This means the `CMP` instruction dereferences the address at 4 bytes *after* the PE header. If we view the dumped DLL in CFF Explorer, we find the PE header signature at hex offset `0x120`. If we view the field at 4 bytes after the header at `0x124`, we arrive at the Machine type. The `CMP` instruction checks for the Machine type `0x8664` (i.e., a 64-bit executable), which matches our dumped DLL. For all machine type values, see <https://for710.com/machinetype>.



Member	Offset	Size	Value	Meaning
Machine	00000124	Word	8664	AMD64 (K8)
NumberOfSections	00000126	Word	0006	
TimeStamp	00000128	Dword	5B0C5C71	

The code includes additional checks for a valid PE format. This includes going through the tedious process of validating each section's size to make sure the PE header accurately describes the file. We won't explore every attempt to validate the file format. Instead, let's review what happens next.

5. ✔ After the various file format checks are complete, we see a call to `VirtualAlloc` at `140001c59`. Shortly before the `CALL`, `RCX` is populated with `qword ptr [RSI + 0x30]`. What field in the decoded PE header is this referring to, and what does this tell you about the purpose of this call to `VirtualAlloc`?

✎ Note

View the dumped DLL within CFF Explorer to identify the appropriate field.

Answer: As discussed in earlier questions, `RSI` is the address of the decoded DLL's PE header. At 0x30 offset from the start of the PE header is the Imagebase field. This call to `VirtualAlloc` is attempting to allocate memory at the preferred address specified in the header. This is likely to create space for the mapped executable.

File: dumped_dll.bin Dos Header Nt Headers File Header Optional Header Data Directories [x] Section Headers [x] Export Directory Import Directory Resource Directory Exception Directory Relocation Directory Debug Directory TLS Directory Address Converter	Magic	0000138	Word	020B	PE64
	MajorLinkerVersion	000013A	Byte	0E	
	MinorLinkerVersion	000013B	Byte	0E	
	SizeOfCode	000013C	Dword	00089800	
	SizeOfInitializedData	0000140	Dword	0002BC00	
	SizeOfUninitializedData	0000144	Dword	00000000	
	AddressOfEntryPoint	0000148	Dword	0004AFE0	.text
	BaseOfCode	000014C	Dword	00001000	
	ImageBase	0000150	Qword	0000001800000000	
	SectionAlignment	0000158	Dword	00001000	

6. ☒ What is the difference between the call to `VirtualAlloc` at 140001c59 and the one at 140001c76? Under what conditions does the second call to `VirtualAlloc` get executed?

Answer: The only difference between the two CALLs is that the first specifies a starting address for the region to allocate (i.e., 0x180000000). If this fails because that region in memory is already reserved, the second CALL to `VirtualAlloc` executes with no starting address specified. In this later case, the system will determine where to allocate memory.

Important

You've reached **Checkpoint #1** in this lab.

7. ☒ After `VirtualAlloc` successfully executes, the program encounters a CALL to `HeapAlloc` at 140001c96. How many bytes does this function attempt to allocate?

Notes

- Debugging is not necessary to answer this question.
- Consult <https://for710.com/heapalloc> as needed.

Answer: 0x68, or decimal 104 bytes

Explanation: The relevant instructions include:

```

140001c8a  MOV     EDX, 0x8
140001c8f  MOV     RCX, RAX
140001c92  LEA     R8D, [RDX + 0x60]
140001c96  CALL    qword ptr [->KERNEL32.DLL:HeapAlloc]

```

As described in the Microsoft documentation, `HeapAlloc`'s third argument specifies the number of bytes to allocate. This will be stored in `R8`, which gets populated at 140001c92 with an LEA instruction. This instructions adds `RDX` and 0x60, and `RDX` is populated with 0x8 at 140001c8a. Therefore, this call to `HeapAlloc` allocates $0x60 + 0x8 = 0x68$, or 104 bytes.

This means the address of an instruction that calls VirtualAlloc is placed into the `info_struct` structure. We can rename `LAB_140001ac0` to `addr_VirtualAlloc` and perform similar steps to resolve the other `DAT_` references. We can then rename the corresponding structure members by right-clicking on each field and choosing **Rename Field**. The resulting decompiled code is:

```
info_struct->field_0x8 = lpAddress;
uVar3 = *(lVar2 + 0x16);
info_struct->field_0x50 = 0;
info_struct->field_0x20 = uVar3 >> 0xd & 1;
info_struct->addr_VirtualAlloc = &addr_VirtualAlloc;
info_struct->addr_VirtualFree = &addr_VirtualFree;
info_struct->addr_LoadLibraryA = &addr_LoadLibraryA;
info_struct->addr_GetProcAddress = &addr_GetProcAddress;
info_struct->addr_FreeLibrary = &addr_FreeLibrary;
info_struct->field_0x60 = local_48.dwPageSize;
```

info

9. ✓ Find the reference to `info_struct->field_0x8` in the Decompile window. What information is stored in that structure member? Describe the content; you do not need to provide the specific value. Then, rename the structure member.

Answer: The image base address of the mapped executable. We can rename `field_0x8` to `mapped_imagebase`.

Explanation: You can use the decompile output and/or disassembly to answer this question. In the decompile window, observe the code `info_struct->field_0x8 = lpAddress;` (the corresponding disassembly is at 140001cc2). If we highlight `lpAddress` with a single click and scroll up, we see it contains the return value of the recent call to `VirtualAlloc` that attempts to allocate memory using the decoded DLL's `ImageBase`. This means `info_struct->field_0x8` contains the base address of the mapped decoded DLL. We can rename `field_0x8` to `mapped_imagebase`.

10. ✓ At 140001d61, what is the purpose of the CALL to VirtualAlloc?

Notes

- Review the next CALL to `memmove` at 140001d74 as part of your analysis (<https://for710.com/memmove>). If the function called at 140001d74 is not labelled `memmove`, it means you forgot to uncheck the box for **Decompiler Switch Analysis** when configuring Ghidra's initial auto-analysis. In this case, simply rename the function to `memmove` so you can proceed with the lab.
- Static code analysis is sufficient to answer this question, but use a debugger if this becomes too time consuming.
- If you choose to use x64dbg, note that when you view the CALL at 140001d74, the debugger will not identify the function name `memmove`. Only Ghidra provides this additional information based on its Function ID (FID) capability.

Answer: It commits space for the header of the mapped decoded DLL.

Explanation: The relevant instructions include:

140001d52	MOV	R9D, 0x4
140001d58	MOV	R8D, 0x1000
140001d5e	MOV	RCX, RBP
140001d61	CALL	qword ptr [->KERNEL32.DLL::VirtualAlloc]
140001d67	MOV	R8D, dword ptr [RSI + 0x54]
140001d6b	MOV	RDX, R14
140001d6e	MOV	RCX, RAX
140001d71	MOV	RBX, RAX
140001d74	CALL	memmove

At 140001d6e, the return value of VirtualAlloc is moved into RCX. This serves as the first argument passed to memmove, which copies a specified number of bytes from one location to another (<https://for710.com/memmove>). As specified in the Microsoft documentation, the first argument points to the destination, the second argument points to the source, and the third specifies the number of bytes to copy. Focusing on the third argument (moved into R8) provides a clue. At 140001d67, RSI contains the address of the decoded DLL's PE header. At offset 0x54 from this location is the SizeOfHeaders field, indicating PE header content will be moved into the recently allocated memory region.

We can confirm this answer if we review the CALL to memmove in a debugger, we will observe that it copies 0x400 (1024) bytes from the beginning of the decoded DLL to the starting address of the memory region allocated for the mapped binary. This is the entire header of the decoded DLL.

11. ✓ At 140001d86, what does the MOV instruction accomplish? (i.e., what is the significance of the value that is copied?) Rename the corresponding member in the `info_struct` structure within the Decompile window.

Answer: The MOV instruction places the virtual address (VA) of the mapped DLL's PE header into the first member of the `info_struct` structure. We can rename `field_0x0` in the Decompile window to `mapped_pe_header`.

Explanation: We are evaluating the instruction `MOV qword ptr [RDI], RAX` at 140001d86. First, what is RDI? Immediately after the CALL to HeapAlloc at 140001c96, the contents of EAX are moved into RDI. This means RDI contains the starting address for the allocated memory. As previously discussed, this is the address of the `info_struct` structure, so the MOV instruction populates the structure's first member.

Next, let's turn our attention to EAX. At 140001d79, a MOV instruction dereferences `R14 + 0x3c`. We encountered this location reference in a previous question—it refers to the `e_lfanew` field within the decoded DLL, which specifies the offset to the PE header. At 140001d80, RBX is added to this value. If we highlight this register with a single click, we can see it contains the return value of the CALL to VirtualAlloc we reviewed earlier—this committed memory to copy the header of the decoded DLL. Adding this return value to the offset of the PE header equals the virtual address of the mapped PE header.

If we highlight the MOV instruction at 140001d86, the corresponding code in the Decompile window references `info_struct->field_0x0`. Given the analysis described above, we can rename `field_0x0` in the Decompile window to `mapped_pe_header`.

12. ✓ At 140001d8f, what does the MOV instruction accomplish?

Answer: It updates the ImageBase field of the mapped DLL to match the actual starting address of the DLL in memory. However, if the DLL was loaded at its preferred base address, this operation results in no change.

Explanation: The relevant instruction is `MOV qword ptr [RAX + 0x30], RBP`. First, let's assess the destination operand. Based on the analysis in the previous question, we know `RAX` is the virtual address of the mapped DLL's PE header. If we use CFF Explorer to view the offset of the PE header, we see it occurs at 0x120. Adding 0x30 to that offset equals 0x150, which is the location of the ImageBase field.

Now, let's review the second operand. If we highlight `RBP` with a single click and scroll up, we see it contains the return value of the recent call to VirtualAlloc that attempts to allocate memory using the decoded DLL's ImageBase. This means `RBP` contains the base address of the mapped decoded DLL.

Therefore, the MOV instruction updates the ImageBase field of the mapped DLL to reflect the starting address of the mapped DLL in memory.

13. ✓ At 140001d93 we see a call to function FUN_1400014f0. Jump to this function and view the decompile window. How many arguments does this function take, and what is the significance of each argument? Rename each argument in the decompile window so the labels are more meaningful.

Notes

- Use a combination of static and dynamic code analysis to speed up your analysis.
- When renaming arguments and variables in the decompile window, consider using the same labels you used in earlier questions if appropriate. Good terms to include in your names are "mapped", "unmapped", "decoded", "dll", "size", and "header", separated by underscores (e.g., `size_decoded`).

Answer: This function accepts four arguments:

- **First:** The starting address of the unmapped DLL.
- **Second:** The size of the decoded DLL (0xB3200).
- **Third:** The address of the unmapped DLL's PE header.
- **Fourth:** The address of the `info_struct` structure.

One approach to renaming these arguments in the decompile window results in the following:

```
FUN_1400014f0(longlong addr_decoded_dll,ulonglong size_decoded,longlong unmapped_pe_header,
             longlong *info_struct)
```

14. ✓ We know that the fourth argument passed to FUN_1400014f0 is `info_struct`. Apply the `astruct` data type to this argument. What Windows API does FUN_1400014f0 call using `info_struct`?

Answer: `FUN_1400014f0` calls `VirtualAlloc` using the address of this API stored within `info_struct`.

Explanation: If you single-click `info_struct` within `FUN_1400014f0` in the decompile window, you will see multiple references. To apply the `astruct` structure, right-click on `info_struct` and choose **Retype Variable**. Then, type `astruct`, choose the first option, and click **OK**. This question focuses on function calls, and only two references within `FUN_1400014f0` use `info_struct` to call a function. In both cases the reference is `info_struct->addr_VirtualAlloc`.

15. ✓ Notice that most of FUN_1400014f0 is actually a do-while loop. Using the decompile window, answer the questions below.

Note

Within the decompile window, remember that an asterisk (*) means the address stored in the specified variable is dereferenced.

Answers: This decompiled code excerpt includes the information necessary to answer these questions.

Note

To assist with your analysis, compare the size of each requested memory allocation with the section header content for the decoded DLL.

Answer: The do-while loop maps each section of the unmapped decoded DLL into memory. We can rename `FUN_1400014f0` to `map_sections`, or something similar.

Explanation: Although the do-while loop includes two CALLs to `VirtualAlloc`, only the second CALL at `1400015b0` is encountered during execution. The first CALL to `VirtualAlloc` at `140001569` is only executed if a section has a zero raw size. This does not apply to our decoded DLL, so we will not explore this further.

If we set a breakpoint at `1400015b0` and run the program, we'll see it hits this breakpoint with each iteration of the loop. Each time `VirtualAlloc` is called, the first argument (i.e., the starting address of the region to allocate) is in close proximity to the image base value of the mapped DLL-this is the first indication that this code is performing additional mapping activities. In addition, each call specifies a `0x00001000` memory allocation type, which represents `MEM_COMMIT`. This means the memory has already been reserved-we previously reviewed the `VirtualAlloc` CALL that reserved the necessary space in memory. If we look at the size of each requested memory allocation and compare them with the section header content for the decoded DLL, we'll find that the numbers match the Raw Size for each section.

If we include the CALL to `memmove` at `1400015cb` into our analysis and review the source and destination each time `memmove` is called, we find that this code copies content from each section in the unmapped DLL to the memory allocated for the mapped DLL. This is further evidence that the do-while loop is responsible for mapping each section of the decoded DLL into memory in preparation for execution. Rename `FUN_1400014f0` to `map_sections`, or something similar.

Important

You've reached **Checkpoint #2** in this lab.

17. ☒ Let's return to `FUN_140001b10`. At `140001dac`, we see a call to `FUN_140001870`. Under what conditions is this function executed (review the nearby conditional jump at `140001da7`)?

Answer: `FUN_140001870` is executed if the image base of the mapped DLL is not equal to the `ImageBase` value within the unmapped DLL. In other words, this function is executed if the DLL was mapped to an address that is different from its preferred address.

Explanation: The relevant instructions for this question include:

<code>140001d9c</code>	<code>MOV</code>	<code>RAX, qword ptr [RDI]</code>
<code>140001d9f</code>	<code>MOV</code>	<code>RDY, qword ptr [RAX + 0x30]</code>
<code>140001da3</code>	<code>SUB</code>	<code>RDY, qword ptr [RSI + 0x30]</code>
<code>140001da7</code>	<code>JZ</code>	<code>LAB_140001db6</code>
<code>140001da9</code>	<code>MOV</code>	<code>RCX, RDI</code>
<code>140001dac</code>	<code>CALL</code>	<code>FUN_140001870</code>

At `140001d9c`, `RDI` contains the address of `info_struct` (see the decompiler output to confirm this). Dereferencing this value places the virtual address of the mapped DLL's PE header into `RAX`.

At `140001d9f`, the `ImageBase` value of the mapped DLL is placed into `RDY`.

At `140001da3`, the second operand dereferences `[RSI + 0x30]`. `RSI` contains the address of the unmapped DLL's PE header (static code analysis, debugging, and the decompiler output can all confirm this), so adding `0x30` points to the unmapped DLL's `ImageBase`.

Therefore, the `SUB` instruction subtracts the unmapped DLL's `ImageBase` value from the mapped DLL's `ImageBase` value. If the result is zero (i.e., the image bases are the same) this means the DLL was loaded at its preferred address. In this case, the jump at `140001da7` is taken, and

`FUN_140001870` is not executed. If the result of the subtraction is *not* zero (i.e., the image bases are different), the jump at `140001da7` is not taken, and `FUN_140001870` is executed.

18. ✓ What is the purpose of `FUN_140001870`? Using static code analysis only, review the function's arguments and the beginning of the function's disassembly. Rename the function.

Notes

- Consider the answer to the previous question when performing your analysis.
- Recall that the first member of `info_struct` is the virtual address of the mapped DLL's PE header.
- An investigation of the function's first seven instructions should be sufficient to determine what this function is likely responsible for.
- Remember, this program has work left to do before it can execute the decoded DLL. We are trying to identify what code is responsible for each step in this preparation process, but this does not require us to analyze every line of code.

Answer: Based on our analysis in the previous question, `FUN_140001870`'s first argument is the address of `info_struct`, and the second argument is the difference between the mapped DLL and unmapped DLL image base values. This function processes the DLL's .reloc section to perform any address fix ups. We can rename the function to `apply_base_relocations`, or something similar.

Explanation: If we jump to `FUN_140001870` the initial instructions include:

```
FUN_140001870
140001870    MOV     qword ptr [RSP + local_res8], RBX
140001875    MOV     qword ptr [RSP + local_res10], RDI
14000187a    MOV     RAX, qword ptr [RCX]
14000187d    MOV     R11, RDX
140001880    MOV     RDI, qword ptr [RCX + 0x8]
140001884    CMP     dword ptr [RAX + 0xb4], 0x0
14000188b    JNZ     LAB_1400018a2
14000188d    XOR     EBX, EBX
14000188f    TEST    RDX, RDX
140001892    SETZ    BL
140001895    MOV     EAX, EBX
140001897    MOV     RBX, qword ptr [RSP + local_res8]
14000189c    MOV     RDI, qword ptr [RSP + local_res10]
1400018a1    RET

LAB_1400018a2
1400018a2    MOV     EDX, dword ptr [RAX + 0xb0]
```

At `14000187a`, the first argument (i.e., the address of `info_struct`) is dereferenced, which places the address of the mapped DLL's PE header into `RAX`. At `140001884`, `[RAX + 0xb4]` takes us to the `0xb4` offset from the PE header-this is the location of the Relocation Directory Size. The `CMP` instruction evaluates if this size is zero, and if so, the function returns. Based only on these instructions, it is likely this function processes the DLL's relocation table to perform the necessary fixups.

19.

- ✓ At 140001dc0, we see a CALL to FUN_140001930. Based on performing static code analysis of the first 10 instructions of this function, what is it likely responsible for?

Answer: Based on the reviewing the CMP instruction at the beginning of the function, it is likely responsible for processing the mapped DLL's import table to resolve dependencies.

Explanation: The first 10 instructions are:

```

                                FUN_140001930
140001930      PUSH      RDI
140001932      PUSH      R12
140001934      PUSH      R13
140001936      SUB       RSP, 0x30
14000193a      MOV       RAX, qword ptr [RCX]
14000193d      MOV       RDI, RCX
140001940      MOV       R12, qword ptr [RCX + 0x8]
140001944      MOV       R13D, 0x1
14000194a      CMP       dword ptr [RAX + 0x94], 0x0
140001951      JZ        LAB_140001ab2
  
```

After typical function prologue activities, the MOV instruction at 14000193a dereferences RCX and places the value in RAX. Static code analysis and debugging will both confirm RCX contains the address of info_struct, so the first element of the array is placed into RAX. We know the first element is the address of the mapped DLL's PE header.

At 14000194a, the CMP instruction evaluates the value stored at [RAX + 0x94]. To identify what resides at this offset, view the dumped DLL in CFF Explorer and first find the offset of the PE header (0x120). Adding 0x94 results in 0x1B4. If we navigate to this offset with CFF Explorer, we find the Import Directory Size field:

File: dumped_dll.bin					
Member	Offset	Size	Value	Section	
Export Directory RVA	000001A8	Dword	000A6870	.rdata	
Export Directory Size	000001AC	Dword	00000044		
Import Directory RVA	000001B0	Dword	000A68B4	.rdata	
Import Directory Size	000001B4	Dword	00000064		
Resource Directory RVA	000001B8	Dword	000B1000	.rsrc	

This function likely parses the import table to resolve dependencies.

While we will not perform a comprehensive analysis of FUN_140001930, we will explore a few additional aspects of this code.

20.

- ✓ At 1400019a4, what function is called?

Note

Attempt to answer this question without debugging the program.

