

710.3

Automating Malware Analysis



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.



Automating Malware Analysis

© 2022 Anuj Soni | All Rights Reserved | Version H02_05

Section FOR710.3, also known as Section 3 of the FOR710 course, focuses on discussing approaches to automating malware analysis.

FOR710.3 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 3

- ***Python for Malware Analysis***
 - Lab 3.1: Automating Config Extraction with Python
- Malware Analysis with DBI Frameworks
 - Lab 3.2: Automate Payload Extraction with Frida
- Automating Analysis with Ghidra
 - Lab 3.3: Scripting with Ghidra

This page intentionally left blank.

Python for Malware Analysis

This page intentionally left blank.

Python for Malware Analysis: Module Objectives

- Develop comfort with Python for malware analysis.
- Gain experience using the pefile Python module for parsing PE files.
- Apply our knowledge of the PE file format to programmatically analyze a Windows executable.
- Understand how Python can help automate the results of our prior static code analysis and debugging efforts.
- Create a malware configuration extractor.

This slide describes the objectives of this module.

Consider These Caveats to Calibrate Your Expectations

- This module focuses on introducing Python programming for RE; it does not provide an in-depth introduction to the language in general.
- Consider attending the SANS course titled SEC573: Automating Information Security with Python (sans.org/sec573).
- When writing scripts for malware analysis, your goal is functioning code that produces the expected output—do not worry about code quality.
- If the script you create is of value in future analysis efforts, you will have plenty of opportunity to iteratively improve your code.
- Consider the Pareto principle: time spent optimizing the code will require most of your time if you focus on a 100% solution.

It's worth emphasizing that the purpose of this module is to introduce Python for malware analysis, and not to introduce *all* aspects of the Python programming language. For a thorough discussion of the Python programming language, consider attending the six-section SANS course titled SEC573: Automating Information Security with Python.

Also, scripts created during malware analysis are typically written under time constraints, and scripts created for one malware variant or family may or may not be applicable to future malware. As a result, the goal is simply to write a script that “works” by producing the intended output. Do not focus on code quality and performance, at least initially. If the script you create has value in the future, there will be opportunity to iteratively improve.

Remember the Pareto principle when writing malware analysis scripts (i.e., the 80/20 rule): 80% of the benefit will come from 20% of the work. The remaining 20% may require an unrealistic amount of time, and in most cases is unnecessary. Applying this to writing scripts, the point is that you can produce functional, helpful scripts in a relatively short period of time. However, if you obsess over code quality, performance, and robustness, you will spend most of your time on these details. Write a script that gets the job done and worry about adding polish to your script later.

Python Background

- Python is an interpreted language, which means a .py script is executed by a helper program (i.e., an interpreter).
- To run a Python program on Windows, use `py.exe` or `python.exe`:
 - `python myscript.py`
 - `py myscript.py`
- The Python interpreter:
 - Compiles the source code into platform-independent bytecode.
 - Runs the bytecode in the Python Virtual Machine (PVM), which converts bytecode into machine-executable code.

Python is an interpreted language, which means a helper program called an interpreter is required to run the source code.

Within our Windows VMs, you can run a Python program using the `python.exe` or `py.exe` executables. It is not necessary to include “.exe” on the command line. We will use “python” or “python.exe” for clarity and consistency.

The interpreter first converts the source code to byte code, and this byte code is provided to the Python Virtual Machine (PVM) for execution. The PVM converts bytecode to machine code and runs it.

Python 3 Is Installed in Your Windows VMs

- The Python installer is available at python.org.
- Several additional modules are installed in your VMs, including pefile, pycryptodome, and frida-tools.
- Packages were installed using `pip`, the package manager for Python:
 - `pip3 install <package_name>`
 - Using `pip3` ensures the package will be installed into a Python 3 environment.
- Search for other available packages at <https://pypi.org/>.

We are using Python 3 during this class. If you have prior exposure to Python, you have probably used Python 2 at some point. Python 2 sunset on January 1, 2020. We will not discuss the differences between Python 2 and Python 3 in class but view this resource for a brief summary about the differences: <https://for710.com/python2v3>.

Python 3 is already installed within your Windows VMs. If you need to install Python on another Windows system, the recommended approach is to download an installer from <https://www.python.org/>. During the install, it is important to add Python to your PATH variable so you can execute the interpreter from anywhere within a terminal.