Answer: The address of info_struct is the only argument passed to FUN_140001930. If we retype (right-click > Retype Variable) the argument to a structure of type astruct within the Decompile window, we find that the CALL at 1400019a4 executes LoadLibraryA:

```

14000195c  LEA    EDX, [R13 + 0x13]
140001960  MOV    R14D, dword ptr [RAX + 0x90]
140001967  ADD    R14, R12
14000196a  MOV    RCX, R14
14000198f  NOP

LAB_140001990
140001990  MOV    EAX, dword ptr [R14 + 0xc]
140001994  TEST   EAX, EAX
140001996  JZ     LAB_140001a99
14000199c  MOV    RDX, qword ptr [RDI + 0x50]
1400019a0  LEA    RCX, [R12 + RAX*0x1]
1400019a4  CALL   qword ptr [RDI + 0x38]
1400019a7  MOV    RBP, RAX
  
```

```

undefined4 FUN_140001930(astruct *param_1)
{
    pvVar1 = param_1->mapped_imagebase;
    if (*(param_1->mapped_pe_header + 0x94) != 0) {
        lp = *(param_1->mapped_pe_header + 0x90) + pvVar1;
        BVar2 = IsBadReadPtr(lp, 0x14);
        if (BVar2 == 0) {
            while (lp[3] != 0) {
                lVar4 = (*param_1->addr_LoadLibraryA)(pvVar1 + lp[3], param_1->field_0x50);
                if (lVar4 == 0) {
                    return 0;
                }
            }
        }
    }
}
  
```

21. ✓ At 140001a30, we see another CALL instruction. What function is called? How does the presence of this CALL and the one in the previous question support our theory about the purpose of FUN_140001930? Rename the function.

Answer: Using a similar approach to the previous question, you can determine this is a call to GetProcAddress. LoadLibrary and GetProcAddress are commonly used to load a module and resolve a function within the module. This supports our theory that the overall function is responsible for loading DLLs and resolving APIs. We can rename `FUN_140001930` to `resolve_imports`.

Let's confirm our knowledge of Imports-related terminology and its associated structure. Within Ghidra, open the dumped DLL from the project view. You should already have the dumped DLL loaded into CFF Explorer.

22. ✓ First, what is the virtual address (*not* relative virtual address) of the Import Directory table for this sample (assuming it is loaded at its preferred image base)?

Answer: `0x1800A68B4`. The preferred image base is `0x180000000`, and the Import Directory RVA is `0xA68B4`. Adding these values results in `0x1800A68B4`.

Member	Offset	Size	Value	Section
Export Directory RVA	000001A8	Dword	000A6870	.rdata
Export Directory Size	000001AC	Dword	00000044	
Import Directory RVA	000001B0	Dword	000A68B4	.rdata
Import Directory Size	000001B4	Dword	00000064	

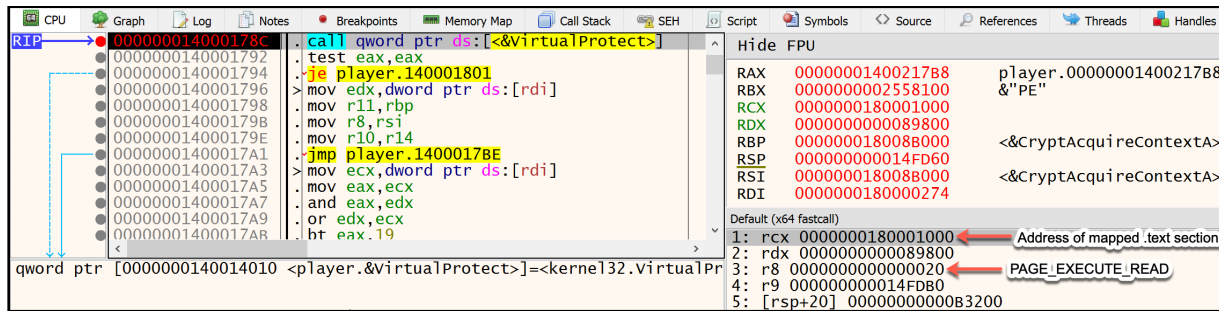
23. ✓ Access the Listing view for the dumped DLL within Ghidra and jump to the address identified in the previous question. This is the location of the Import Directory Table, which includes an `IMAGE_IMPORT_DESCRIPTOR` structure for each imported DLL. Ghidra did not correctly interpret the 32-bit relative virtual addresses within this structure. First, modify the data types for the non-zero elements in the first `IMAGE_IMPORT_DESCRIPTOR` structure. Then, answer the following questions about this first structure.

Answer: First, jump to `0x1800A68B4` within the Listing view for the dumped DLL. You should see the following:

```

*****
*  IMAGE_IMPORT_DESCRIPTOR  *
*****
DWORD_1800a68b4                XREF[1]: 1800001b0 (*)

1800a68b4  ddw    A6E68h
1800a68b8  ddw    0h
1800a68bc  ddw    0h
1800a68c0  ddw    A6F6Ch
1800a68c4  ddw    8B550h
  
```



We can rename `FUN_140001620` to `update_section_protections`.

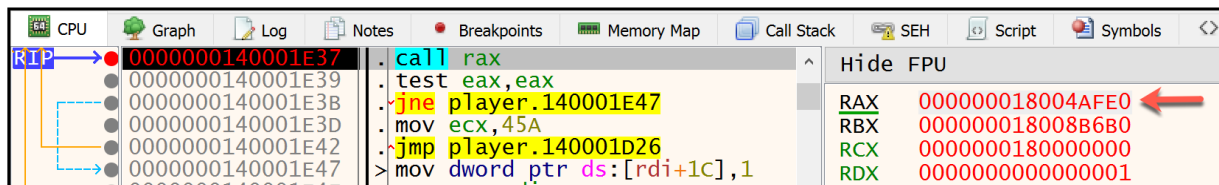
26. ✓ The CALL at 140001e37 within `player.exe` executes a function in the decoded DLL. What function within the decoded DLL is called?

Note

Debug `player.exe` to assess the contents of `RAX` at 140001e37.

Answer: The CALL at 140001e37 executes the mapped DLL's entry point located at 18004AFE0. This is an optional step when loading a DLL for execution.

Explanation: If we set a breakpoint at 140001e37 and run the program, we see `RAX` contains 18004AFE0:



Within Ghidra's Listing view, we can jump to this location in the decoded DLL. It is clear this is the DLL's entry point:

```

*****
*
*                               FUNCTION
*****

ulonglong __fastcall entry(undefined8 param_1,..

ulonglong      RAX:8      <RETURN>
undefined8     RCX:8      param_1
int            EDX:4      param_2
longlong       R8:8       param_3
undefined8     Stack[0x20... local_res20
undefined8     Stack[0x18... local_res18
undefined8     Stack[0x10... local_res10

undefined8     Stack[0x8]:8local_res8

undefined4     Stack[-0x2... local_28

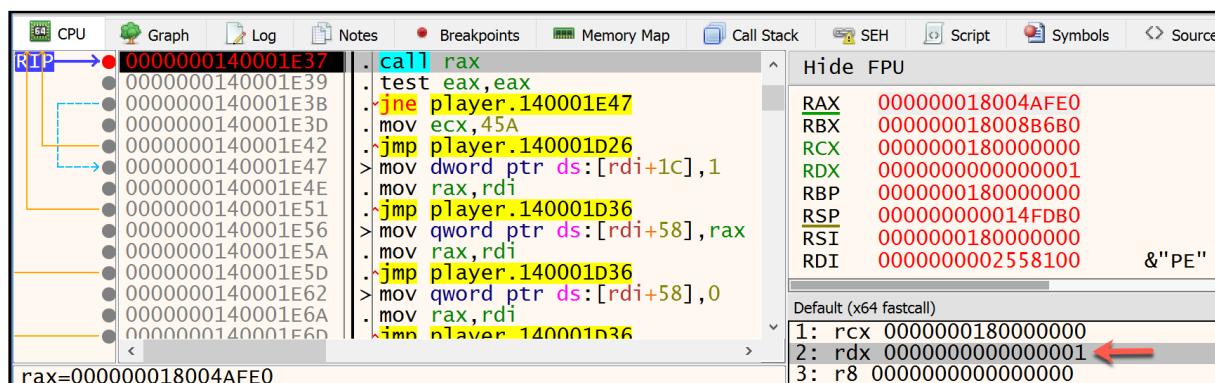
entry XREF[2]

18004afe0      MOV      qword ptr [RSP + local_res8], RBX

```

More information about a DLL's entry point can be found here: <https://for710.com/dll-ep>.

Debugging also reveals the second argument passed to the entry point is 1:



The second argument specifies the reason code, which describes why the DLL entry point is called. In this case, the value is 1 or `DLL_PROCESS_ATTACH`. This means the DLL is being initialized in preparation for execution.

We've completed our review of `FUN_140001b10`. We can rename it to `check_prep_dll`, or something similar. Now, let's return to the parent function, which we renamed `desteg`.

At `14000148e` we have another `CALL`, which executes `FUN_140001e80`. Let's jump to this function and investigate its purpose.

27. ✓ Within `player.exe`, what does the instruction at `140001e96` evaluate, and what does this tell you about the likely purpose of this function?

Answer: The `CMP` instruction checks if the Export Directory Size is zero. If the size is zero, the code calls `SetLastError` and returns. Based on this brief analysis, `FUN_140001e80` probably resolves the DLL's export(s) for execution.

Explanation: As we've seen several times during this lab, `FUN_140001e80` takes only one argument, and it is the address of `info_struct`. If we retype (right-click > **Retype Variable**) the argument to a structure of type `astruct` within the Decompile window, we find that the instruction at 140001e96 accesses the `info_struct` member that contains the address of the mapped DLL's PE header. Looking at the decoded DLL in CFF Explorer, we know that the PE header begins at 0x120, and adding 0x8c equals 0x1ac. At this offset we find the Export Directory Size field:

Member	Offset	Size	Value	Section
Export Directory RVA	000001A8	Dword	000A6870	.rdata
Export Directory Size	000001AC	Dword	00000044	
Import Directory RVA	000001B0	Dword	000A68B4	.rdata

If the Export Directory Size is zero, the function returns:

140001e96	CMP	dword ptr [RAX + 0x8c], 0x0
140001e9d	JNZ	LAB_140001eb5
140001e9f	MOV	ECX, 0x7f
140001ea4	CALL	qword ptr [->KERNEL32.DLL::SetLastError]
140001eaa	XOR	EAX, EAX
140001eac	ADD	RSP, 0x28
140001eb0	POP	R15
140001eb2	POP	R14
140001eb4	RET	

Based on this review, `FUN_140001e80` probably resolves the mapped DLL's exports in preparation for execution.

Let's explore a few more instructions within `FUN_140001e80`. We will focus on the instructions between 140001f73 and 140001f84:

140001f73	MOV	EDX, dword ptr [RBX + 0x1c]
140001f76	LEA	ECX, [RAX*0x4]
140001f7d	LEA	RAX, [R14 + RDX*0x1]
140001f81	MOV	EAX, dword ptr [RCX + RAX*0x1]
140001f84	ADD	RAX, R14

28. ✓ The instruction at 140001f73 is `MOV EDX, dword ptr [RBX + 0x1c]`. `RBX` contains the address of an export-related structure. What structure resides at the address, and what specific field within the structure does `RBX + 0x1c` point to?

Answer: `RBX` contains the virtual address of the DLL's Export Directory Table. `RBX + 0x1c` is the location of the `AddressOfFunctions` field, which specifies the RVA of the Export Address Table.

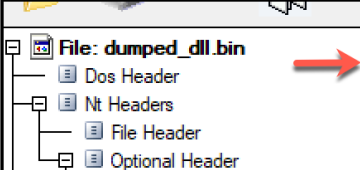
Explanation: The instruction at 140001f73 is `MOV EDX, dword ptr [RBX + 0x1c]`. We can identify the contents of `RBX` via static code analysis. Earlier in the function, `RBX` is populated:

```

140001eba    MOV     EBX, dword ptr [RAX + 0x88]
140001ec0    ADD     RBX, R14

```

In the `MOV` instruction at 140001eba `RAX` contains the virtual address of the mapped DLL's PE header (this is evident in the Decompile window if you retyped the argument, as discussed in the previous question). The 0x88 offset from the PE header is the Export Directory RVA:

	Member	Offset	Size	Value	Section
	Export Directory RVA	000001A8	Dword	000A6870	.rdata
	Export Directory Size	000001AC	Dword	00000044	
	Import Directory RVA	000001B0	Dword	000A68B4	.rdata

`R14` contains the image base of the mapped DLL, so the `ADD` instruction at 140001ec0 populates `RBX` with the virtual address of the Export Directory Table. The structure for this table is referred to as `IMAGE_EXPORT_DIRECTORY` (Ghidra refers to it as `IMAGE_DIRECTORY_ENTRY_EXPORT`).

The `MOV` instruction at 140001f73, which this question refers to, dereferences the 0x1c offset from the address in `RBX`. The 0x1c offset within the Export Directory Table is the location of the `AddressOfFunctions` field. This field contains the RVA of the array of addresses for exported functions, and this value is placed into `EDX`. `EDX` will therefore contain the RVA of the Export Address Table. Ghidra refers to this area as Export Function Pointers.

29. ☒ What do the remaining instructions (140001f73-140001f84) within `player.exe` accomplish, and how does this confirm our suspicions about this function's purpose? Rename the function. ☐

Note

Use a combination of both static and dynamic code analysis to answer this question.

Answer: The remaining instructions set up the return value for this function. Specifically, they identify the virtual address of the decoded DLL's single export named `start`, and place this address into `RAX`. We can rename `FUN_140001e80` to `get_export_address`.

Explanation: Based on the analysis we performed in the previous question, we know `EDX` contains the RVA of the Export Address Table. At 140001f7d, the image base value stored in `R14` is added to `EDX`, and the result is placed into `RAX`. Therefore, `RAX` contains the VA of the Export Address Table.

At 140001f81, the `MOV` instruction places the only RVA in the Export Address Table into `EAX` (`RCX` is zero in this instruction, and you can confirm this via debugging). As a result, `EAX` will store the RVA of the exported function named `start`.

Finally, at 140001f84 the `ADD` instruction adds the mapped DLL's image base to the RVA of the `start` function. This means, the return value of `FUN_140001e80` is the VA of the `start` function.

We can rename `FUN_140001e80` to `get_export_address`.

30. ☒ Return to the calling function `desteg` and view the next instruction at 140001493. What does this instruction accomplish? ☐

Answer: It executes the DLL's exported function `start`. The address of `start` is retrieved from the Export Address Table, as discussed in the previous question.

We'll stop our analysis of this sample here. With additional time and effort, you would identify that the remaining functions in `desteg` free loaded libraries and perform routine security checks.

Lab Objectives, Revisited

After completing this lab, you now have experience performing the following:

- Identifying code that checks for a valid Windows Executable.
- Identifying code that maps an executable into memory in preparation for execution.
- Identifying code that applies relocations, if needed.
- Identifying code that loads dependent DLLs and resolves APIs.
- Identifying code that updates section permissions in memory.
- Identifying code that locates the entry point for execution.

Lab 1.3: Analyzing Shellcode Execution

Background

In this lab, we will extend our knowledge of program execution to malicious shellcode execution. We will extract shellcode from memory and perform a combination of static and dynamic code analysis to understand the executable content. We will make use of x32dbg, WinDbg, and Ghidra to support our analysis.

Lab Objectives

- Extract shellcode from a running process.
- Perform static code analysis of shellcode.
- Analyze code that accesses the Process Environment Block (PEB).
- Identify code that resolves Windows APIs.
- Identify the hashing algorithm used to obfuscated imported DLLs and API names.
- Use WinDBG to interrogate various data structures and members.
- Experience an analysis workflow that involves Ghidra, x32dbg, and WinDbg.

Lab Preparation

First, extract `host32.exe` from `Malware/Section1/host32.zip` within both the Static and Dynamic VMs (password: `malware`).

Within the Dynamic VM, load `host32.exe` into x32dbg.

In addition, load `host32.exe` into WinDbg Preview. To accomplish this task, first double-click the WinDbgX shortcut on the desktop. Then, browse to **File > Start debugging > Launch executable** and navigate to `host32.exe`. If you do not see any code, it means you need to 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 rearrange windows within the interface, drag and drop each title bar as desired. To access the **Breakpoints** window, browse to the **View** tab and choose **Breakpoints**.

Note

We disabled ALSR for `host32.exe` so the virtual addresses in the solutions will match those in your environment.

Lab Questions

In the first part of this lab, we will use x32dbg to extract shellcode from memory when `host32.exe` is executed.

1. ? During execution, `host32.exe` allocates space for shellcode using the `VirtualAlloc` API. At what address within `host32.exe` is `VirtualAlloc` called?

 Note

Answer this question using x32dbg within the Dynamic VM.

2.  Identify the starting address for the newly allocated region, and dump the address into a Dump window. Continue stepping over the code (i.e, use the keyboard shortcut `F8`). At what address within `host32.exe` does a `CALL` instruction produce content in the Dump window that is likely shellcode? 
-

3.  Dump the shellcode to disk within the Dynamic VM. Then, copy and paste the dumped shellcode file to the Static VM and load it into Ghidra. Specifically, add the file to the `Section1` project, disassemble all bytes, and perform the auto-analysis. 

 Notes

- Do not exit the debugger in the Dynamic VM.
 - Give the dumped file a descriptive name such as `host32_sc.bin`.
 - When loading the dumped shellcode into Ghidra within the Static VM, choose the Language listed as **x86-default-32-little-Visual Studio** (see Explanation for a screenshot).
-

4.  At offset `00000001` within Ghidra, we see the instruction `CALL FUN_0000008f`. Jump to `0000008f` and observe four `PUSH` instructions leading up to a `CALL EBP` at offset `000000a0`. Highlight the `EBP` operand with a single-click, and scroll down to observe other `CALL EBP` instructions throughout the shellcode. Based on a brief review of the argument pushed onto the stack before each `CALL`, what can you say about the likely purpose of these function calls? 
-

5.  Return to the instruction `CALL EBP` at offset `000000a0` within Ghidra. Relying on static code analysis only, at what address is the function that will be called? 

 Note

This question asks for the offset of the function within Ghidra, *not* the address of the function if you debugged the shellcode.

6. ? Let's review the function beginning at offset 00000006. First, observe the instruction at 0000000b. What data structure is likely referenced by the source operand? ^

We just discussed the structure referenced by the source operand in the MOV instruction at 0000000b. In the subsequent instructions, we see other offsets that are likely related to this structure and its members. To accurately identify what structures and members the upcoming operands point to, we will use WinDbg within the Dynamic VM. However, keep Ghidra open within the Static VM as it continues to be our primary tool for static code analysis, and we will use its interface to add comments and document our work.

7. ? You should already have `host32.exe` loaded into WinDbg within the Dynamic VM. Set a breakpoint and run the program so that it arrives at the entry point. ^

Important

You've reached **Checkpoint #1** in this exercise.

8. ? Identify the starting address of the memory region allocated via VirtualAlloc. ^

Note

This question asks you to accomplish the same task we performed in x32dbg, but this time using WinDbg Preview.

9. ? Within WinDbg, set an *access breakpoint* on the starting address of the allocated memory region. Then, run the program so we arrive at the first instruction of the shellcode. ^

You should now be looking at the shellcode within WinDbg.

Warning

When referring to an instruction address in WinDbg, this exercise will use offsets instead of addresses. This is because the starting address of the region allocated will vary.

As we discussed earlier, the MOV instruction at offset 0000000b (from the beginning of the allocated memory region) includes `fs:[edx+30h]` in the source operand. We believe this source operand references the address of the PEB-let's confirm this.