Several additional Python modules were installed within your Windows VMs, including pefile, pycryptodome, and frida-tools. All modules will be discussed later in this section. These modules were installed using `pip`, a package manager for Python. To learn more about `pip`, see <https://for710.com/pip>. Using `pip3` instead of `pip` in the install command ensures the specified program is installed into the Python 3 environment. This is an important detail if you have both Python 2 and Python 3 installed on your system.

To explore other available Python packages, browse to <https://pypi.org/>.

Microsoft's Visual Studio (VS) Code

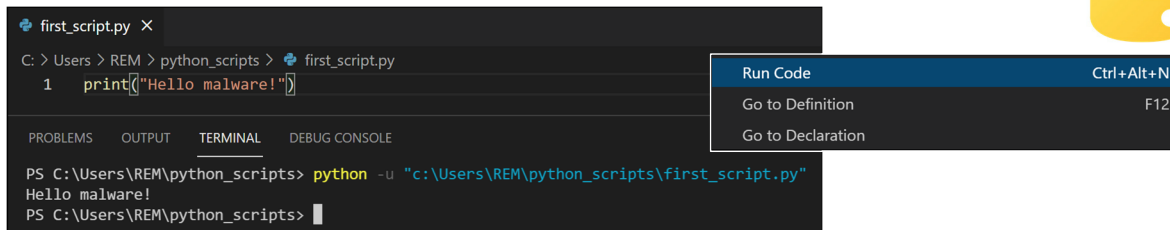
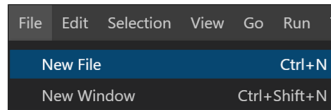
- Visual Studio Code is a free and open-source Integrated Development Environment (IDE) that accommodates many programming languages.
- Several VS extensions are installed, including:
 - Python
 - Code Runner
 - Vim emulation (disabled)
- PyCharm is another popular IDE, specifically for Python.



There are multiple code editors to choose from. We will use Microsoft's Visual Studio Code, which is free and open source. VS Code is available for macOS, Windows, and Linux. Another popular alternative is PyCharm, an IDE exclusively for Python programming.

VS Code has many extensions available through the Visual Studio Code Marketplace. The VS Code install within your VMs has several modules already installed, including Python, Code Runner, and Vim emulation (disabled by default).

Create and Execute Your First Python Script



Save the file (Ctrl+S) after each modification before running the code.

To get started with Visual Studio Code, first launch the program from your Dynamic VM's desktop.

Then, create a new file by browsing to File > New File.

Then, go to File > Save. Within C:\Users\REM, create a folder named "python_scripts" (right-click > New > Folder). Save your file with the file name first_script.py.

Let's complete our first Python script. Type the text `print("Hello malware!")`.

Then, go to File > Save or use the keyboard shortcut Ctrl+S. A common mistake is to run the code without saving, which will run the code last saved to the file.

To run the code, you can click on the Play symbol on the top-right of the window. Alternatively, you can execute the code via the context menu with a right-click > Run Code. Within the context menu, notice the keyboard shortcut Ctrl + Alt + N (on a Mac this translates to Ctrl + Option + N). This shortcut is provided by the third-party Code Runner extension installed in VS Code. We will use shortcuts often in this class to save time and clicks.

Python Basics

- We use variables to store data:
 - Use descriptive variable names (case-sensitive).
 - Add an underscore (“_”) between words.
- Strings are within double or single quotes—we will use double quotes.
- `print()` : built-in function to output content (default stdout).
- Use `#` for a single-line comment and place multi-line comments between triple quotes (“`"""`”).

```
#My first Python script
text1 = "Hello"
text2 = " malware!"
print(text1 + text2)
```

The next few slides introduce some basic concepts in Python programming.

We will use variables to store data temporarily. The code snippet at the bottom-right of this slide shows two examples of variable assignments. When specifying a variable name, choose terms that are concise and descriptive. If the variable name includes two words, separate those words using an underscore.

In Python, strings are enclosed in quotes. You can use single or double quotes. In this class, we will only use double quotes for consistency.

The built-in `print()` function prints content. We will use `print()` to output content to the terminal. When using `print()`, we can use the add (“+”) operator to concatenate strings.

When writing comments, use the pound symbol (“#”) for single-line comments and enclose multiple lines between triple quotes.