Important

You've reached **Checkpoint #2** in this exercise.

15.  The `InMemoryOrderModuleList` member is comprised of two pointers, and the first points to the head of a double-linked list of other `LIST_ENTRY` structures. These `LIST_ENTRY` structures reside in yet another larger data structure. What type of data structure is the larger structure? 

Note

To answer this question, it may be helpful to review the slides for this module. Specifically, see the slide with a title that begins with "Each Module List Contains Two Pointers"

16.  The pointers that comprise the `InMemoryOrderModuleList` `LIST_ENTRY` structure point to a member within `LDR_DATA_TABLE_ENTRY`. What is the name of the member within `LDR_DATA_TABLE_ENTRY`? 

17.  When execution arrives at offset 00000015, `EDX` will contain the address of the `InMemoryOrderLinks` member within the first `LDR_DATA_TABLE_ENTRY` structure. With this in mind, how would you describe the value placed into `ESI` in the instruction at offset 00000015? 

 Note

This question is asking for a description of the value placed into ESI, not the exact value. As a reminder, this is the LDR_DATA_TABLE_ENTRY

```
0:000> dt -r1 _LDR_DATA_TABLE_ENTRY
ntdll!_LDR_DATA_TABLE_ENTRY
+0x000 InLoadOrderLinks : _LIST_ENTRY
+0x000 Flink             : Ptr32 _LIST_ENTRY
+0x004 Blink             : Ptr32 _LIST_ENTRY
+0x008 InMemoryOrderLinks : _LIST_ENTRY
+0x000 Flink             : Ptr32 _LIST_ENTRY
+0x004 Blink             : Ptr32 _LIST_ENTRY
+0x010 InInitializationOrderLinks : _LIST_ENTRY
+0x000 Flink             : Ptr32 _LIST_ENTRY
+0x004 Blink             : Ptr32 _LIST_ENTRY
+0x018 DllBase           : Ptr32 Void
+0x01c EntryPoint        : Ptr32 Void
+0x020 SizeOfImage       : Uint4B
+0x024 FullDllName       : _UNICODE_STRING
+0x000 Length            : Uint2B
+0x002 MaximumLength     : Uint2B
+0x004 Buffer             : Ptr32 Wchar
+0x02c BaseDllName       : _UNICODE_STRING
+0x000 Length            : Uint2B
+0x002 MaximumLength     : Uint2B
+0x004 Buffer             : Ptr32 Wchar
+0x034 FlagGroup         : [4] UChar
```

structure:

18.  Within WinDbg, allow the MOV instruction at offset 00000015 to execute. Confirm that the first module name resides at the instruction in ESI. 
19.  Within Ghidra in the Static VM, review the loop beginning at 0000001e. What is the likely purpose of this loop? After your analysis is complete, use Ghidra to rename the label associated with this loop. 

Notes

- The LODSB (load single byte) operation at offset 00000020 places one byte (in this case, a character) from the buffer in ESI into AL.
- You can ignore the CMP, JL, and SUB instructions between offsets 00000021 and 00000025 when answering this question. These instructions simply convert each character placed into AL to uppercase.
- Static code analysis should be sufficient, but you can debug the code with x32dbg or WinDbg if you prefer.

Before moving forward, observe the instruction `PUSH EDI` at offset 0000002f. This pushes the output from the loop just discussed onto the stack. Make a comment in Ghidra, and we will return to this observation later.

20. ? Let us continue reviewing key components of the shellcode. At offset 00000030 within Ghidra, we have the instruction `MOV EDX, dword ptr [EDX + 0x10]`. Before this instruction is executed, EDX still contains the address of the `InMemoryOrderLinks` member within an `LDR_DATA_TABLE_ENTRY` structure. With this in mind, describe the value placed into EDX in this instruction.

Note

This question is asking for a description of the value placed into ESI, not the exact value. As a reminder, this is the `LDR_DATA_TABLE_ENTRY`

```
0:000> dt -r1 _LDR_DATA_TABLE_ENTRY
ntdll!_LDR_DATA_TABLE_ENTRY
+0x000 InLoadOrderLinks : _LIST_ENTRY
+0x000 Flink             : Ptr32 _LIST_ENTRY
+0x004 Blink             : Ptr32 _LIST_ENTRY
+0x008 InMemoryOrderLinks : _LIST_ENTRY
+0x000 Flink             : Ptr32 _LIST_ENTRY
+0x004 Blink             : Ptr32 _LIST_ENTRY
+0x010 InInitializationOrderLinks : _LIST_ENTRY
+0x000 Flink             : Ptr32 _LIST_ENTRY
+0x004 Blink             : Ptr32 _LIST_ENTRY
+0x018 DllBase           : Ptr32 Void
+0x01c EntryPoint         : Ptr32 Void
+0x020 SizeOfImage        : Uint4B
+0x024 FullDllName        : _UNICODE_STRING
+0x000 Length             : Uint2B
+0x002 MaximumLength      : Uint2B
+0x004 Buffer             : Ptr32 Wchar
+0x02c BaseDllName        : _UNICODE_STRING
+0x000 Length             : Uint2B
+0x002 MaximumLength      : Uint2B
+0x004 Buffer             : Ptr32 Wchar
+0x034 FlagGroup          : [4] UChar
```

structure:

28. ? Review all remaining calls to resolved functions. How would you summarize the purpose of this shellcode? Again, use a debugger to answer this question.

Lab Solutions

In the first part of this lab, we will use x32dbg to extract shellcode from memory when `host32.exe` is executed.

1. ✓ During execution, `host32.exe` allocates space for shellcode using the `VirtualAlloc` API. At what address within `host32.exe` is `VirtualAlloc` called?

Note

Answer this question using x32dbg within the Dynamic VM.

Answer: `VirtualAlloc` is called at 00408453.

Explanation: Within x32dbg, set a breakpoint on `VirtualAlloc` by typing `bp VirtualAlloc` in the command window. Then, run the program. When the breakpoint is hit, allow the function to **Debug > Execute till return** and step into the calling function. Scroll up and you will observe a `CALL ECX` at 00408453. This instruction calls `VirtualAlloc`.

2. ✓ Identify the starting address for the newly allocated region, and dump the address into a Dump window. Continue stepping over the code (i.e, use the keyboard shortcut `F8`). At what address within `host32.exe` does a `CALL` instruction produce content in the Dump window that is likely shellcode?

Answer: At 00408460, the `CALL` instruction places shellcode in the allocated memory region.

Explanation: After `VirtualAlloc` is called at 00408453, the starting address for the newly allocated memory region is stored in `EAX`. **Right-click > Follow in Dump** on the register value to observe the memory region and continue stepping over the code. At 00408460, a function is called that produces shellcode in the dump window:

 Dump 1	 Dump 2	 Dump 3	 Dump 4	 Dump 5	 Watch 1	[x=] Locals	 Struct										
Address	Hex																ASCII
001E0000	FC	E8	89	00	00	00	60	89	E5	31	D2	64	8B	52	30	8B	űè...`..ălòd.R0.
001E0010	52	0C	8B	52	14	8B	72	28	0F	B7	4A	26	31	FF	31	C0	R..R..r(..J&1y1A
001E0020	AC	3C	61	7C	02	2C	20	C1	CF	0D	01	C7	E2	F0	52	57	~<a ., ÁĬ..ÇãðRW
001E0030	8B	52	10	8B	42	3C	01	D0	8B	40	78	85	C0	74	4A	01	.R..B<.Đ.@x.ÀtJ.
001E0040	D0	50	8B	48	18	8B	58	20	01	D3	E3	3C	49	8B	34	8B	ðP.H..X..óã<I.4.
001E0050	01	D6	31	FF	31	C0	AC	C1	CF	0D	01	C7	38	E0	75	F4	.Öÿ1A-ÄĬ..Ç8âuô
001E0060	03	7D	F8	3B	7D	24	75	E2	58	8B	58	24	01	D3	66	8B	.}ø;}\$uâx.X\$.óf.
001E0070	0C	4B	8B	58	1C	01	D3	8B	04	8B	01	D0	89	44	24	24	.K.X..Ó....Đ.D\$\$

Notice the opcodes `FC E8`, which represent the instructions `CLD` and `CALL`. This instruction is commonly seen at the beginning of shellcode.

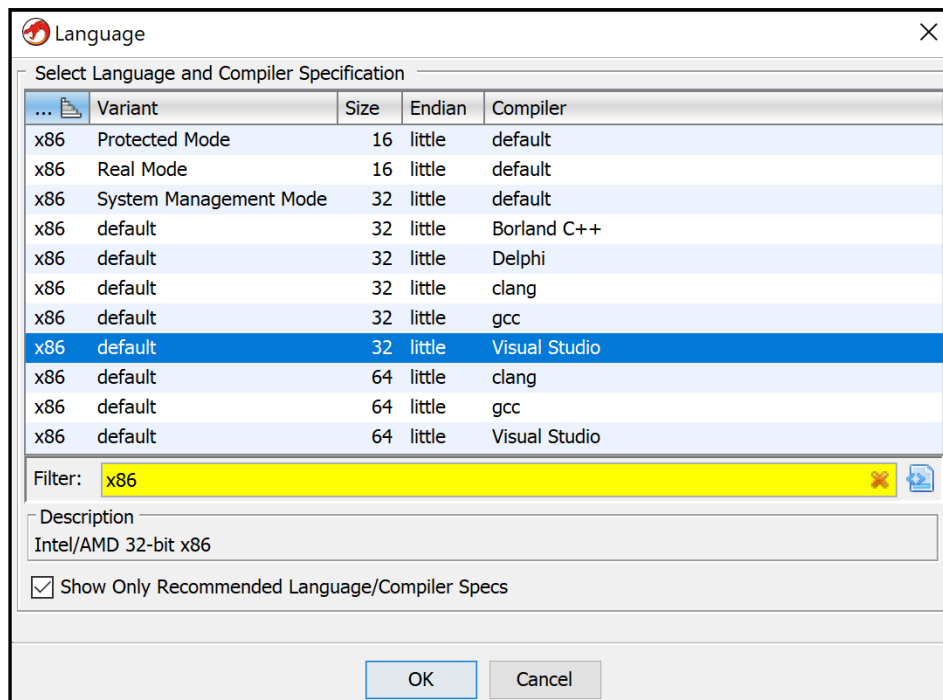
3. ✓ Dump the shellcode to disk within the Dynamic VM. Then, copy and paste the dumped shellcode file to the Static VM and load it into Ghidra. Specifically, add the file to the `Section1` project, disassemble all bytes, and perform the auto-analysis.

Notes

- Do not exit the debugger in the Dynamic VM.
- Give the dumped file a descriptive name such as `host32_sc.bin`.
- When loading the dumped shellcode into Ghidra within the Static VM, choose the Language listed as **x86-default-32-little-Visual Studio** (see Explanation for a screenshot).

Explanation: To dump the shellcode, right-click on the first byte in the dump window and choose **Follow in Memory Map**. Then, right-click on the memory region highlighted in gray and choose **Dump Memory To File**. Give the dumped file a descriptive name such as `host32_sc.bin`.

Copy and paste `host32_sc.bin` from the Dynamic VM to the Static VM. Load the file into the **Section1** project within Ghidra. When prompted to start the import, choose the following language:



Then, double-click file in the project window and choose *not* to analyze the file. Next, click on the first byte of shellcode and type `D` on the keyboard to disassembly the bytes. Finally, browse to **Analysis > Auto Analyze** and click **Analyze** to begin processing.

4. ✓ At offset 00000001 within Ghidra, we see the instruction `CALL FUN_0000008f`. Jump to 0000008f and observe four PUSH instructions leading up to a `CALL EBP` at offset 000000a0. Highlight the EBP operand with a single-click, and scroll down to observe other `CALL EBP` instructions throughout the shellcode. Based on a brief review of the argument pushed onto the stack before each `CALL`, what can you say about the likely purpose of these function calls?

Answer: In all cases, a `CALL EBP` instruction is preceded by a PUSH instruction that places a hexadecimal value onto the stack. When analyzing shellcode, this is a good indication that the function called is responsible for resolving APIs based on a provided hash.

5. ✓ Return to the instruction `CALL EBP` at offset 000000a0 within Ghidra. Relying on static code analysis only, at what address is the function that will be called?

Note

This question asks for the offset of the function within Ghidra, *not* the address of the function if you debugged the shellcode.

Answer: The CALL instruction at offset 000000a0 will execute the function beginning at offset 00000006.

Explanation: To investigate what value EBP will contain when the instruction `CALL EBP` is executed, we look for other references to EBP. Several instructions earlier, observe the `POP EBP` at 0000008f. Since this is the first instruction in function `FUN_0000008f`, the value popped into EBP will be the return value pushed onto the stack when the function is called. If we return to the beginning of the shellcode, we can see that the offset of the instruction *after* the CALL to `FUN_0000008f` is 00000006.

6. ✓ **Let's review the function beginning at offset 00000006. First, observe the instruction at 0000000b. What data structure is likely referenced by the source operand?**

Answer: The Process Environment Block (PEB).

Explanation: At address 00000009, EDX is zeroed out. At 0000000b, `FS:[EDX + 0x30]` refers to FS:[30]. This is the location of the pointer to the PEB in 32-bit code.

We just discussed the structure referenced by the source operand in the MOV instruction at 0000000b. In the subsequent instructions, we see other offsets that are likely related to this structure and its members. To accurately identify what structures and members the upcoming operands point to, we will use WinDbg within the Dynamic VM. However, keep Ghidra open within the Static VM as it continues to be our primary tool for static code analysis, and we will use its interface to add comments and document our work.

7. ✓ **You should already have `host32.exe` loaded into WinDbg within the Dynamic VM. Set a breakpoint and run the program so that it arrives at the entry point.**

Explanation: In the command window, type `bp $entry`. As a reminder, `$entry` is a pseudo-register (<https://for710.com/pseudo-register>). Then, run the program by clicking **Go** on the top left of the WinDBG GUI, or type the `F5` key. Alternatively, type `g` in the command window and press `Enter`. You should arrive at address 004094E0.

Important

You've reached **Checkpoint #1** in this exercise.

8. ✓ **Within WinDbg, identify the starting address of the memory region allocated via `VirtualAlloc`.**

Note

This question asks you to accomplish the same task we performed in x32dbg, but this time using WinDbg Preview.

Explanation: There are multiple approaches to accomplishing this task.

One option is to use the command `bp 00408453` to set a breakpoint where `VirtualAlloc` is called (we identified this address in the first question of this lab). Stepping over this instruction reveals the appropriate return value in EAX.

Alternatively, you could set a breakpoint on `VirtualAlloc` using the command `bp KERNEL32!VirtualAllocStub` and then run the program. When the breakpoint is hit, click the **Step Out** button located on the top-left of the WinDbg Preview GUI or type `gu` to return to the calling function.

Note

- As mentioned in the **Warning** above "0000000b" refers to the offset from the beginning of the allocated region.
- Once you confirm the addresses match, write a comment next to the instruction in Ghidra. Remember, we are using WinDBG to investigate structures in memory, but Ghidra is still our main interface for static code analysis.

Explanation: Within the WinDbg GUI, click **Step Over** six times until the MOV instruction at offset `0b` is executed. Then, print the value stored within EDX with the command `r edx`. This should match your output from the previous question.

12. ✓ Within WinDBG, you should now be at offset 0000000f. Type a command to print out the contents of the PEB, including its members and values.

Note

Your command should include the automatic pseudo register that corresponds to the PEB or the PEB's address.

Explanation: Type the command `dt ntdll!_PEB @$peb`. You can also simply type `dt _PEB @$peb`, though the former command is more precise. In addition, you could use the PEB address output from the previous question in the following format: `dt _PEB <PEB address>`, where `<PEB address>` is the hexadecimal address of the PEB. This outputs the PEB structure and values without using an automatic pseudo register.

Your output should resemble the following:

```
0:000> dt ntdll!_PEB @$peb
+0x000 InheritedAddressSpace : 0 ''
+0x001 ReadImageFileExecOptions : 0 ''
+0x002 BeingDebugged : 0x1 ''
+0x003 BitField : 0 ''
+0x003 ImageUsesLargePages : 0y0
+0x003 IsProtectedProcess : 0y0
+0x003 IsImageDynamicallyRelocated : 0y0
+0x003 SkipPatchingUser32Forwarders : 0y0
+0x003 IsPackagedProcess : 0y0
+0x003 IsAppContainer : 0y0
+0x003 IsProtectedProcessLight : 0y0
+0x003 IsLongPathAwareProcess : 0y0
+0x004 Mutant : 0xffffffff Void
+0x008 ImageBaseAddress : 0x00400000 Void
+0x00c Ldr : 0x77007be0 _PEB_LDR_DATA
+0x010 ProcessParameters : 0x02651bf8 _RTL_USER_PROCESS_PARAMETERS
```

13. ✓ Review the MOV instruction at offset 0000000f in the shellcode (you can use Ghidra or WinDbg to view this code). What member within the PEB is referenced in the source operand (i.e., `EDX + 0xc`), and what type of data structure does the value of that member point to? Review the output from the previous question to answer this question, and write an appropriate comment next to the instruction in Ghidra.

Answer: The source operand references the `Ldr` member, and this member's value is a pointer to a data structure of type `PEB_LDR_DATA`. The MOV instruction places the address of the `PEB_LDR_DATA` data structure into EDX.

Explanation: The source operand in the instruction at 0000000f is `dword ptr [EDX + 0xc]`. EDX contains the address of the PEB. View the PEB structure output from the previous question and find the member located at Offset `0xc` - it is `Ldr`. The `Ldr` member's value is a pointer to a structure of type `PEB_LDR_DATA` (see <https://for710.com/peb>). This structure has information about a process's loaded modules.

Using WinDbg, review the MOV instruction at offset 00000012 and answer the following three questions.

14. ✓ What member within the `PEB_LDR_DATA` structure does the source operand reference (i.e., `edx+14h`), and what type of structure is this member? Make an appropriate comment for this instruction in Ghidra.

Answer: `InMemoryOrderModuleList` is a structure of type `LIST_ENTRY`.

Explanation: `EDX` contains the address of the `PEB_LDR_DATA` structure, so we need to understand what member is located at its `0x14` offset. To view the appropriate `PEB_LDR_DATA` structure, you can click on the `Ldr` link in the previously generated PEB output. Alternatively, you can observe the `Ldr` value in the previously generated PEB output (i.e., the address of the `PEB_LDR_DATA` structure) and type a command using the format `dt ntdll!_PEB_LDR_DATA <address>`. For example, using the values in the PEB output shown earlier, the command is `dt ntdll!_PEB_LDR_DATA 0x77007be0`. Whichever approach you take, printing the `PEB_LDR_DATA` structure should display similar output:

```
0:000> dt ntdll!_PEB_LDR_DATA 0x77007be0
+0x000 Length           : 0x30
+0x004 Initialized      : 0x1 ''
+0x008 SsHandle         : (null)
+0x00c InLoadOrderModuleList : _LIST_ENTRY [ 0x26534a0 - 0x265d8f0 ]
+0x014 InMemoryOrderModuleList : _LIST_ENTRY [ 0x26534a8 - 0x265d8f8 ]
+0x01c InInitializationOrderModuleList : _LIST_ENTRY [ 0x26533a8 - 0x265a268 ]
+0x024 EntryInProgress  : (null)
+0x028 ShutdownInProgress : 0 ''
+0x02c ShutdownThreadId : (null)
```

At the `0x14` output we find the `InMemoryOrderModuleList` member (i.e., `PEB->Ldr->InMemoryOrderModuleList`). As shown in the WinDbg output, this member is of type `LIST_ENTRY`.

Important

You've reached **Checkpoint #2** in this exercise.

15. ✓ The `InMemoryOrderModuleList` member is comprised of two pointers, and the first points to the head of a double-linked list of other `LIST_ENTRY` structures. These `LIST_ENTRY` structures reside in yet another larger data structure. What type of data structure is the larger structure?

Note

To answer this question, it may be helpful to review the slides for this module. Specifically, see the slide with a title that begins with "Each Module List Contains Two Pointers".

Answer: The pointers that comprise the `InMemoryOrderModuleList` member point to other `LIST_ENTRY` structures within a data structure of type `LDR_DATA_TABLE_ENTRY`.

16. ✓ The pointers that comprise the `InMemoryOrderModuleList` `LIST_ENTRY` structure point to a member within `LDR_DATA_TABLE_ENTRY`. What is the name of the member within `LDR_DATA_TABLE_ENTRY`?

Answer: Each pointer in the `InMemoryOrderModuleList` `LIST_ENTRY` structure points to an `InMemoryOrderLinks` member within a `LDR_DATA_TABLE_ENTRY` structure. As a reminder, this member is a structure of type `LIST_ENTRY`.

19.

- ✓ Within Ghidra in the Static VM, review the loop beginning at 0000001e. What is the likely purpose of this loop? After your analysis is complete, use Ghidra to rename the label associated with this loop.

Notes

- The LODSB (load single byte) operation at offset 00000020 places one byte (in this case, a character) from the buffer in ESI into AL.
- You can ignore the CMP, JL, and SUB instructions between offsets 00000021 and 00000025 when answering this question. These instructions simply convert each character placed into AL to uppercase.
- Static code analysis should be sufficient, but you can debug the code with x32dbg or WinDbg if you prefer.

Answer: This loop calculates the ROR 13 hash of a module name. Within Ghidra, we could rename the label at offset 0000001e to `hash_filename`.

Explanation: At offset 00000020, each byte of a module name is placed into AL (i.e., EAX). The ROR instruction at offset 00000027 rotates the existing value of EDI by 13 bits (EDI is initialized to zero at offset 0000001c). At 0000002a a single character from the module name (previously placed into EAX at offset 00000020) is added to EDI. This process continues for each character in the module file name.

Before moving forward, observe the instruction `PUSH EDI` at offset 0000002f. This pushes the output from the loop just discussed onto the stack. Make a comment in Ghidra, and we will return to this observation later.

20.

- ✓ Let us continue reviewing key components of the shellcode. At offset 00000030 within Ghidra, we have the instruction `MOV EDX, dword ptr [EDX + 0x10]`. Before this instruction is executed, EDX still contains the address of the `InMemoryOrderLinks` member within an `LDR_DATA_TABLE_ENTRY` structure. With this in mind, describe the value placed into EDX in this instruction.

Note

This question is asking for a description of the value placed into ESI, not the exact value. As a reminder, this is the LDR_DATA_TABLE_ENTRY

```
0:000> dt -r1 _LDR_DATA_TABLE_ENTRY
ntdll!_LDR_DATA_TABLE_ENTRY
+0x000 InLoadOrderLinks : _LIST_ENTRY
+0x000 Flink             : Ptr32 _LIST_ENTRY
+0x004 Blink             : Ptr32 _LIST_ENTRY
+0x008 InMemoryOrderLinks : _LIST_ENTRY
+0x000 Flink             : Ptr32 _LIST_ENTRY
+0x004 Blink             : Ptr32 _LIST_ENTRY
+0x010 InInitializationOrderLinks : _LIST_ENTRY
+0x000 Flink             : Ptr32 _LIST_ENTRY
+0x004 Blink             : Ptr32 _LIST_ENTRY
+0x018 DllBase           : Ptr32 Void
+0x01c EntryPoint        : Ptr32 Void
+0x020 SizeOfImage       : Uint4B
+0x024 FullDllName       : _UNICODE_STRING
+0x000 Length            : Uint2B
+0x002 MaximumLength     : Uint2B
+0x004 Buffer             : Ptr32 Wchar
+0x02c BaseDllName       : _UNICODE_STRING
+0x000 Length            : Uint2B
+0x002 MaximumLength     : Uint2B
+0x004 Buffer             : Ptr32 Wchar
+0x034 FlagGroup         : [4] UChar
```

structure:

Answer: The MOV instruction at offset 00000030 places the base address of a loaded module into EDX.

Explanation: EDX contains the address of the `InMemoryOrderLinks` member, which is at offset 0x8 within an `LDR_DATA_TABLE_ENTRY` data structure. Adding 0x10 to this offset equals 0x18, which brings us to the `DllBase` member. This member's value specifies the base address of the module (i.e., the starting address of the executable in memory).

Important

You've reached **Checkpoint #3** in this exercise.

Within Ghidra, view the instructions at offsets 00000033 through 00000038:

```
00000030    MOV     EDX, dword ptr [EDX + 0x10]
00000033    MOV     EAX, dword ptr [EDX + 0x3c]
00000036    ADD     EAX, EDX
00000038    MOV     EAX, dword ptr [EAX + 0x78]
```

21.

- ✓ At offset 00000033, consider the contents of EDX discussed in the previous question. With this in mind, what field within a 32-bit Windows executable header is dereferenced in the source operand of the MOV instruction at offset 00000038? Why would shellcode need to access this field? After performing your analysis, insert appropriate comments within Ghidra for each instruction.

Notes

- Static analysis should be sufficient to answer this question.
- For a reminder of what fields appear at various offsets within a 32-bit executable, open any 32-bit program in CFF explorer (e.g., `host32.exe`)

Answer: The MOV instruction at offset 00000038 places the Export Directory RVA of a loaded module into EAX. Shellcode commonly accesses the export directory of loaded modules to iterate over the module's exported functions and resolve APIs required by the shellcode.

Explanation: The MOV instruction at offset 00000033 dereferences `EDX + 0x3c` and places the value into EAX. Based on our analysis in the previous question, we know EDX contains the base address of a loaded module. Within a 32-bit Windows executable, the `e_lfanew` field is located at offset 0x3c. Dereferencing this value places the RVA of the PE header into EAX.

The ADD instruction at offset 00000036 adds the base address of a module to the PE header RVA to place the VA of the PE header into EAX.

The MOV instruction at offset 00000038 adds 0x78 to the starting address of the PE header. To identify what field this refers to, we can open `host32.exe` (or any 32-bit executable) within CFF Explorer. Within `host32.exe`, the PE header begins at offset 0x78. Adding 0x78 to this value equals 0xF0, and browsing to this offset within CFF Explorer leads us to the Export Directory RVA.

Observe that the ADD instruction at offset 0000003f adds the Export Directory RVA (first operand) to the base address of the module (second operand) to calculate the Export Directory VA. At 00000041 this VA is pushed onto the stack. Make a comment in Ghidra with this information. We will revisit this observation later.

22.

- ✓ Review the loop beginning at offset 00000054 in the shellcode. What is its likely purpose? After your analysis is complete, use Ghidra to rename the label associated with this loop.

Notes

- Consider using x32dbg to set a breakpoint on the LODSB instruction at offset 00000056. When the breakpoint is hit, view the address contained in ESI in the dump window (as a reminder, ESI specifies the address of the byte to load). For additional context, right-click on a byte in the dump window and **Follow in Memory Map**.
- To arrive at the shellcode within x32dbg, allow the VirtualAlloc API to execute and set a hardware execution breakpoint on the first byte of the newly allocated memory.
- You do not need to evaluate each instruction in the loop to determine the purpose of this code with high confidence.

Answer: This loop calculates the ROR 13 hash of an exported function name from a loaded module and places the resulting value in EDI. Within Ghidra, we could rename the label at offset 00000054 to `hash_exportname`.

Explanation: This loop includes ROR and ADD operations that are identical to instructions within the loop at offset 0000001e. We concluded that earlier loop hashes the file name of a loaded module. This suggests a similar hash algorithm is being used in the current loop under analysis, but what content is this loop hashing? Our analysis from the previous question suggests it is probably hashing a module's exported function names. Let's confirm this.

To investigate, we will debug the shellcode in x32dbg. First, restart the `host32.exe` process. Then, allow the memory region to be allocated via VirtualAlloc. When VirtualAlloc returns, dump the address in EAX to the dump window to view the allocated memory region. Next, set a hardware execution breakpoint on the first byte of the allocated region (i.e., **right-click > Breakpoint > Hardware, Execute**). Finally, run the program and you should arrive at the beginning of the shellcode.

Next, set a breakpoint on the LODSB instruction at offset 00000056. As a reminder, the LODSB instruction loads one byte from ESI into EAX. Since EAX is manipulated by the ADD instruction with each iteration of the loop, ESI likely contains the content that is hashed.

Within x32dbg, continue running the shellcode. Each time execution hits the LODSB instruction at 00000056, ESI points to a function name. If we dump the address in ESI to the dump window, you will observe multiple function names nearby. For additional context, we can pivot from the dump window to the memory map, and it will be clear these function names are stored in a loaded DLL-specifically, the function names list each module's exported functions.

23. ✓ What does the ADD instruction at offset 00000060 accomplish?

 Note

Recall the instruction `PUSH EDI` at offset 0000002f, which pushed the output from the loop at offset 0000001e onto the stack.

Answer: It adds the hash of a module name (right operand) to the hash of an exported function name (left operand).

Explanation: We're focused on the instruction `ADD EDI, dword ptr [EBP + -0x8]` at offset 00000060. EDI contains a hashed exported function name, as discussed in the previous question. The second operand references the hashed value of a module file name that was pushed onto the stack at offset 0000002f. The ADD instruction calculates the sum of both hashes and places the result in EDI.

24. ✓ Beginning at 00000063, we see CMP and JNZ instructions. What is evaluated, and under what conditions is the jump taken?

 Notes

- Recall that, in 32-bit code, EBP *plus* a value often refers to an argument.
- Consider using x32dbg to investigate the CMP and JNZ instructions.

Answer: The CMP instruction evaluates if the hash provided as an argument (the first argument) matches the calculated hash. The jump is taken if there is *not* a match.

Explanation: First, let's review the instruction `CMP EDI, dword ptr [EBP + 0x24]` at 00000063. We know EDI contains the sum of a module and export function hashes. The pointer references EBP *plus* a value, which often refers to an argument in 32-bit code. If we highlight EBP and observe other references within the current function, we will find only one in the function prologue. This means EBP is being used as a base pointer, supporting our theory that the second operand in the CMP instruction references an argument passed to the function beginning at offset 00000006.

To confirm our theory, first set a breakpoint on the CMP instruction at offset 00000063 within x32dbg. Then, run the program and view what the second operand points to:

	001E0060	add edi,dword ptr ss:[ebp-8]
	001E0063	cmp edi,dword ptr ss:[ebp+24]
	001E0066	jne 1E004A
	001E0068	pop eax
	001E0069	mov ebx,dword ptr ds:[eax+24]
	001E006C	add ebx,edx
<		
edi=A18D83A9		
dword ptr [ebp+24]=[0019FEBF]=726774C		

As we can see in the information window below the disassembly, the second operand points to 726774C. If we look at the first CALL to the function beginning at offset 00000006, we can see this matches the first argument passed (i.e., the last value pushed before the CALL):

- Analyze code that accesses the Process Environment Block (PEB).
- Identify code that resolves Windows APIs.
- Identify the hashing algorithm used to obfuscated imported DLLs and API names.
- Use WinDBG to interrogate various data structures and members.
- Use an analysis workflow that involves Ghidra, x32dbg, and WinDbg.

Lab 2.1: Encryption Essentials: Quiz

Background

In this brief lab, the goal is to reinforce and recall essential information necessary to understand encryption in malware.

Lab Objectives

- Confirm your understanding of basic cryptography terminology.
- Differentiate symmetric and asymmetric cryptography.
- Compare cryptography modes of operation.
- Differentiate similar ciphers.
- Identify Microsoft APIs commonly used for encryption and decryption.

Lab Preparation

None required.

Lab Questions

1. ? Describe the primary differences between asymmetric and symmetric-key algorithms. ^

2. ? How would you characterize the difference between a block cipher and stream cipher? ^

3. ? AES uses an S-box. What does the "S" in S-box refer to? ^

4.  When considering block ciphers, there are three modes of operation we discussed. What is the most basic mode of operation? 
-
5.  How would you characterize the difference between the ECB and CBC modes of operation? 
-
-
-
6.  Salsa and ChaCha algorithms consist of ARX operations. What do the A, R, and X represent? 
-
7.  What constant is associated with Salsa and ChaCha when a 32-bit key is used? 
-
8.  What is the most obvious visual difference between the initial state of ChaCha vs. the initial state of Salsa? 
-
-
9.  Curve25519 is an Elliptic Curve Cryptography (ECC) curve that generates a shared key. With two users, User A and User B, and their corresponding public and private keys, which of the below combination of keys will produce the same shared secret when provided to the curve algorithm? (Choose 2.) 
- Curve(PrivateA, PublicB)
- Curve(PrivateA, PrivateB)
- Curve(PublicA, PublicB)
- Curve(PrivateB, PublicA)
-
-
10.  If a program uses the Microsoft Crypto API, which of the following APIs is typically called first? 
- CryptImportKey

CryptAcquireContext

CryptGenRandom

CryptGenKey

CryptEncrypt

CryptExportKey

11. ? Given a CALL to CryptImportKey during a debugging session, how could you determine the algorithm associated with the imported key? ^

12. ? Why would a program that performs encryption call the CryptGenRandom API? ^

Lab Solutions

1. ✓ Describe the primary differences between asymmetric and symmetric-key algorithms. v

Answer: Symmetric key algorithms use the same key for encryption and decryption, while asymmetric key algorithms use different keys for encryption and decryption. Symmetric encryption is generally faster from a performance perspective.

2. ✓ How would you characterize the difference between a block cipher and stream cipher? v

Answer: Block ciphers encrypt a block of plaintext data of a fixed length and output an encrypted block of data. Stream ciphers operate against individual bytes of data, typically using the XOR operation.

3. ✓ AES uses an S-box. What does the "S" in S-box refer to? v

Answer: Substitution box. An S-box helps substitute bytes and contributes to the scrambling of data associated with an encryption or decryption algorithm.

4. ✓ When considering block ciphers, there are three modes of operation we discussed. What is the most basic mode of operation? v

Answer: Electronic Code Book (ECB).

5. ✓ **How would you characterize the difference between the ECB and CBC modes of operation?**

Answer: CBC (Cipher Block Chaining) interconnects each encrypted block with the next encrypted block. In contrast, there is no relationship between one encrypted block and the next block when using ECB mode.

6. ✓ **Salsa and ChaCha algorithms consist of ARX operations. What do the A, R, and X represent?**

Answer: Add, Rotate, and XOR.

7. ✓ **What constant is associated with Salsa and ChaCha when a 32-bit key is used?**

Answer: expand 32-byte k

8. ✓ **What is the most obvious visual difference between the initial state of ChaCha vs. the initial state of Salsa?**

Answer: In ChaCha, the "expand 32-byte k" constant characters appear consecutively. In Salsa, that same constant is split up into four four-byte chunks and they are separated from one another (i.e., the constant is split along the top-left to bottom-right diagonal).

9. ✓ **Curve25519 is an Elliptic Curve Cryptography (ECC) curve that generates a shared key. With two users, User A and User B, and their corresponding public and private keys, which combination of keys will produce the same shared secret when provided to the curve algorithm?**

Answer: Curve(PrivateA, PublicB) and Curve(PrivateB, PublicA). In other words, both combinations of one user's private key and the other user's public key will produce the same shared key.

10. ✓ **If a program uses the Microsoft Crypto API, which of the following APIs is typically called first?**

Answer: CryptAcquireContext

11. ✓ **Given a CALL to CryptImportKey during a debugging session, how could you determine the algorithm associated with the imported key?**

Answer: The second argument passed to CryptImportKey points to a BLOBHEADER structure. One member of this structure is ALG_ID, and it specifies the algorithm associated with the key BLOB. For a comprehensive list of algorithm IDs, see <https://for710.com/algid>.

12. ✓ **Why would a program that performs encryption call the CryptGenRandom API?**

Answer: CryptGenRandom generates a specified number of random bytes. In programs that perform encryption or decryption, random bytes are often required to generate a key, nonce, or initialization vector (IV).

Lab Objectives, Revisited

This lab reinforced the following knowledge:

- Confirm your understanding of basic cryptography terminology.
- Differentiate symmetric and asymmetric cryptography.
- Compare cryptography modes of operation.
- Differentiate similar ciphers.
- Identify Microsoft APIs commonly used for encryption and decryption.

Lab 2.2: Identifying File Encryption and Key Protection in Ransomware

Background

In this lab, we will analyze ransomware and evaluate how it performs file encryption and key protection.

Lab Objectives

- Identify specific ciphers used in a ransomware sample.
- Determine the purpose of an identified cipher (e.g., file encryption or key protection).
- Recognize Windows APIs that facilitate encryption.
- Summarize key aspects of how ransomware performs file encryption.

Lab Preparation

First, extract `boot.dll` from `Malware\Section2\boot.zip` within both the Static and Dynamic 710 VMs (password: `malware`). Place the extracted DLLs on each desktop.

Within the Static VM, load `boot.dll` into Ghidra. When prompted, create a new project named `710_Section2`. Process the file and initiate auto-analysis, but there is no need to choose `WindowsPE x86 Propagate External Parameters` because the DLL is 64-bit. Ensure that the `FindCrypt` analyzer is checked.

Within the Dynamic VM, load `boot.dll` into x64dbg in preparation for debugging. This 64-bit DLL exports a function named `DllRegisterServer`, so you will need to load `C:\windows\system32\regsvr32.exe` into x64dbg, and update the command line via **File > Change Command Line**. The command line to use is `"C:\Windows\System32\regsvr32.exe" C:\Users\REM\Desktop\boot.dll`.

Within x64dbg in the Dynamic VM, browse to **Options > Preferences** from the menu bar. Then, go to the **Events** tab and check the **Dll Entry** option. Also jump to the **Exceptions** tab, single-click on the only Exception Filter, and click the radio button **Do not break**. This will ensure exceptions are processed as Windows would normally process them without halting the debugger. Next, click **Save**.

Then, restart the target program by browsing to **Debug > Restart**.

Finally, take a VMware snapshot of the Dynamic VM and name it `Lab2.2`, or something similar.

Note

We disabled ALSR for `boot.dll` so the virtual addresses in the solutions will match those in your environment.

Lab Questions

If you performed behavioral analysis with this sample, you could confirm it is ransomware and it encrypts files as expected. However, we will skip this step to save time.

1. ? Within the Static VM, perform some brief static file analysis. Load `boot.dll` into PeStudio and view the strings output. Based on a review of the embedded strings, what potential crypto algorithm(s) might this program use? ^

Within PeStudio, briefly review the imports. Observe that there are no CryptoAPI functions in the IAT, but it is possible these APIs are resolved dynamically. In fact, you might observe the APIs `LoadLibraryExW` and `GetProcAddress` which are commonly used to load libraries and resolve APIs at runtime. Also observe file interaction APIs including `CreateFileW`, `WriteFile` and `SetFilePointerEx`.

2. ? Within the Static VM, switch to Ghidra. Since we ran the FindCrypt extension during the auto-analysis, review the Symbol Tree > Labels for any `CRYPT_` prefixes. What constants did FindCrypt identify? ^

3. ? A logical next step is to identify the function that references these constants in search of the primary encryption function. Unfortunately, Ghidra does not identify any references to these constants. We need another approach. What is one approach we can use within Ghidra to search for functions that perform Salsa or ChaCha encryption? ^

4. ? Using the manual approach discussed in the previous question, at what address is the function that likely performs encryption? Also, what algorithm do you suspect the function implements and why? ^

Important

You've reached **Checkpoint #1** in this lab.