Three Basic Data Types Are Strings, Numbers, and Booleans

```
#My first Python script

#String example
output_text = "Hello malware!"
print(output_text)

#Number and integer example
num = 1

#Number and float example
num_float = 1.1

#Boolean example
is_dll = True
```

Three basic types in Python you should be familiar with are strings, numbers, and Booleans. We already have a good understanding of what a string is.

The numbers category can be broken down into integers and floats. Floats are numbers with decimal places included.

Booleans are True or False (they must begin with a capital letter).

A String Is an Object, and Objects Have Methods (i.e., Functions)

- Use a dot (“.”) after an object to view available methods.
- VS Code provides helpful background about a function’s purpose, its parameters, and return value.

```
C: > Users > REM > python_scripts > first_script.py > ...  
1  #My first Python script  
2  text1 = "Hello"  
3  text2 = " malware!"  
4  print(text1 + text2)  
5  
6  text1.  
   capitalize  
   casefold  
   center  
   count  
   encode
```

```
text1.find()  
(__sub: str, __start: SupportsIndex | None = ...,  
 __end: SupportsIndex | None = ...) -> int  
  
S.find(sub[, start[, end]]) -> int  
  
Return the lowest index in S where substring sub is found,  
such that sub is contained within S[start:end]. Optional arguments  
start and end are interpreted as in slice notation.  
  
Return -1 on failure.
```

A string is an object, and Python provides certain functions that are specific to strings. Functions specific to a certain type of object are called “methods.” You can call a method by typing the object followed by a dot (“.”). VS Code will automatically provide a list of available methods to choose from.

For more additional detail on a method, choose one and type “(”. VS Code will pop up a box with details on the method, its parameters, and return value.

Mathematical, Bitwise, Comparison, and Logical Operations

Mathematical Operators		Comparison Operators		Bitwise Operators	
Operator	Description	Operator	Description	Operator	Description
+	Add	==	Equal	&	AND
-	Subtract	!=	Not equal		OR
/	Division	<	Less than	^	XOR
*	Multiplication	>	Greater than	~	NOT
**	Exponent	<=	Less than or equal to	<<	Left shift
%	Modulus	>=	Greater than or equal to	>>	Right shift

Logical Operators	
Operator	Description
and	True if both statements true
or	True if at least one statement true
not	Reverse the result

This slide lists Python operators that you are likely to encounter and use. This is not a comprehensive list of all operators. See <https://for710.com/pyoperators> for more information.

Note that when using the bitwise shift operators, any gaps created by shifting bits are replaced with zero bits (i.e., bits do not wrap around from one side to the other).

Use Mathematical Comparison and Logical Operators with “if” Statements to Evaluate a Condition

```
C:\Users\REM> python_scripts > first_script.py > ...
1  #Operating on numbers
2  num1 = 5
3  num2 = 1
4
5  num = num1 + num2
6  print("The sum is " + str(num))
7
8  if num > 1 and num < 10:
9      print("The result is within range!")
10 elif num >= 10:
11     print("It's too high!")
12 else:
13     print("It's too low.")
14
15 #Comparing text
16 text = "malware"
17 if text == "malware":
18     print("We have a text match!")

PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

PS C:\Users\REM\python_scripts> python -u "c:\Users\REM\python_scripts\first_script.py"
The sum is 6
The result is within range!
We have a text match!
PS C:\Users\REM\python_scripts> 
```

- Parentheses are only used in an if-statement to force an order of operations.
- Append a colon (“:”) at the end of the if-statement.
- Indentation creates blocks of code (i.e., no {}).

The code on this slide shows an example of how operators are used in Python. The first example involves numbers while the second example involves strings.

This slide also uses an if-statement in Python, which allows us to assess a condition. When using an if-statement in Python, there is no need for parenthesis unless you want to force an order of operations. Also, there are no curly braces to delineate code blocks as is common in other programming languages. Instead, indentation is used. If the assessed condition is true, the indented code block below the if-statement is executed.