At this point, it makes sense to debug the program to confirm our suspicions about the function we just discussed. Within the Dynamic VM, switch to x64dbg where the debugger should be paused at the entrypoint for `regsvr32.exe`. Run the target program until `boot.dll` is loaded (you should see the DLL mentioned in the title bar). Finally, enter the command `bp 180004e30` to set a breakpoint on the function we suspect implements ChaCha.

5.  Continue executing the program. You should eventually arrive at 180004e30, where we set the breakpoint. Review the arguments passed to this function. Which argument supports our theory that the function implements ChaCha and not Salsa? (e.g., first argument, second argument) What about this argument confirms the algorithm? 

6.  Since we are paused at a function that implements a cipher, some content is about to be encrypted or decrypted (remember, this is a symmetric algorithm so the function could be used to both process plaintext or ciphertext). Within the debugger, review the remaining arguments passed to 180004e30 and allow the function to return. Does this first execution of 180004e30 encrypt or decrypt data? What can you conclude about the content encrypted or decrypted? 

 Note

Consider that an encryption/decryption function has to operate against some plaintext or encrypted data and the resulting content must be placed somewhere in memory.

Since the function at 180004e30 appears to implement the ChaCha cipher, let's update its name within Ghidra. Switch briefly to your Static VM and update the function name to `chacha_cipher`.

Return to the Dynamic VM. If you continue running the program, it will pause at 180004e30 repeatedly. In many cases, the function is used to process the same content discussed in the previous question as the ransomware note is placed throughout the filesystem. We want to explore other potential uses of this cipher. Is the function at 180004e30 only used to decrypt the ransom note, or is it also used for common ransomware tasks like encrypting files?

7.  To investigate how file encryption is performed, let's identify any code that takes file content as input, since this would be necessary step to encrypt a file. What Windows API is often responsible for placing file content into a buffer? 

Remove the breakpoint on the 180004e30 for now (i.e. go to the **Breakpoint** tab, choose the breakpoint and hit the **Delete** key on your keyboard). Set a breakpoint on the API mentioned in the answer to the previous question and continue running the program. The debugger should arrive at the breakpoint.

8.  At what address within `boot.dll` does this CALL to ReadFile occur? 

14. ? Does this program use one key for all files or a separate key per file? How can you confirm the correct answer using the debugger? ^

15. ? Shift briefly to your static VM and jump to the body of the `chacha_cipher` function. We confirmed this code is likely ChaCha, but we have not determined the number of rounds implemented. Is this ChaCha20, ChaCha12, or something else? ^

 Note

Review the slide titled `In FUN_004034f0`, each loop iteration has 8 quarter-rounds...

 Important

You've reached **Checkpoint #3** in this lab.

We identified the function and algorithm used to encrypt file data, and we also understand how the per-file key and nonce are generated. In order for an attacker to successfully hold files for ransom, they must be able to decrypt the data upon payment. This means the per-file encryption keys must be stored and protected such that the attacker can gain access but the target organization cannot independently decrypt files. Let's explore how this ransomware protects the per-file encryption key.

Recall that ransomware often uses asymmetric algorithms (e.g., RSA, Elliptic-curve cryptography) to protect symmetric keys. When we performed static file analysis of `boot.dll`, recall that we observed the embedded string `RSA1`. As we discussed in the module, hardcoded RSA keys are often imported during execution using `CryptImportKey`. Within x64dbg, configure a breakpoint for this API and a few additional CryptoAPI functions we discussed with the command: `bp CryptAcquireContextA; bp CryptAcquireContextW; bp CryptImportKey; bp CryptEncrypt`.

We could set breakpoints on additional CryptoAPI functions, but we will begin with this smaller set of core APIs. If you already have any of these APIs configured as breakpoints, running the command will *not* remove those breakpoints. Once the breakpoints are configured, restart the program via **Debug > Restart**. and execute until you arrive at the entry point for `boot.dll`.

16. ? Run the program again and the debugger should pause at one of the configured breakpoints. Within what function is the debugger paused, and why does it make sense that we arrived within this function first? ^

Continue running the program and you will encounter another call to WriteFile. We do not have time to explore this reference in class, but students are encouraged to explore this function call outside of class.

26. ? Continue running the program. What API does the debugger arrive at next, and what does it accomplish? ^

27. ? Continue running the program and the debugger will pause at 18000E3E7, where ReadFile is called. We already analyzed this CALL. Dump the address of the buffer (second argument) to a dump window and document two arguments: ^

Address of buffer where content will be stored (second argument): _____

Number of bytes read (third argument): _____

Continue running the program and the debugger will pause at 18000E405, where the ChaCha8 function is called. As previously discussed, this will encrypt the file contents. Step over this function (i.e., **Debug > Step over**) and observe that the original file content in the dump window is overwritten with the encrypted content.

28. ? Continue running the program. What API does the debugger arrive at next, and what does it accomplish? ^

 Note

For a calculator to convert signed hex values to negative decimal values, see [this site](#).

29. ? Continue running the program. What API does the debugger arrive at next, and what does it accomplish? ^

30. ? Summarize what you learned about how this ransomware performs file encryption and key protection. Also, what information is required to decrypt files? ^

Important

You've reached **Checkpoint #1** in this lab.

At this point, it makes sense to debug the program to confirm our suspicions about the function we just discussed. Switch to x64dbg within the Dynamic VM, where the debugger should be paused at the entrypoint for `regsvr32.exe`. Run the target program until `boot.dll` is loaded (you should see the DLL mentioned in the title bar). Finally, enter the command `bp 180004e30` to set a breakpoint on the function we suspect implements ChaCha.

5.  Continue executing the program. You should eventually arrive at 180004e30, where we set the breakpoint. Review the arguments passed to this function. Which argument supports our theory that the function implements ChaCha and not Salsa? (e.g., first argument, second argument) What about this argument confirms the algorithm?

Answer: Dump the first argument to a dump window to observe the initial state passed to the function. The characters that comprise `expand 32-byte k` are not fragmented. This matches the initial state of ChaCha.

6.  Since we are paused at a function that implements a cipher, some content is about to be encrypted or decrypted (remember, this is a symmetric algorithm so the function could be used to both process plaintext or ciphertext). Within the debugger, review the remaining arguments passed to 180004e30 and allow the function to return. Does this first execution of 180004e30 encrypt or decrypt data? What can you conclude about the content encrypted or decrypted?

Note

Consider that an encryption/decryption function has to operate against some plaintext or encrypted data and the resulting content must be placed somewhere in memory.

Answer: This first run of the function at 180004e30 decrypts ransom note text.

Explanation: Dump the second argument to a dump window and observe unreadable data. This may be encrypted content or simply binary data. If we dump the third argument to a dump window, we observe a location that is zeroed out. There is a good chance some content will be placed here

If we allow the function to return via **Debug > Execute till return**, we observe new content at the location that was zeroed out. The content appears to be a ransom note. We can conclude that the first call to the ChaCha function decrypted the ransom note that will be written to disk.

Since the function at 180004e30 appears to implement the ChaCha cipher, let's update its name within Ghidra. Switch briefly to your Static VM and update the function name to `chacha_cipher`.

Return to the Dynamic VM. If you continue running the program, it will pause at 180004e30 repeatedly. In many cases, the function is used to process the same content discussed in the previous question as the ransomware note is placed throughout the filesystem. We want to explore other potential uses of this cipher. Is the function at 180004e30 only used to decrypt the ransom note, or is it also used for common ransomware tasks like encrypting files?

7.  To investigate how file encryption is performed, let's identify any code that takes file content as input, since this would be necessary step to encrypt a file. What Windows API is often responsible for placing file content into a buffer?

Answer: Readfile

Remove the breakpoint on the 180004e30 for now (i.e. go to the **Breakpoint** tab, choose the breakpoint and hit the **Delete** key on your keyboard). Set a breakpoint on the API mentioned in the answer to the previous question and continue running the program. The debugger should arrive at the breakpoint.

8. ✓ **At what address within `boot.dll` does this CALL to ReadFile occur?**

Answer: At address 18000e3e7, the instruction `CALL RAX` calls ReadFile.

Explanation: You can identify the location of the CALL within `boot.dll` using two approaches. First, you can allow ReadFile to return to the user code. Alternatively, you can view the **Call Stack** tab and right-click on the first entry and choose **Follow To**. Both these approaches will take you the instruction immediately after the CALL, and you can scroll up to identify the address where ReadFile is executed.

9. ✓ **We're still reviewing the CALL to ReadFile at 18000e3e7. After a file's content is read into a buffer, what happens next? You can use a combination of both static code analysis and debugging to determine the answer to this question.**

Answer: The file is read in and then the data is encrypted via the CALL instruction at 18000e405.

Explanation: Recall that ReadFile's second argument specifies the address of the buffer for content that is read. Static code analysis with Ghidra shows that this address is passed as the second argument to `chacha_cipher`. If you debug the CALL at 18000e405, you will observe that the first argument passed to `chacha_cipher` specifies the initial state, the second argument specifies the data to encrypt or decrypt, and the third argument specifies the address of the resulting encrypted or decrypted data. For file encryption, the third argument will point to the encrypted file content (it actually overwrites the plaintext content).

 Important

You've reached **Checkpoint #2** in this lab.

10. ✓ **If 180004e30 implements ChaCha, the program must generate some random bytes for the key and nonce before data is encrypted or decrypted. What Crypto API is often used to produce random bytes?**

Answer: CryptGenRandom

Before we move on, let's delete some breakpoints we no longer need and set some new breakpoints within the debugger:

- Delete the breakpoint on the ReadFile API.
- Set a breakpoint on the CALL to ReadFile at 18000e3e7 (i.e., `bp 18000e3e7`). We will focus on this specific CALL to better understand file encryption.
- Set a breakpoint on 18000e405, where `chacha_cipher` is called (i.e., `bp 18000e405`). There are multiple references to `chacha_cipher` in this program, but we will focus on this one in class. Students are encouraged to investigate other references to `chacha_cipher` outside of class.
- Set a breakpoint on the CryptGenRandom API (i.e., `bp CryptGenRandom`), which we just discussed.

Setting the breakpoints mentioned above will help us understand how any randomly generated bytes relate to file encryption. If you have other breakpoints configured, you can disable or delete them.

11. ✓ **Continue executing the program until you arrive at CryptGenRandom. Within x64dbg, review the context of the function call. What does it accomplish? Allow the function to execute until return and dump any generated data to a dump window.**

Answer: This function call generates 0x20 (decimal 32) bytes of random data.

Explanation: The second argument (i.e., `RDX`) specifies the number of random bytes to generate. The third argument (i.e., `R8`) passed to `CryptGenRandom` is the address of a buffer that receives the random bytes. Dump this address to a dump window and allow the function to return so you can view the returned bytes.

12. ✓ Continue executing the program. You will encounter `CryptGenRandom` again. How does this function call differ from the previous call to the same function? Again, allow the function to execute until return and dump any generated data to a dump window. ▾

Answer: This `CryptGenRandom` returns 8 bytes of random data, while the previous function call returned 20 bytes. Dump the address stored in `R8` to a different dump window. Allow the function to return so you can view the returned bytes.

13. ✓ Continue executing the program. When you arrive at the CALL to `ReadFile`, continue execution. The debugger should now pause at 18000E405, where the `ChaCha` function is executed. Review the initial state passed to this function (i.e., see the first argument). How can you confirm the significance of the 32 bytes and 8 bytes of random data generated earlier? ▾

Answer: Look at the initial state and compare it to module slide that included the format of `ChaCha`'s initial state. You will observe that the 32 bytes of random data is the key, and the 8 bytes of random data is the nonce.

14. ✓ Does this program use one key for all files or a separate key per file? How can you confirm the correct answer using the debugger? ▾

Answer: This ransomware generates a new 32-byte key and 8-byte nonce for each file it encrypts.

Explanation: If you continue debugging the program, you will encounter an API pattern that generally consists of two calls to `CryptGenRandom` followed by a call to `ReadFile` and then `chacha_cipher` at 18000E405. This pattern occurs again and again as individual files are processed. Each time a new file's content is read in, the CALL to `chacha_cipher` shows an initial state with a new key and nonce.

15. ✓ Shift briefly to your static VM and jump to the body of the `chacha_cipher` function. We confirmed this code is likely `ChaCha`, but we have not determined the number of rounds implemented. Is this `ChaCha20`, `ChaCha12`, or something else? ▾

 Note

Review the slide titled `In FUN_004034f0, each loop iteration has 8 quarter-rounds...`

Answer: `ChaCha8`

Explanation:

Within `chacha_cipher`, first identify the loop that performs rotate operations. Visually searching for ROL instructions shows the loop occurs between addresses 180004fe0 and 180005162. The first ROL operation occurs at 180004fff, and it is `ROL ESI, 0x10`. Recall that each `ChaCha` quarter-round includes rotate operations that shift bits 16 (0x10), 12 (0xc), 8 (0x8), and 7 (0x7) bit positions. If we highlight `0x10` in the instruction `ROL ESI, 0x10` at 180004fff, we see eight occurrences of this value within ROL operation in the loop. Each ROL operation for a particular shift value represents one quarter-round, so there are 8 quarter-rounds per loop, or 2 complete rounds.

Next, we need to assess how many times the loop iterates. Scroll to the end of the loop and observe the conditional jump that determines if the loop continues iterating. The `JNZ` instruction at 180005162 evaluates if `local_c0` is not zero. If the variable is zero, the loop is exited. If we highlight `local_c0` with a single click and scroll up, we observe it is assigned the value 4 with a `MOV` operation at 180004fcf. This means the loop iterates 4 times, with each iteration consisting of 2 complete rounds. $4 \times 2 = 8$, so we can refer to this `ChaCha` implementation as `ChaCha8`.

 Important

You've reached **Checkpoint #3** in this lab.

Observe that the first value is 0x06, which indicates the data is a public key. The second byte is the version number, which requires no further explanation. The next two bytes are reserved, and they are generally zeroes. The next four bytes specify the algorithm ID (i.e., ALG_ID member of the BLOBHEADER structure) and we can see the hex value a400 (little endian). According to MS documentation (<https://for710.com/algid>), the text representation of this value is `CALG_RSA_KEYX`.

Since what follows is an RSA public key blob, we can view the MS documentation on the RSAPUBKEY structure (<https://for710.com/rsapubkey>). Based on that lookup, we are reminded that the first DWORD `RSA1` confirms this is an RSA public key, and the second DWORD specifies the bit length of the key. In this case, it is 0x1000 (decimal 4096) bits.

20.  **Continue executing the program. You may pause again at `CryptImportKey`, but observe that all calls to this function import the same RSA key. We're seeing multiple references to this API due to the multi-threaded nature of this ransomware. Continue running the program. When you arrive at `CryptGenRandom`, dump the 32 random bytes it produces as you did earlier in this lab. Run the program again and dump the 8 random bytes produces in the next call to `CryptGenRandom`. Run the program again until you arrive within `CryptEncrypt`. How many bytes does this call encrypt, and how would you describe the data that this function call encrypts?**

Answer: `CryptEncrypt` encrypts 40 bytes of data that is comprised of the randomly generated 32-byte key and 8-byte nonce.

Explanation: The fifth argument points to the data to encrypt, and the sixth argument points to the size of data to encrypt. To view the data to encrypt, right-click on the fifth argument in the center window on the right within x64dbg and dump that address to a dump window. Comparing this data to the data generated via `CryptGenRandom` shows this content is a copy of the 40 bytes of random data. This API encrypts the per-file symmetric key and nonce.

To view the size of the data to encrypt, additional steps are necessary because x64dbg only lists the first five arguments by default. The simplest approach to view the sixth argument is to click the up arrow on the spinner button above the list of arguments so that the numerical field increments from 5 to 6. Alternatively, you can right-click on the fifth argument you dumped earlier and choose **Follow...in Stack**. This will take you to the fifth argument on the stack, and the sixth argument is immediately below that value.

Next, right-click on the sixth argument and dump it to a dump window. This reveals the size 0x28 or decimal 40 bytes. This makes sense because the program first generated 32 bytes of random data followed by 8 bytes of random data, totaling 40 bytes.

21.  **We are still reviewing the call to `CryptEncrypt` discussed in the previous question. Based on our analysis of CryptoAPI functions thus far, what key does `CryptEncrypt` likely use for encryption?**

Answer: The RSA public key imported by `CryptImportKey`. You can confirm this by comparing the imported key handles returned by `CryptImportKey` (see this API's sixth argument) to the handle provided as the first argument to `CryptEncrypt`. However, this task is tedious due to the multi-threaded nature of this ransomware.

Important

You've reached **Checkpoint #4** in this lab.

We now understand how the per-file encryption key is protected, but we still need to determine where the key is stored. One common approach in ransomware is to add the encrypted key to each encrypted file. Before we proceed with our investigation, let's revisit our breakpoints. We no longer need to evaluate calls to `CryptAcquireContextA`, `CryptAcquireContextW`, or `CryptImportKey`, because these APIs are only called at the beginning of execution, and we understand the purpose of these functions. We also understand that `CryptGenRandom` generates the key and nonce for each file. Therefore, remove breakpoints for `CryptAcquireContextA`, `CryptAcquireContextW`, `CryptImportKey`, and `CryptGenRandom`.

At this point, you should only have breakpoints set on 18000E3E7, 18000E405, and `CryptEncrypt`. To ensure consistency with the flow of this lab, continue executing the program until you arrive at the beginning of `CryptEncrypt` again. As we discussed, this API encrypts a per-file symmetric key and nonce. You can confirm this by checking the sixth argument, which confirms that `CryptEncrypt` operates against 40

26. ✓ Continue running the program. What API does the debugger arrive at next, and what does it accomplish? ▾

Answer: The debugger pauses at SetFilePointerEx once again. This time, it moves the file pointer to the beginning of the file.

Explanation: The file handle referenced in the first argument matches the file handle in the previous SetFilePointerEx call. The second argument (liDistanceToMove) is zero, so it does not move the file pointer. The fourth argument (dwMoveMethod) is zero, which means the file pointer is moved to the beginning of the file.

27. ✓ Continue running the program and the debugger will pause at 18000E3E7, where ReadFile is called. We already analyzed this CALL. ▾
Dump the address of the buffer (second argument) to a dump window and document two arguments:

Address of buffer where content will be stored (second argument): *This value will vary.*

Number of bytes read (third argument): *This value will vary depending upon the file.*

Continue running the program and the debugger will pause at 18000E405, where the ChaCha8 function is called. As previously discussed, this will encrypt the file contents. Step over this function (i.e., **Debug > Step over**) and observe that the original file content in the dump window is overwritten with the encrypted content.

28. ✓ Continue running the program. What API does the debugger arrive at next, and what does it accomplish? ▾

 Note

For a calculator to convert signed hex values to negative decimal values, see [this site](https://for710.com/hextodec).

Answer: The debugger pauses at SetFilePointerEx again. This time, it moves the file pointer back to the beginning of the file using a signed value in the second argument.

Explanation: The file handle referenced in the first argument matches the file handle in the previous CreateFileW, ReadFile, and SetFilePointerEx calls. The fourth argument (dwMoveMethod) is 1, which means this argument does not impact the file pointer. However, the second argument (liDistanceToMove) appears signed. If you convert this value to a negative decimal value, it is the negative version of the number of bytes ReadFile read. For example, if ReadFile read 100 bytes (which moves the file pointer forward 100 bytes), this call to SetFilePointerEx would move the file pointer backwards 100 bytes (i.e., -100). For a calculator to convert signed hex values to negative decimal values, see <https://for710.com/hextodec>.

29. ✓ Continue running the program. What API does the debugger arrive at next, and what does it accomplish? ▾

Answer: The debugger pauses at WriteFile, which writes the encrypted content to the beginning of file. In other words, it overwrites the original file with the encrypted data.

Explanation: The second argument passed to WriteFile points to the data to write. This is the address of the encrypted data.

30. ✓ Summarize what you learned about how this ransomware performs file encryption and key protection. Also, what information is required to decrypt files? ▾

Answer: This ransomware uses the ChaCha8 algorithm and a per-file 32-byte key to encrypt file contents. Per-file encryption keys and nonce data are encrypted with RSA 4096 using an embedded RSA public key, and this encrypted content is appended to the target file. To decrypt files, the attacker must provide the RSA private key that corresponds with the embedded RSA public key.

Lab Objectives, Revisited

This lab reinforced the following analysis activities:

- Identify specific ciphers used in a ransomware sample.
- Determine the purpose of an identified cipher (e.g., file encryption or key protection).
- Recognize Windows APIs that facilitate encryption.
- Summarize key aspects of how ransomware performs file encryption.

Notes

- There are actually *two* functions that include KSA code. List both.
- When viewing search hits returned via **Search > Program Text**, clicking on a row jumps to the relevant assembly. To highlight the corresponding decompiler output, use the mouse to drag across one or more lines of assembly.
- Recall that there are two components to KSA code: a loop that initializes the S-box and another loop that mixes up S-box values using the key.

3.  Using Ghidra, view references to each of the two functions identified in the previous question. Based on this review, which function is more likely used by this program for encryption and/or decryption? 

4.  Our focus is on the KSA code within the function discussed in the previous question. The loop that initializes the S-box appears to include more than the identity permutation. What is the additional code within the loop likely responsible for, and how does it relate to the RC4 variations we discussed in the module? 

5.  In the Decompiler view, view the second do-while loop after the initialization loop. What does this loop likely accomplish? 

6.  Which function includes the RC4 pseudo-random generation algorithm (PRGA) code? 

Note

This question asks to find the PRGA code associated with the KSA code discussed in the previous question.

7.  Next, switch to the Dynamic VM and use a debugger to execute the PRGA loop. Does it encrypt or decrypt content, and how would you describe the plaintext data associated with this operation? 

 Note

The XOR instruction at 40b2e3 places content at a location in memory. You may want to keep an eye on this location in memory when you debug the PRGA loop.

 Important

You've reached the **Checkpoint** in this lab.

We just decrypted some content, but we still need to confirm the algorithm used. If we configure CyberChef to RC4 decrypt the same content and our results match the decrypted content in the debugger, we can be certain this malware implements RC4. To perform this test, we need to extract the RC4 key and encrypted data.

8. **What is the RC4 key length?**

 Notes

- To locate the RC4 key length, recall that the KSA code includes a mod operation where the key length is the modulus.
- Use your Static VM to perform static code analysis within Ghidra. Then, switch to your Dynamic VM to set a breakpoint and debug the code.

9. **What argument passed to FUN_0040b1c0 contains the key length (e.g., first argument, second argument)?**

 Note

Static code analysis is sufficient to answer this question.

10. **What is the RC4 key? Provide your answer in hexadecimal format.**

Notes

- Again, perform some static code analysis within Ghidra. Then, switch to your Dynamic VM to set a breakpoint and debug the code.
- To locate the RC4 key, remember that this malware implements a variation of RC4 where the mod operation in the KSA occurs in the same loop as the S-box initialization. Recall that the purpose of the mod operation is to identify a single byte within the key that is within the range of the key size. Therefore, after the mod operation is executed, the remainder is likely used to access an offset within the key.

11.  What argument passed to FUN_0040b1c0 likely points to the RC4 key (e.g., first argument, second argument)? 

Note

Static code analysis is sufficient to answer this question.

We have a key and keylength. Now, we need encrypted data so we can test decryption.

12.  What is the size of the encrypted data? 

Notes

- Recall that the PRGA code will execute one byte of keystream data with one byte of encrypted data. This means the PRGA loop will iterate once for each byte of encrypted data.
- Again, perform some static code analysis and then debug the program as needed.
- The JC (Jump if carry) instruction is identical to JB (Jump if below).

13.  What argument passed to FUN_0040b1c0 likely contains the size of the encrypted data (e.g., first argument, second argument)? 

Note

Static code analysis is sufficient to answer this question.

14.  What argument passed to FUN_0040b1c0 likely points to the encrypted data (e.g., first argument, second argument)? 

2. ✓ Behavioral analysis would reveal this malware likely belongs to a ransomware family. However, we will skip that phase of analysis for this lab. Based on static file analysis, we observed crypto constants associated with the ChaCha and Salsa ciphers. Running the FindCrypt extension during auto-analysis confirms the presence of these constants. Could other crypto algorithms be implemented in this code as well? We know RC4 is common in malware and it does not use any constants, so we'll have to perform a manual search to explore its presence. Within Ghidra, identify the function(s) that includes the RC4 key scheduling algorithm (KSA) code.

Notes

- There are actually two functions that include KSA code. List both.
- When viewing search hits returned via **Search > Program Text**, clicking on a row jumps to the relevant assembly. To highlight the corresponding decompiler output, use the mouse to drag across one or more lines of assembly.
- Recall that there are two components to KSA code: a loop that initializes the S-box and another loop that mixes up S-box values using the key.

Answer: Functions FUN_0040b1c0 and FUN_0040b060 both contain RC4 KSA code.

Explanation:

Recall that the S-box initialization involves creating a 256 byte array, where each element is assigned the value of its index number. This is referred to as the identity permutation. This assignment is typically performed within a loop that iterates over each element. One approach to identify this loop is to search for the value 0x100 (decimal 256), which often appears in the loop condition that determines if the loop continues executing. Then, we can look for the identity permutation.

To search for this value, go to the menu bar and browse to **Search > Search Program Text**. In the **Search for** field, type `0x100`. Among the **Fields** options on the bottom left of the window, only check **Instruction Operands**. Then, click **Search All**.

While there are numerous results, we want to focus on instructions that assess a value. Sort the results by the **Preview** column and observe that there are only five CMP instructions. Jump to each CMP instruction and highlight nearby code to find the corresponding decompiler output. Among the five CMP instructions, only two are part of loops where the control variable (i.e., the variable that is incremented with each iteration) is assigned to an array element index that matches the control variable. This likely represents the identity permutation. The two functions that contain KSA code are FUN_0040b1c0 and FUN_0040b060. The other functions that contain CMP instructions include code that is far too complex in the loop body to be considered KSA functions.

3. ✓ Using Ghidra, view references to each of the two functions identified in the previous question. Based on this review, which function is more likely used by this program for encryption and/or decryption?

Answer: FUN_0040b1c0

Explanation: While both FUN_0040b1c0 and FUN_0040b060 contain KSA code, FUN_0040b1c0 has one reference while FUN_0040b060 has none (i.e., zero references). This means FUN_0040b060 may be an unused function, so it is not a good target of our analysis. While not necessary in this case, you could debug the program for further confirmation. If you set a breakpoint on 40b060 and executed the program, you would find that the debugger never pauses at that address.

4. ✓ Our focus is on the KSA code within the function discussed in the previous question. The loop that initializes the S-box appears to include more than the identity permutation. What is the additional code within the loop likely responsible for, and how does it relate to the RC4 variations we discussed in the module?

Answer: In the Decompiler view, observe the loop that initializes the S-box to the identity permutation. It includes a mod (i.e., %) operation. This is likely a variation of the KSA where the initialization loop includes key expansion.

5. ✓ In the Decompiler view, view the second do-while loop after the initialization loop. What does this loop likely accomplish?

Answer: As we discussed in the module, there are two components of the KSA code: 1) initialize the S-box and 2) mix up S-box values. We already identified the S-box initialization code. This second loop is likely where the S-box values are mixed up.

6. ✓ Which function includes the RC4 pseudo-random generation algorithm (PRGA) code?

 Notes

This question asks to find the PRGA code associated with the KSA code discussed in the previous question.

Answer: FUN_0040b1c0

Explanation: PRGA code is usually executed soon after the KSA code. PRGA code may be in the same function as the KSA code or in a separate function. First, let's look at the function that contains KSA code (i.e. FUN_0040b1c0) to see if the KSA resides there.

One initial indicator we can look for is an XOR operation. If you review FUN_0040b1c0 in the decompiler view and scroll down, there is a third do-while loop. View the body of this loop and observe an XOR (i.e., ^) operation.

We should also look for the presence of at least two mod 256 operations in the loop. The decompiler view shows two & 0xff operations, which are equivalent to a mod 256. This third loop in FUN_0040b1c0 is likely the PRGA, and it is contained within the same function as the KSA code.

7. ✓ Next, switch to the Dynamic VM and use a debugger to execute the PRGA loop. Does it encrypt or decrypt content, and what is the plaintext data associated with this operation?

 Notes

The XOR instruction at 40b2e3 places content at a location in memory. You may want to keep an eye on this location in memory when you debug the PRGA loop.

Answer: The PRGA loop decrypts configuration information for this ransomware.

Explanation: The XOR operation in RC4 PRGA code XORs a byte of data with a byte of keystream data. Reviewing the operands of the XOR instruction should give us insight into the data being XORed, and the result of the XOR operation should provide visibility into the encrypted or decrypted content.

The XOR operation within the PRGA code occurs at 40b2e3. Within x32dbg, set a breakpoint at this address (bp 40b2e3). Then, run the program until it pauses at the breakpoint. The destination operand is the only operand that points to a memory location, so let's observe what content is placed there. Right-click on the XOR instruction and choose **Follow in Dump > Address: ESI+ECX*1**. Step over the XOR instruction and you should observe a change to a single byte. To view the resulting content after the PRGA code completes all iterations of the loop, remove the breakpoint at 40b2e3 and set a new one at 40b2ef, the address immediately after the loop. Then, continue running the program. The dump window now has additional content that looks like configuration data.

 Important

You've reached the **Checkpoint** in this lab.

We just decrypted some content, but we still need to confirm the algorithm used. If we configure CyberChef to RC4 decrypt the same content and our results match the decrypted content in the debugger, we can be certain this malware implements RC4. To perform this test, we need to extract the RC4 key and encrypted data.

8. ✓ What is the RC4 key length?

Notes

- To locate the RC4 key length, recall that the KSA code includes a mod operation where the key length is the modulus.
- Use your Static VM to perform static code analysis within Ghidra. Then, switch to your Dynamic VM to set a breakpoint and debug the code.

Answer: 5 bytes

Explanation:

View the decompiler output for FUN_0040b1c0 within Ghidra. To locate the RC4 key, identify the assembly instruction associated with the mod operation in the KSA. Within the first do-while loop, drag and highlight the code with the mod operation. Ghidra shows that this correlates with the DIV instruction at 40b23b. As a reminder, the DIV instruction divides EDX:EAX by the specified operand. The quotient is placed into EAX and the remainder is placed into EDX. The DIV operand should reference the key size.

Next, use your Dynamic VM to debug the program. First, restart `winbio.exe` within x32dbg. Then, set a breakpoint at 40b23b and run the program. When you arrive at the breakpoint, click on the div instruction and view the window immediately below the disassembly. There, you will observe the value stored at esp+20. The size of the RC4 key is 5 bytes.

9. What argument passed to FUN_0040b1c0 contains the key length (e.g., first argument, second argument)?

Note

Static code analysis is sufficient to answer this question.

Answer: The second argument.

Explanation: In the previous question, we identified that the single operand for the DIV instruction at 40b23b is the key length. Ghidra shows this operand as `param_2`, so the second argument passed to FUN_0040b1c0 is likely the key length.

10. What is the RC4 key? Provide your answer in hexadecimal format.

Notes

- Again, perform some static code analysis within Ghidra. Then, switch to your Dynamic VM to set a breakpoint and debug the code.
- To locate the RC4 key, remember that this malware implements a variation of RC4 where the mod operation in the KSA occurs in the same loop as the S-box initialization. Recall that the purpose of the mod operation is to identify a single byte within the key that is within the range of the key size. Therefore, after the mod operation is executed, the remainder is likely used to access an offset within the key.

Answer: The key is `2C 47 76 71 63` (ascii `,Gvqc`).

Explanation:

In the previous question, we interpreted the DIV instruction at 40b23b. We understand this instruction divides a value by the key length, and the remainder is placed into EDX. If we look for references to EDX after the DIV instruction, we see one at 40b240 with the instruction `MOV AL, byte ptr [EDX + EBX*0x1]`. If EDX is an offset within the key, perhaps EBX contains the starting address of the key.

To test this theory, switch to the debugger. Remove all other breakpoints and restart the target program. Then, set a breakpoint at 40b240 and continue execution. When the debugger pauses at 40b240, dump the value in EBX to a dump window. Observe a null-terminated 5 bytes consisting of the hexadecimal values `2C 47 76 71 63` (ascii `,Gvqc`). This is likely the key.

11. What argument passed to FUN_0040b1c0 likely points to the RC4 key (e.g., first argument, second argument)?

 Note

Static code analysis is sufficient to answer this question.

Answer: The first argument.

Explanation: In the previous question, we identified that when the MOV instruction at 40b240 is executed, EBX contains the address of the RC4 key. Highlight EBX in this instruction and observe earlier references. At 40b22b, `param_1` is placed into EBX. This means the first argument passed to FUN_0040b1c0 likely points to the RC4 key.

We have a key and keylength. Now, we need encrypted data so we can test decryption.

12. **What is the size of the encrypted data?**

 Notes

- Recall that the PRGA code will XOR one byte of keystream data with one byte of encrypted data. This means the PRGA loop will iterate once for each byte of encrypted data.
- Again, perform some static code analysis and then debug the program as needed.
- The JC (Jump if carry) instruction is identical to JB (Jump if below).

Answer: 0x1496, or decimal 5270 bytes.

Explanation: The third loop within FUN_0040b1c0 is the PRGA code. We need to identify how many times it will iterate, and this should equal the size of the encrypted data. In the decompiler view, drag and highlight the `while` condition within the third loop. This correlates with the CMP and JC instructions beginning at address 40b2eb. These instructions evaluate if ESI is below EBP. EBP should contain the number of bytes of encrypted data.

To determine the size of the encrypted data, we could set a breakpoint on the CMP instruction at 40b2eb and view the contents of EBP. This approach reveals the hex value 1496, or decimal 5270 bytes.

13. **What argument passed to FUN_0040b1c0 likely contains the size of the encrypted data (e.g., first argument, second argument)?**

 Note

Static code analysis is sufficient to answer this question.

Answer: The fourth argument.

Explanation: Continuing our analysis from the previous question, highlight EBP with a single-click and scroll up. Observe that the fourth argument passed to FUN_0040b1c0 is placed into EBP at 40b29b. This means the fourth argument passed to FUN_0040b1c0 is likely the size of the encrypted content to decrypt.

14. **What argument passed to FUN_0040b1c0 likely points to the encrypted data (e.g., first argument, second argument)?**

 Note

Static code analysis is sufficient to answer this question.

Answer: The third argument.

Lab Objectives, Revisited

This lab reinforced the following analysis activities:

- Gain familiarity with RC4, a common symmetric cipher for encrypting data in malware.
- Identify an RC4 implementation in malware.
- Debug malware to decrypt data.
- Extract an RC4 key using a combination of static and dynamic code analysis.
- Confirm the presence of RC4 by testing decryption outside of the target program.

Lab 3.1: Automating Config Extraction with Python

Background

In this lab, we will write a Python config extractor.

Lab Objectives

- Develop comfort with the Python programming language for malware analysis.
- Practice using the pefile Python module to parse a PE file.
- Gain familiarity with other Python modules, including argparse, pycryptodome, and json.
- Apply output from prior static code analysis and debugging to automate a malware analysis task.
- Create a malware configuration extractor.

Lab Preparation

To prepare your Dynamic VM:

1. Revert your Dynamic VM to a baseline state. Then, extract `Malware\Section3\winbio_and_more.zip` and place the directory `winbio_and_more` on the desktop. Within `winbio_and_more`, observe two files. Most of this lab will focus on `winbio.exe`, the same file we analyzed in Lab 2.3.
2. If VS Code is not already open, launch the program using the desktop shortcut.
3. Within VS Code, create a new file by going to the menu bar and choosing **File > New File**.
4. Still within VS Code, browse to **File > Save**. Create the directory (if it doesn't already exist) `C:\Users\REM\python_scripts` and browse to it. Then, specify the file name `extract_config_lab31.py` and click **Save**.
5. In the lower part of the VS code window, ensure the terminal is visible. If it is not present, go to the menu bar and choose **View > Terminal**. Then, split the terminal so you have two terminal windows open. To accomplish this, mouse over the buttons on the top-right of the terminal until you arrive at the one with the description `Split Terminal`. Both terminals should be at the location `C:\Users\REM\python_scripts` (use the `cd` command to arrive there if needed).

To prepare your Static VM: Launch Ghidra and open your analysis of `winbio.exe` from Section 2. If you do not have access to your prior analysis for some reason, simply create a new `Section 3` project and load `winbio.exe` from `Malware\Section2\winbio.zip`. In this case, process the file and initiate auto-analysis. Be sure to check `WindowsPE x86 Propagate External Parameters` because the EXE is 32-bit.

Lab Questions

Let's warm up with some basic usage of pefile within an interactive Python shell. In VS code, type `python` in the terminal on the bottom-right.

1.  First, import the pefile module using the appropriate command.

2. ? Next, type the command to load `winbio.exe` using `pefile`. The command should assign the loaded executable to a variable so we can use various methods to query the target file. ^

 Note

Remember to use double slashes in the path to the target executable.

3. ? What Python code will print the name of the first section within `winbio.exe` ? What is the section name? ^

 Note

- Remember, we want to use Python and `pefile` to determine the answer to this question.
- Review `help(pefile.PE)` to help guide how you access sections.
- Consider using the built-in `type()` function if you are unsure how an attribute should be accessed, or if you want to clarify what type of data a function returns. For example, typing `type(target.is_exe())` outputs `<class 'bool'>`.

4. ? Type a command to calculate the SHA-256 hash of the *first* section within `winbio.exe` . What is the SHA-256 hash? ^

 Note

Consider using the `dir()` command for information about methods available to an object. For example, if your loaded PE file is in a variable `target`, then `dir(target)` shows methods available for the loaded binary.

Now that we are warmed up, let's proceed to write our config extractor for `winbio.exe` . As a reminder, this is the executable you first analyzed in Lab 2.3.

5. ? Based on your analysis in Lab 2.3, where was the encrypted configuration data stored within `winbio.exe` ? ^

6. ? Within the resource that contains the encrypted key data and encrypted data, what is the structure of the content? For example, in the walkthrough we discussed in this module, the anomalous section included a 32-byte RC4 key, then a 4-byte CRC32 checksum, then 4 bytes that specified the size of the encrypted data, and finally the encrypted data. ^

Lab Solutions

Let's warm up with some basic usage of pefile within an interactive Python shell. In VS code, type `python` in the terminal on the bottom-right.

1. ✓ First, import the pefile module using the appropriate command. ▼

Answer: `import pefile`

2. ✓ Next, type the command to load `winbio.exe` using pefile. The command should assign the loaded executable to a variable so we can use various methods to query the target file. ▼

Note

Remember to use double slashes in the path to the target executable.

Answer: `target = pefile.PE("C:\\Users\\REM\\Desktop\\winbio_and_more\\winbio.exe")`

3. ✓ What Python code will print the name of the first section within `winbio.exe` ? What is the section name? ▼

Note

- Remember, we want to use Python and pefile to determine the answer to this question.
- Review `help(pefile.PE)` to help guide how you access sections.
- Consider using the built-in `type()` function if you are unsure how an attribute should be accessed, or if you want to clarify what type of data a function returns. For example, typing `type(target.is_exe())` outputs `<class 'bool'>`.