A List Is a Flexible Data Type That Includes Multiple Values

- Lists are *mutable*, so values can be modified, added, and removed.
- Use bracket notation to access items.
- Use the colon (":") slicing operator to access part of a list.
- Use the **in** operator:
 - With **for** to iterate through a list.
 - With **if** to check if an item is in the list.

```
C:\Users\REM> python_scripts > first_script.py > ...
1 weekdays = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday"]
2
3 #What is the first weekday?
4 print("The first weekday is " + weekdays[0])
5 print("*****")
6
7 #Print the middle three days
8 print("Middle three weekdays:")
9 print(weekdays[1:4])
10 print("*****")
11
12 #Print weekdays
13 print("All weekdays: ")
14 for day in weekdays:
15     print(day)
16 print("*****")
17
18 #Check today
19 today = "Saturday"
20 print("Today is " + today)
21 if today in weekdays:
22     print("It's a weekday!")
23 else:
24     print("It's the weekend!")

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
PS C:\Users\REM> python -u "c:\Users\REM\python_scripts\first_script.py"
The first weekday is Monday
*****
Middle three weekdays:
['Tuesday', 'Wednesday', 'Thursday']
*****
All weekdays:
Monday
Tuesday
Wednesday
Thursday
Friday
*****
Today is Saturday
It's the weekend!
```

Lists are commonly used to store multiple values. For more information on lists and available methods, see <https://for710.com/python-lists>.

We can iterate over a list to assess its multiple values. The code on this slide demonstrates the use of a `for` loop to iterate over a list. In the example, “day” is considered a loop variable.

To check if a list contains a certain value, we can use an if-statement and the “`in`” keyword:

To access a subset of values within a list, we can use the colon (":") slicing operator. The general format for list slicing is:

- `sample_list[start:stop]` # includes first element through stop-1
- `sample_list[start:]` # includes first element through the entire array
- `sample_list[:stop]` # includes first element through stop-

Specify Command Line Inputs and Outputs with argparse

```
C: > Users > REM > python_scripts > first_script.py > ...
1  import argparse
2
3  parser = argparse.ArgumentParser(description="My script.")
4  parser.add_argument("-o", "--output", help="Output file.", required=True)
5  args = parser.parse_args()
6
7  with open(args.output, "w") as f:
8      f.write("Hello malware!")
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
PS C:\Users\REM\python_scripts> python first_script.py
usage: first_script.py [-h] -o OUTPUT
first_script.py: error: the following arguments are required: -o/--output
PS C:\Users\REM\python_scripts> python first_script.py -h
usage: first_script.py [-h] -o OUTPUT

My script.

optional arguments:
  -h, --help            show this help message and exit
  -o OUTPUT, --output OUTPUT
                        Output file.
PS C:\Users\REM\python_scripts> python first_script.py -o outfile.txt
PS C:\Users\REM\python_scripts> cat outfile.txt
Hello malware!
PS C:\Users\REM\python_scripts> |
```

The argparse module allows us to specify command line arguments for a script. The argparse functionality is not built-in, so the module must be imported. The example on this slide demonstrates how to specify command line arguments.

When using the argparse module, we first create an ArgumentParser object:
`parser = argparse.ArgumentParser(description="My script.")`

We populate the ArgumentParser object with information about relevant arguments using the `add_argument()` method.

Calling `parse_args()` provides access to the supplied command-line arguments. For example, the code on this slide adds an argument called "output". The script then includes the statement:
`args = parser.parse_args()`

After `parse_args()` is executed, we can access the "output" command-line argument as an attribute of `args` via `args.output`.

The code on this slide uses the command-line argument to write text to the specified file. The `with` keyword is often used for file access because it ensures the file is properly closed without explicitly including code to do so.

For more information on argparse, see <https://for710.com/argparse>. For more information on reading and writing files in Python, see <https://for710.com/pythonfiles>.

Creating a Configuration Extractor for file.exe

- In Section 2, we analyzed file.exe and determined that the function at 004059fc implements RC4 and decrypts configuration data.