Answer: The command is `target.sections[0]`, and the first section name is `.text`.

Explanation: As stated in the output from `help(pefile.PE)`, `sections` will be available as a list in the `sections` attribute. Assuming the loaded executable is available in the `target` variable, we can access the first section with the command `target.sections[0]`. In the output, observe this first section is named `.text`.

4. ✓ Type a command to calculate the SHA-256 hash of the first section within `winbio.exe`. What is the SHA-256 hash? ▼

Note

Consider using the `dir()` command for information about methods available to an object. For example, if your loaded PE file is in a variable `target`, then `dir(target)` shows methods available for the loaded binary.

Answer: The command is `target.sections[0].get_hash_sha256()`. The resulting hash is `9994bca758fe1ed4ca868e2e1a474b145778278ab5b4a57ac65ca9719a31f886`.

Now that we are warmed up, let's proceed to write our config extractor for `winbio.exe`. As a reminder, this is the executable you first analyzed in Lab 2.3.

5. ✓ Based on your analysis in Lab 2.3, where was the encrypted configuration data stored within `winbio.exe` ? ▼

Answer: In the final question in Lab 2.3, we confirmed the encrypted configuration is embedded in `winbio.exe` within the resource section.

6. ✓ Within the resource that contains the encrypted key data and encrypted data, what is the structure of the content? For example, in the walk-through we discussed in this module, the anomalous section included a 32-byte RC4 key, then a 4-byte CRC32 checksum, then 4 bytes that specified the size of the encrypted data, and finally the encrypted data.

Note

- Within Ghidra, find the appropriate resource within the `.rsrc` section. By default Ghidra will have the resource bytes contracted. Click the `+` button to expand its contents.
- Take advantage of the **Window > Bytes** view if desired to view the raw bytes.
- Review the final questions in Lab 2.3 as needed to remind yourself of the key, key size, and encrypted data values.

Answer: The structure of the resource is as follows, in this order:

4 bytes: RC4 key size (little endian)

5 bytes: RC4 key

Remaining data in resource: encrypted data

7. ✓ Let's take a brief look at `dirmon.exe`, located in the same directory you unzipped for this Lab. This executable belongs to the same malware family as `winbio.exe`, it performs similar malicious behavior, and it implements the RC4 encryption algorithm. Load this program into CFF Explorer (see desktop shortcut). Once loaded, click on *Resource Editor* on the left and expand the folder structure to view the contents of the single resource. Notice that the first 4 bytes differ from the values in `winbio.exe`. What impact does this have on your extraction script?

Answer: The extraction script must use the first four bytes of the resource data to determine the size of the RC4 key. Using the appropriate key size, the script will then extract the RC4 key.

Important

You've reached the **Checkpoint** in this lab.

8. ✓ Write a Python script that accomplishes the tasks noted below.

Requirements

- Accept a target file (required) and output file (optional) on the command line.
- Parse a PE file.
- Iterate over resource directories until reaching the resource data.
- Identify the RC4 key size, the RC4 key, and the encrypted data within the resource data.
- RC4 decrypt the encrypted configuration data.
- Write the decrypted configuration to a file.

```
import pefile
import argparse
from Crypto.Cipher import ARC4
import json
```

```

parser = argparse.ArgumentParser(description="Config extractor for Lab 3.1.")
parser.add_argument("-f", "--file", help="Target file for config extraction.", required=True)
parser.add_argument("-o", "--output", help="Config output file.", required=False)
args = parser.parse_args()

target = pefile.PE(args.file)

#Investigate resources
for resource in target.DIRECTORY_ENTRY_RESOURCE.entries:
    for entry in resource.directory.entries:
        for entry2 in entry.directory.entries:
            rsrc_data = target.get_data(entry2.data.struct.OffsetToData, entry2.data.struct.Size)

            #Get key length
            key_length = rsrc_data[:4]
            key_length_int = int.from_bytes(key_length, "little")
            print("Key length is: " + str(key_length_int))

            #Get key
            key = rsrc_data[4:4+key_length_int]
            print("Key: " + key.hex())

            #Get encrypted data
            data = rsrc_data[4+key_length_int:]

            #Decrypt data
            cipher = ARC4.new(key)
            decrypted_data = cipher.decrypt(data)

            #Output decrypted content; converting to string for consistency but not needed in this case.
            decrypted_str = decrypted_data.decode()
            config_data = json.loads(decrypted_str)

            if args.output:
                with open(args.output, "w") as f:
                    f.write(json.dumps(config_data, indent=4))

```

Lab Objectives, Revisited

This lab reinforced the following analysis activities:

- Develop comfort with the Python programming language for malware analysis.
- Practice using the pefile Python module to parse a PE file.
- Gain familiarity with other Python modules, including argparse, pycryptodome, and json.
- Apply output from prior static code analysis and debugging to automate a malware analysis task.
- Create a malware configuration extractor.

Note

As a reminder, to debug a program that exports `DllRegisterServer`:

- Launch `x64dbg`.
- Browse to **File > Open** and choose `C:\Windows\System32\regsvr32.exe`
- Browse to **File > Change Command Line** and use the command line `"C:\Windows\System32\regsvr32.exe" C:\Users\REM\Desktop\frida_files\fcd.dll`.
- Browse to **Options > Preferences** and view the **Events** tab. There, check the option for **DLL Entry**.
- Restart the target program by browsing to **Debug > Restart**.
- Run the program, and you should arrive at the entry point for `fcd.dll`.

8. ? Dump the content you observed in the previous question and perform some basic static analysis. How would you characterize the content? Do the functions listed under *imports* seem familiar?

We just manually extracted a payload. Let's automate this process with Frida's python bindings.

9. ? When writing a Frida python script to deobfuscate content, we need to articulate the debugging workflow that revealed the code. How would you describe the steps we took to identify the second-stage executable in memory?

Important

You've reached the **Checkpoint** in this lab.

10. ? Write a Frida Python script that accomplishes the tasks noted below.

Requirements

- Accept arguments to run a single EXE or a DLL (with rundll32.exe or regsvr32.exe). For example, if your script is called `lab32.py` you will run `fcd.dll` with the command `python lab32.py C:\Windows\System32\regsvr32.exe fcd.dll`.
- Intercept calls to VirtualAlloc.
- Each time VirtualAlloc returns, add the starting address for the newly allocated region to an array.
- Each time VirtualAlloc is called, check if a previously noted region begins with the ascii characters `MZ`.
- If the `MZ` bytes are detected at the beginning of a memory region, write the content at that location to a file on disk.
- Test the script against all four files within the unzipped 'frida_files' folder.
- Use the file `frida_template.py` located in the folder for this lab as a starting point for your script.
- Consult the Frida JavaScript API as necessary (<https://for710.com/fridaapi>).

Lab Solutions

1.  Within the Dynamic VM, perform some brief file analysis of `fcd.dll` with PeStudio. Based on PeStudio output, what approach would be used to launch this executable? 

Answers: The target file is a 64-bit DLL, and it exports `DllRegisterServer`. This means it should be executed using `C:\windows\system32\regsvr32.exe`.

2.  Continue viewing PeStudio output and observe the list of imported functions. With so few imports, it is likely that additional libraries will be resolved at runtime. We know that calling `GetProcAddress` is one approach to resolving functions during execution. Open a command prompt and navigate to `C:\Users\REM\Desktop`. What `frida-trace` command line can you use to execute `fcd.dll` and observe any calls to `GetProcAddress`? 

Notes

- For this question, *only* focus on the `GetProcAddress` API exported by `KERNEL32.DLL`.
- Remember that `fcd.dll` is a 64-bit DLL. This should help determine which program you use to run the DLL.

Answer: `frida-trace C:\windows\system32\regsvr32.exe frida_files\fcd.dll -i KERNEL32.DLL!GetProcAddress`

3.  Run the command specified in the previous question. What key information is missing from the command output? 

Answer: The output does not specify the name of the API that is resolved.

After viewing `frida-trace` output, type `Ctrl+C` to exit the process. Also, launch Process Hacker from the desktop and terminate `regsvr32.exe`.

4.  Executing `frida-trace` created a folder named `__handlers__` on the Desktop (assuming that is where you ran `frida-trace` from). Modify the handler for `GetProcAddress` to print out the additional information we require (see the answer to the previous question if

you need more context). The output should print the name of the API resolved. For example, if the program resolved `CreateProcessA`, the relevant line of output would read `GetProcAddress(): CreateProcessA`.

Answer:: Modify the handler located at `__handlers__\KERNEL32.DLL\GetProcAddress.js`. The second argument passed to `GetProcAddress` is a pointer to the API name. The updated `OnEnter` function is:

```
onEnter(log, args, state) {  
  log('GetProcAddress(): ' + args[1].readUtf8String());  
},
```

The new output includes the following:

```
GetProcAddress(): wsprintfA  
GetProcAddress(): GetUserNameA  
GetProcAddress(): LookupAccountNameW  
GetProcAddress(): SHGetFolderPathA  
GetProcAddress(): CreateProcessA  
GetProcAddress(): CreateDirectoryA  
GetProcAddress(): GetProcAddress  
GetProcAddress(): lstrcpYA  
GetProcAddress(): GetTempPathA  
GetProcAddress(): Sleep  
GetProcAddress(): CreateThread  
GetProcAddress(): ExitProcess  
GetProcAddress(): WriteFile  
GetProcAddress(): CreateFileA  
GetProcAddress(): CloseHandle  
GetProcAddress(): HeapFree  
GetProcAddress(): HeapReAlloc  
GetProcAddress(): HeapAlloc  
GetProcAddress(): GetProcessHeap  
GetProcAddress(): GetComputerNameExW  
GetProcAddress(): GetTickCount64  
GetProcAddress(): GetLastError  
GetProcAddress(): LoadLibraryA  
GetProcAddress(): SwitchToThread  
GetProcAddress(): lstrcatA  
GetProcAddress(): GetComputerNameExA  
GetProcAddress(): WinHttpRequestDataAvailable  
GetProcAddress(): WinHttpRequestConnect  
GetProcAddress(): WinHttpRequestSetStatusCallback  
GetProcAddress(): WinHttpRequestSendRequest  
GetProcAddress(): WinHttpRequestCloseHandle  
GetProcAddress(): WinHttpRequestSetOption  
GetProcAddress(): WinHttpRequestOpenRequest  
GetProcAddress(): WinHttpRequestReadData  
GetProcAddress(): WinHttpRequestQueryHeaders  
GetProcAddress(): WinHttpRequestOpen  
GetProcAddress(): WinHttpRequestReceiveResponse  
GetProcAddress(): WinHttpRequestQueryOption  
GetProcAddress(): memset  
GetProcAddress(): memcpy
```

After modifying the `GetProcAddress` handler, we observe many APIs resolved at runtime. As we discussed in Section 1 of this course, one reason malware may resolve APIs at runtime is to prepare for the next stage of execution (i.e., an underlying payload). Recall that when malware unpacks additional code or deobfuscates data, it often needs to allocate memory for this content. One approach to allocating memory involves using the Virtual API (e.g., `VirtualAlloc`, `VirtualProtect`). We did not see any Virtual API calls resolved using `GetProcAddress`, but the malware may use other approaches to resolve APIs. Let's explore this possibility.

Answer:

- We set a breakpoint on VirtualAlloc and ran the program.
- Each time we encountered this API, we allowed it to return and dumped the returned address of the newly allocated region to the dump window.
- When we arrived at the third call to VirtualAlloc, the first region allocated now had an MZ header and appeared to contain a Windows executable.

Important

You've reached the **Checkpoint** in this lab.

10. ✓ Write a Frida Python script that accomplishes the tasks noted below.

Notes

Requirements for the script:

- Accept arguments to run a single EXE or a DLL (with rundll32.exe or regsvr32.exe). For example, if your script is called `lab32.py` you will run `fcd.dll` with the command `python lab32.py C:\Windows\System32\regsvr32.exe fcd.dll`.
- Intercept calls to VirtualAlloc.
- Each time VirtualAlloc returns, add the starting address for the newly allocated region to an array.
- Each time VirtualAlloc is called, check if a previously noted region begins with the ascii characters `MZ`.
- If the `MZ` bytes are detected at the beginning of a memory region, write the content at that location to a file on disk.
- Test the script against all four files within the unzipped 'frida_files' folder.
- Use the file `frida_template.py` located in the folder for this lab as a starting point for your script.
- Consult the Frida JavaScript API as necessary (<https://for710.com/fridaapi>).

Answer: The script below is one approach to writing a payload extraction script using Frida's python bindings.

```
import frida
import sys
import argparse

def main():
    parser = argparse.ArgumentParser(description='Dump payload.')
    parser.add_argument('targets', nargs='+')
    args = parser.parse_args()

    pid = frida.spawn(args.targets)
    session = frida.attach(pid)

    script = session.create_script("""

        //Load module
        try {
            Module.load('KERNEL32.DLL');
        } catch {
            console.log(err);
        }

        //Get function address.
        try {
```

```

        var vaExportAddress = Module.getExportByName("KERNEL32.dll", "VirtualAlloc");
    } catch(err) {
        console.log(err);
    }

    //Array of memory regions to monitor
    var memRegions = [];

    //Configure interceptor(s)
    Interceptor.attach(vaExportAddress,
    {
        onEnter: function (args) {
            this.vaSize = args[1].toInt32();
            var vaProtect = args[3];
            console.log("\n\nVirtualAlloc called => Size: " + this.vaSize + " | Protection: " +
vaProtect);

            for(var i = 0; i < memRegions.length; i++)
            {
                console.log("\n\nChecking memory at " + memRegions[i].memBase.toString());

                try {
                    var firstBytes = memRegions[i].memBase.readAnsiString(2);
                } catch(err) {
                    console.log(err);
                }

                if (firstBytes == "MZ")
                {
                    console.log("\t\tFound an MZ!\n\n");
                    console.log(hexdump(memRegions[i].memBase));

                    //Write file to disk
                    var binContent =
memRegions[i].memBase.readByteArray(memRegions[i].memSize);
                    var filename = memRegions[i].memBase + "_mz.bin";
                    var file = new File(filename, "wb");
                    file.write(binContent);
                    file.flush();
                    file.close();

                    console.log("\n\nDumped file: " + filename);
                }
            }
        },
        onLeave: function (retval) {
            console.log("\n\nVirtualAlloc returned => Address: " + retval);
            memRegions.push({memBase:ptr(retval), memSize:this.vaSize});
        }
    });
    """)

    script.load()
    frida.resume(pid)
    sys.stdin.read()
    session.detach()

    if __name__ == '__main__':
        main()

```

Lab Objectives, Revisited

This lab reinforced the following analysis activities:

- Gain familiarity with frida-trace, a command line tool in the Frida framework.

- Write a script using Frida's Python bindings to automatically dump a malicious payload.

3. ? Type a command to determine the entry point of the function, and assign that value to a variable named `ep_addr`. To verify your answer, type `ep_addr` and compare the output to the address of the first instruction in the Listing view. ^
-  **Note**
- Consider using the Program API for this answer. We discussed a method in the slides that can help.
4. ? Type a command to get the instruction at the entry point for this function and assign it to a variable named `instr`. Again, to verify your answer, type `instr` and compare your output to the instruction in the Listing view within Ghidra. ^
-  **Note**
- Return to using the Flat API when considering a command.
5. ? Type a command to print the mnemonic at this address. ^
6. ? Type a command to print the single operand for the instruction. ^
-  **Note**
- You will need the Program API for this command. Review the slides for a method that might help.
7. ? Now, type a command to get the *next* instruction. Return the instruction to a variable named `instr_next`. ^
-  **Note**
- Return to using the Flat API for this command.
8. ? Type a command to print out the *second* operand of the instruction in `instr_next`. ^
9. ? Write code to get references to `ep_addr` (i.e., the entry point of the function) and print out the address where each reference is located. ^

Important

You've reached the **Checkpoint** in this lab.

Now, let's write our string decoding script. To create a new file for our code, follow these steps:

- Open the Script Manager by clicking on the green "Play" button under the menu bar, or browse to **Window > Script Manager**.
- Mouse over the buttons on the top-right and click on the one with the description **Create New Script**.
- Choose the option to create a Python script and click **OK**.
- Use the default directory `C:\Users\REM\ghidra_scripts` and enter the script file name `lab33_string_decode.py`.
- In your new script, see the metadata template located at the top of the file. For the `@category` field, specify `_FOR710`. The underscore ensures the new category will be placed at the top of the Scripts listing located on the left side of the Script Manager window (you will need to click the button with green arrows in the Script Manager to refresh the list of script categories).

10.  **Write a Ghidra Python script that deobfuscates strings processed by FUN_004093f0. For each decoded string, add an EOL comment at the address where FUN_004093f0 is referenced. The comment should include the decoded string. Below is one approach to consider. You may implement an alternative approach if you desire.**

Approach

- Find references to the decoding function.
- For each reference, look up to 5 instructions before the reference for a MOV instruction where the first operand is EDX. Consider using the `range()` function in a for loop to accomplish this (<https://for710.com/range>).
- If an instruction that satisfies the above requirement is found, get the second operand and interpret it as an address.
- For each reference to the data address identified in the previous requirement, check if the instruction has a MOV mnemonic and if the first operand is ECX.
- If the above condition is true, go to the instruction that precedes the MOV and get the single operand. This operand should point to the encoded data we desire.
- Get the encoded data and then get its value. When calling `getValue()`, consider doing so within a try/except statement. This provides an opportunity to print an error message if the script encounters undefined data. If you have time, you can manually define strings at the necessary locations to address all errors. Alternatively, you can explore using the Flat API's `createAsciiString()` method (which we did not explicitly discuss) to define a string.
- Then, base64 decode this data and perform the operations specified in the decoding loop. Note that when `3` is added in the loop, you must `& 0xFF` the result since it is placed in a single byte register.
- Print the decoded value to the console.
- Finally, add an EOL comment at the address where the decoding function is referenced. Include the decoded string.

Lab Solutions

Let's warm up by taking advantage of the Ghidra Flat and Program API within an interactive Python shell. To access the Python interpreter within Ghidra, go to the menu bar and choose **Window > Python**. As you consider and type commands, remember to take advantage of `tab` to view

suggested commands, and `help()` and `dir()`. In addition, consult the slides in this module and the API documentation. Shortcuts to the Flat and Program API documentation are accessible via two toolbar bookmarks in Firefox. You can launch Firefox via the Desktop shortcut.

1. ✓ In preparation for this lab, the slides refer to the decoding loop beginning at 409490 within `program.exe`. Type the command to convert this hexadecimal value to an address and assign it to a variable named `loop_addr` variable. After typing your command, confirm success with the command `type(loop_addr)`. The output should be `<type 'ghidra.program.model.address.GenericAddress'>`

Answer: `loop_addr = toAddr("409490")` or `loop_addr = toAddr(0x409490)`

2. ✓ Next, type a command to determine the function where this address resides. Assign the returned value to a variable named `decoding_fn`.

 Note

Review the slides or the Flat API documentation when considering your command.

Answer: `decoding_fn = getFunctionContaining(loop_addr)`

3. ✓ Type a command to determine the entry point of the function, and assign that value to a variable named `ep_addr`. To verify your answer, type `ep_addr` and compare the output to the address of the first instruction in the Listing view.

 Note

Consider using the Program API for this answer. We discussed a method in the slides that can help.

Answer:

```
>>> ep_addr = decoding_fn.getEntryPoint()
>>> ep_addr
004093f0
```

4. ✓ Type a command to get the instruction at the entry point for this function and assign it to a variable named `instr`. Again, to verify your answer, type `instr` and compare your output to the instruction in the Listing view within Ghidra.

 Note

Return to using the Flat API when considering a command.

Answer:

```
>>> instr = getInstructionAt(ep_addr)
>>> instr
PUSH EBX
```

Approach

- Find references to the decoding function.
- For each reference, look up to 5 instructions before the reference for a MOV instruction where the first operand is EDX. Consider using the `range()` function in a for loop to accomplish this (<https://for710.com/range>).
- If an instruction that satisfies the above requirement is found, get the second operand and interpret it as an address.
- For each reference to the data address identified in the previous requirement, check if the instruction has a MOV mnemonic and if the first operand is ECX.
- If the above condition is true, go to the instruction that precedes the MOV and get the single operand. This operand should point to the encoded data we desire.
- Get the encoded data and then get its value. When calling `getValue()`, consider doing so within a try/except statement. This provides an opportunity to print an error message if the script encounters undefined data. If you have time, you can manually define strings at the necessary locations to address all errors. Alternatively, you can explore using the Flat API's `createAsciiString()` method (which we did not explicitly discuss) to define a string.
- Then, base64 decode this data and perform the operations specified in the decoding loop. Note that when `3` is added in the loop, you must `& 0xFF` the result since it is placed in a single byte register.
- Print the decoded value to the console.
- Finally, add an EOL comment at the address where the decoding function is referenced. Include the decoded string.

Answer: The script below is one approach to decoding the obfuscated strings in this sample.

```
#This script deobfuscates strings in sample with SHA-256 hash
0bd3eb756c9297f9be08f79fa7a93e925c08df18656b54b9e2333d4a2445a58f.
#@author Anuj Soni
#@category _FOR710
#@keybinding
#@menupath
#@toolbar

import base64

#Insert function address below
decoding_fn_addr = toAddr(0x4093f0)
fn_refs = getReferencesTo(decoding_fn_addr)

for ref in fn_refs:
    from_addr = ref.getFromAddress()
    instr = getInstructionBefore(from_addr)
    for i in range(5):
        if instr.getMnemonicString() == "MOV" and str(instr.getOpObjects(0)[0]) == "EDX":
            dat_addr = toAddr(str(instr.getOpObjects(1)[0]))
            break
        instr = getInstructionBefore(instr.getAddress())
    else:
        print("NOTE: No data address found near decoding function reference at " + str(from_addr))
        continue

    dat_refs = getReferencesTo(dat_addr)
    for ref2 in dat_refs:
        from_addr2 = ref2.getFromAddress()
        instr = getInstructionAt(from_addr2)
        data_str = ""
        if instr.getMnemonicString() == "MOV" and str(instr.getOpObjects(0)[0]) == "ECX":
            instr2 = getInstructionBefore(instr.getAddress())
            addr_encoded = instr2.getOpObjects(0)[0]
            data = getDataAt(toAddr(str(addr_encoded)))
            try:
                data_str = data.getValue()
```

```

        except:
            print("NOTE: No defined data at this location, see address in Ghidra: " +
                  addr_encoded.toString())
            continue
        else:
            continue

        b64_decoded_str = base64.b64decode(data_str)
        decoded_str = ""

        for i in b64_decoded_str:
            decoded_val = ord(i) ^ 8
            decoded_val = (decoded_val + 3) & 0xFF
            decoded_val = decoded_val ^ 84
            decoded_str = decoded_str + chr(decoded_val)

        print(decoded_str)
        setEOLComment(from_addr, decoded_str)

```

Lab Objectives, Revisited

This lab reinforced the following analysis activities:

- Gain familiarity with Ghidra's available APIs.
- Use Ghidra's built-in Python interpreter to explore available APIs.
- Write a Python script within Ghidra that performs string deobfuscation and adds helpful comments in the Listing view.

After generating a brief overview of our target files, we will explore relationships between some of them. First, we will compare `program.exe` and `dns.exe`.

2. ? Use the provided `pecompare.py` script to compare `program.exe` and `dns.exe`. This script will compare embedded strings and functions listed in the IAT. Using the terminal, run the following command from `C:\Users\REM\Desktop\Malware\Section4`: `python pecompare.py collection\program.exe collection\dns.exe > diff_program_dns.txt`. Double-click the output file to view its contents in Notepad++. What do you learn from this comparison of `program.exe` and `dns.exe`? Can you identify any strings that may be unique to this malware?

3. ? Use BinDiff to assess similarities and differences between `program.exe` and `dns.exe`. What conclusions can you draw from BinDiff output?

 Note

To compare these programs using BinDiff, do the following:

- Launch BinDiff using the desktop shortcut.
- From the menu bar, browse to **File > New Workspace...**
- Name the workspace `workspace_collection`.
- For the **Location** field, click on ..., browse to `C:\Users\REM\Desktop\Section4\bindiff` and click **Open**.
- In the left window pane, right-click and choose **New Diff...**
- For the **Primary file**, browse to `C:\Users\REM\Desktop\Malware\Section4\bindiff` and choose `program.exe.BinExport`.
- For the **Secondary file**, browse to `C:\Users\REM\Desktop\Malware\Section4\bindiff` and choose `dns.exe.BinExport`.
- Then, click **Diff**.
- To view BinDiff output, view the left window pane and double-click `program.exe vs. dns.exe`.

Remember that we are most concerned about unlabeled functions because these are more likely to represent code developed by the malware author.

4. ? For a more granular comparison of `program.exe` and `dns.exe`, we will try Ghidra's Program Diff feature. After performing the diff, use the down arrow at the top of the listing view to click past the first 10 or so differences. You will eventually arrive at 409495, where

Ghidra displays differences in code within a loop. You should recognize the displayed instructions from our analysis in Lab 3.3. How would you characterize the difference in code at this location?

 Note

To compare these programs with Ghidra's Program Diff feature, do the following:

- Go to the Ghidra project window and open the analysis of `program.exe`.
- From the menu bar, choose **Tools > Program Differences....**
- When prompted, double-click `dns.exe` to provide it as the second program for comparison.
- When choosing how to assess differences between the programs, only check **Bytes**, **Code Units**, and **Functions**.
- Click **OK**.

Let's briefly explore how `program.exe` compares to one of the larger executables that has a different imphash. Specifically, we will compare `program.exe` with `save.exe`.

5.  Use `pecompare.py` again to compare `program.exe` and `save.exe`. What do you learn from this comparison of `program.exe` and `save.exe`? For example, despite the significant differences between these programs, are there any overlapping strings that may uniquely identify this malware? Also, what can you learn from the differences in imported functions?

6.  Finally, use BinDiff to compare `program.exe` and `save.exe` (follow the process described in the Notes for question #3). What percentage of functions was BinDiff able to match?

7.  Reviewing the differences and similarities in this case would be tedious and time consuming, so we will focus on one particular difference. View the **Primary Unmatched Functions** and type `4226c0` in the input field at the top to filter by this value. Why might this unmatched function in `program.exe` be of concern?

1. ✓ Run `pestats.py` against the `collection` directory to generate some basic information about each file. Using the terminal, run the following command from `C:\Users\REM\Desktop\Malware\Section4`: `python pestats.py collection`. This will produce a file `pestats_out.csv`. Drag-and-drop the output file to the Timeline Explorer shortcut, which is located on your desktop. Based on reviewing the output, what do you learn about this group of files?

Answer: Both `program.exe` and `dns.exe` are the same size and they have the same import table hash (i.e., `imphash`), indicating they have similar functionality. The `.text` section hashes for these two files do differ, however, indicating some differences in the code. These two executables are also much smaller in size when compared to the others. The compile time indicates these two programs were also compiled later (2021 vs. 2020).

After generating a brief overview of our target files, we will explore relationships between some of them. First, we will compare `program.exe` and `dns.exe`.

2. ✓ Use the provided `pecompare.py` script to compare `program.exe` and `dns.exe`. This script will compare embedded strings and functions listed in the IAT. Using the terminal, run the following command from `C:\Users\REM\Desktop\Malware\Section4`: `python pecompare.py collection\program.exe collection\dns.exe > diff_program_dns.txt`. Double-click the output file to view its contents in Notepad++. What do you learn from this comparison of `program.exe` and `dns.exe`? Can you identify any strings that may be unique to this malware?

Answer: Your review of `diff_program_dns.txt` may include the following observations:

- We know both executables have the same `imphash`, so there is not much to gain from viewing import related information in this output.
- The vast majority of strings within the two programs are identical. This is not a surprise because the executables are identical in size and the `.data` sections have the same hash.
- Some interesting overlapping strings that may be unique to these malware samples include:

```
"ext": "  
"hdd": "  
"lang": "  
"name": "  
"rcid": "  
"size": "  
"type": "
```

- If we scroll down to the sections that list strings unique to each sample, we observe many strings that appear to be base64 encoded (see strings ending in `=`, which represents padding). We'll return to this observation later.

3. ✓ Use BinDiff to assess similarities and differences between `program.exe` and `dns.exe`. What conclusions can you draw from BinDiff output?

Note

To compare these programs using BinDiff, do the following:

- Launch BinDiff using the desktop shortcut.
- From the menu bar, browse to **File > New Workspace...**
- Name the workspace `workspace_collection`.
- For the **Location** field, click on ..., browse to `C:\Users\REM\Desktop\Section4\bindiff` and click **Open**.
- In the left window pane, right-click and choose **New Diff...**
- For the **Primary file**, browse to `C:\Users\REM\Desktop\Malware\Section4\bindiff` and choose `program.exe.BinExport`.
- For the **Secondary file**, browse to `C:\Users\REM\Desktop\Malware\Section4\bindiff` and choose `dns.exe.BinExport`.
- Then, click **Diff**.
- To view BinDiff output, view the left window pane and double-click `program.exe vs. dns.exe`.

Remember that we are most concerned about unlabeled functions because these are more likely to represent code developed by the malware author.

Answer: Bindiff reports that all functions match (i.e., no unmatched functions). When viewing the list of matched functions, we see some that have a similarity of less than 1.00, but these are labelled and unlikely to be developed by the malware author.

4. ✓ For a more granular comparison of `program.exe` and `dns.exe`, we will try Ghidra's Program Diff feature. After performing the diff, use the down arrow at the top of the listing view to click past the first 10 or so differences. You will eventually arrive at 409495, where Ghidra displays differences in code within a loop. You should recognize the displayed instructions from our analysis in Lab 3.3. How would you characterize the difference in code at this location? ✓

Note

To compare these programs with Ghidra's Program Diff feature, do the following:

- Go to the Ghidra project window and open the analysis of `program.exe`.
- From the menu bar, choose **Tools > Program Differences...**
- When prompted, double-click `dns.exe` to provide it as the second program for comparison.
- When choosing how to assess differences between the programs, only check **Bytes, Code Units, and Functions**.
- Click **OK**.

Answer: When reviewing Ghidra's Program Diff output, we eventually arrive at 409495, where we see code that matches the decoding loop we analyzed in Lab 3.3. Ghidra makes it clear that although the decoding loop still involves an XOR, ADD, and XOR operations, the numerical values have changed. This also explains why `pecompare.py` showed so many base64-encoded strings that were unique to each sample—a different decoding algorithm would require different based64 encoded content to produce the same string.

Let's briefly explore how `program.exe` compares to one of the larger executables that has a different imphash. Specifically, we will compare `program.exe` with `save.exe`.

5. ✓ Use `pecompare.py` again to compare `program.exe` and `save.exe`. What do you learn from this comparison of `program.exe` and `save.exe`? For example, despite the significant differences between these programs, are there any overlapping strings that may uniquely identify this malware? Also, what can you learn from the differences in imported functions? ✓

Answer: Your review of `pecompare.py` output may include the following observations:

- `save.exe` has many more strings than `program.exe`, which makes sense because it is a larger file.
- Among the 1514 common strings, most appear generic. However, you might spot `"rcid":` near the beginning. This could be a string unique to these malware samples.
- Among the many strings unique to `save.exe`, we observe some worthy of note including:

```
%s.exe
%s.tmp
.\%s.exe
.\gm.exe
.\medcon.exe
```

- The imphash values for these two programs do not match, and this is supported by the list of imports unique to each executable. For example, `program.exe` imports APIs including `LookupPrivilegeValueW` and `AdjustTokenPrivileges`. If you are not familiar with these APIs, some brief open-source research will indicate malware often uses these APIs to modify access permissions. `save.exe` has its own unique imports, including the use of `wininet.dll` to import various HTTP and Internet-related APIs. These differences are good starting points for additional code analysis.

6. ✓ **Finally, use BinDiff to compare `program.exe` and `save.exe` (follow the process described in the Notes for question #3). What percentage of functions was BinDiff able to match?**

Answer: 88.6%.

7. ✓ **Reviewing the differences and similarities in this case would be tedious and time consuming, so we will focus on one particular difference. View the Primary Unmatched Functions and type `4226c0` in the input filed at the top to filter by this value. Why might this unmatched function in `program.exe` be of concern?**

Answer: The function at `4226c0` calls the `ShellExecuteW` API, which can be used to execute an arbitrary command.

Lab Objectives, Revisited

This lab reinforced the following analysis activities:

- Practice approaches to comparing executables.
- Use BinDiff to compare functions in executables.
- Use Ghidra's Program Diff feature to identify byte-level differences in code.

Lab Solutions

1. ✓ This lab has only one task-write a YARA rule within `collection.yara` that meets the criteria described below. Perform code analysis and use `pecompare.py` as necessary to write the rule. As part of your testing, you will need to execute YARA with the command `c:\Users\REM\Desktop\Malware\Section4>yara64.exe collection.yara collection`

Notes

Your YARA rule should meet the following criteria:

- Only detect `program.exe` and `dns.exe` (i.e., not `open.exe` and `save.exe`)
- Include text strings.
- Include hex strings that identify code in the string deobfuscation loop at 409495 within *both* `program.exe` and `dns.exe` (consider using wildcards).
- Include a condition that checks for an 'MZ' header using this ascii string's byte values.
- Include a condition that checks for an imphash.
- Include a condition that checks the target file's size.
- The rule should hit on a file if it matches the specified imphash OR if it contains all specified strings AND the specified decoding routine.

Answer: Below is one approach to writing a YARA rule that satisfies the specified criteria.

```
import "pe"

rule collection_rule {
  meta:
    description = "This rule is for lab 4.2, and it identifies the smaller sized samples."
    author = "Anuj Soni"
    hash1 = "0BD3EB756C9297F9BE08F79FA7A93E925C08DF18656B54B9E2333D4A2445A58F"
    hash2 = "BC4E8BEFEA8F4E3A37F24C84109CA39BB427B953BCF2FCCEC8D5BC819B83DC20"

  strings:
    $s1 = "\\ext\\:\\\\" nocase ascii wide
    $s2 = "\\hdd\\:\\\\" nocase ascii wide
    $s3 = "\\lang\\:\\\\" nocase ascii wide
    $s4 = "\\name\\:\\\\" nocase ascii wide
    $s5 = "\\rcid\\:\\\\" nocase ascii wide
    $s6 = "\\size\\:\\\\" nocase ascii wide
    $s7 = "\\type\\:\\\\" nocase ascii wide

    $decode_add_xor = { 8a 06 8d 4d bc 34 ?? 04 ?? 34 ?? 0f b6 c0 50 }

  condition:
    uint16be(0) == 0x4D5A and
    filesize < 1048576 and
    (pe.imphash() == "B56503B8C4F46A3A086734C09C6BD0F3" or all of them)
}
```

Lab Objectives, Revisited

This lab reinforced the following analysis activities:

- Practice writing and tweaking a YARA rule.

Lab 4.3: Writing capa Rules

Background

In this lab, we will write a capa rule to identify the ChaCha encryption algorithm in code.

Lab Objectives

- Practice writing and tweaking a capa rule.

Lab Preparation

For this lab, we will only use the Static VM. Perform the following steps:

1. Browse to `Malware\Section4\`.
2. Extract `capa43.zip` using the password `malware`. This should produce a folder named `capa43` that contains a DLL, EXE, and an empty folder named `lab43_rules`.
3. Within `lab43_rules`, create a text file via a **right-click > New > Notepad++ Document**. Rename the file to `encrypt-data-using-chacha.yml` and open it in Notepad++.
4. Launch Ghidra from the desktop.
5. Create a new project named `Section4_capa`
6. Load and initiate auto-analysis for both `boot.dll` and `winfax.exe`.
7. After processing is complete, go to the Listing view for `boot.dll` and jump to 180004feb. There, you will find code we previously analyzed associated with the ChaCha encryption algorithm.
8. Open a command prompt and browse to `C:\Users\REM\Desktop\Malware\Section4\capa43`.

Lab Questions

1.  **Write a capa rule within `lab43_rules\encrypt-data-using-chacha.yml` that meets the criteria described below. Perform code analysis as necessary to write the rule. As part of your testing, you will need to execute capa with the command `c:\Users\REM\Desktop\Malware\Section4\capa43> C:\tools\capa.exe -r lab43_rules\encrypt-data-using-chacha.yml boot.dll`** 

Notes

Your capa rule should meet the following criteria:

- Use the namespace `lab43_rules`.
- Identify the ChaCha algorithm based on key instructions we have seen in implementations of this algorithm.
- Include number and mnemonic features.
- Use a scope such that the results indicate the location within a function where the quarter-rounds occur.

2. ? Run the capa rule against the second executable, `winfax.exe`. At what location does capa identify the basic block that contains the quarter-round operations? ^

 Notes

Consider using the command line flags `-v` or `-vv`.

3. ? Using Ghidra, jump to the location within `winfax.exe` where the identified basic block resides. What do you notice about that code and how can we proceed to confirm the activity at this location within the program? ^

Lab Solutions

1. ✓ Write a capa rule within `lab43_rules\encrypt-data-using-chacha.yml` that meets the criteria described below. Perform code analysis as necessary to write the rule. As part of your testing, you will need to execute capa with the command `C:\Users\REM\Desktop\Malware\Section4\capa43> C:\tools\capa.exe -r lab43_rules\encrypt-data-using-chacha.yml boot.dll` v

 Notes

Your capa rule should meet the following criteria:

- Use the namespace `lab43_rules`.
- Identify the ChaCha algorithm based on key instructions we have seen in implementations of this algorithm.
- Include number and mnemonic features.
- Use a scope such that the results indicate the location within a function where the quarter-rounds occur.

Answer: Below is one approach to writing a capa rule that satisfies the specified criteria.

```
rule:
  meta:
    name: encrypt data using ChaCha
    namespace: lab43_rules
    author: Anuj Soni
    scope: basic block
    examples:
      - 29ed74821564be25cedc3ad0aa091b5e7fb8ad979b8eadbb48ddec9d3013bad:0x180004e30
  features:
    - and:
      - and:
        - mnemonic: rol
        - number: 0x7
```

```
- and:
  - mnemonic: rol
  - number: 0x8
- and:
  - mnemonic: rol
  - number: 0xc
- and:
  - mnemonic: rol
  - number: 0x10
```

2. ✓ Run the capa rule against the second executable, `winfax.exe`. At what location does capa identify the basic block that contains the quarter-round operations? ▼

 Notes

Consider using the command line flags `-v` or `-vv`.

Answer: To view more detail on capabilities that capa identifies, we can use the `-v` or `-vv` command line flags. For example, if we type `c:\Users\REM\Desktop\Malware\Section4\capa43> C:\tools\capa.exe -v -r lab43_rules\encrypt-data-using-chacha.yml winfax.dll`, the output reveals that the identified basic block is at address 4346700 within `winfax.exe`.

3. ✓ Using Ghidra, jump to the location within `winfax.exe` where the identified basic block resides. What do you notice about that code and how can we proceed to confirm the activity at this location within the program? ▼

Answer: Ghidra did not disassemble this content. To view the code, click at the beginning of the code block and type `D` on the keyboard to disassemble.

Lab Objectives, Revisited

This lab reinforced the following analysis activities:

- Practice writing and tweaking a capa rule.