The screenshot shows a debugger window with a 'Recipe' pane on the left. The 'RC4' recipe is selected, showing a 'Passphrase' of '59 4E 73 38 51 44 36 51 ...' in 'HEX' format. The 'Input' pane shows a hex dump of the encrypted data. The 'Output' pane shows the decrypted JSON configuration data:

```
{
  "pk": "/SVNLPYVd04yhjQWfntNHZ0bsHYzZdZRI-FHj kQuTmE=",
  "pid": "331",
  "sub": "331",
  "dbg": false,
  "fast": true,
  "wipe": true,
  "whl": {

```

Below the debugger window, a memory dump is shown with columns for 'Offset', 'Hex', and 'Ascii'. The 'Hex' column shows the decrypted data, and the 'Ascii' column shows the corresponding text:

Offset	Hex	Ascii
00000000	59 4E 73 38 51 44 36 51 78 57 62 77 51 63 74 44	YNs8QD6QxUbwQctD
00000010	34 74 38 6B 34 64 71 4D 4B 4D 62 41 54 44 34 6B	4t8k4dqMKMbATD4k
00000020	0E B0 27 66 CC 66 00 00 5E 8E A2 05 BC 82 3D 65	0'fIf..^c0%l=e
00000030	F9 F5 C5 A8 A9 90 20 10 B7 69 EC 2E 09 01 65 2F	u5A'@.0..ii..0e/
00000040	B9 62 82 E4 2B A9 6F 2B 44 35 CC A5 97 8C 51 8F	'b a+@o+D5I#11Q
00000050	79 B5 A3 19 9C C8 A7 08 C7 02 63 F6 EC 87 56 60	yyu0 1ES0 Ç c0i1V'
00000060	BC CD 28 92 EB 5C D7 4D 01 7F 5A 82 7C 25 63 01	%I('èxM0 1Z1 %c0

- The encrypted data resided in a section with an unusual name.

Now that we understand some Python basics, let's focus on automating a task. Our goal is to extract the encrypted, embedded configuration from file.exe, an executable we discussed in Section 2. As a reminder, this malware sample has SHA-256 hash 6b212864731c131bd095c2537ca14e10338d3ebf997dda59465c5f1ce73d418b. You can find file.exe and other executables referred to during this walk-through within Malware\Section3\file_and_more.zip.

When we performed some initial static file analysis of file.exe, we observed a section with an unusual name that contained unreadable data. This led us to perform some additional debugging, and eventually we arrived at the conclusion that this content was RC4 decrypted to reveal configuration data.

How could we automate this decryption outside of a debugger? Could we create a python script to run against this and similar executables to extract the configuration content?

Document the Script's Requirements before Writing Any Code

Our script must:

- Accept a target file and output file on the command line.
- Parse a PE file.
- Find the section that contains the obfuscated configuration data.
- Extract data from that section.
- Parse the section data and identify the RC4 key, encrypted data, and any related information based on our understanding of how the data is structured.
- RC4-decrypt the encrypted configuration data.
- Write the decrypted configuration to a file.

The target file must be unpacked.

This slide lists the requirements for the Python script we want to write.

Note that we are creating a *static* config extractor intended to be run against a target file on disk. This means the encrypted content is embedded somewhere in the file, and we can extract it for processing. If the encrypted data were only available in memory after an initial executable unpacked itself, we would need to unpack that executable first.

Explore pefile Usage with a Legitimate Program, notepad.exe

```
PS C:\Users\REM\python_scripts> python
Python 3.9.7 (tags/v3.9.7:1016ef3, Aug 30 2021, 20:19:38) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import pefile
>>> help(pefile)
```

```
class PE(builtins.object)
| PE(name=None, data=None, fast_load=None, max_symbol_exports=8192, max_repeated_symbol=120)
|
| A Portable Executable representation.
|
| This class provides access to most of the information in a PE file.
|
| It expects to be supplied the name of the file to load or PE data
| to process and an optional argument 'fast_load' (False by default)
| which controls whether to load all the directories information,
| which can be quite time consuming.
|
| pe = pefile.PE('module.dll')
```

Since we are exploring a new PE file parsing capability, it's a good idea to use a legitimate program to learn more about the Python module. Malicious programs often have an unusual characteristics and these anomalies may confuse our initial attempts to understand pefile's capabilities.

Within the terminal screen in the lower part of VS Code, type `python` to launch the interactive shell. Then, load the pefile module with the command `import pefile`. In Python, the "import" statement finds the specified module and provides access to the module's functionality.

Let's explore this module by viewing its help information. Type `help(pefile)`, as shown on this slide. In addition to an overview of the module, we see a description of classes contained within the module. Scrolling down (hit the spacebar) provides information about each class. For now, we will only focus on the PE class. The description tells us that this class will give us access to the structure of a PE file, which is precisely what we need for our Windows file analysis. The output also explains how to create an instance of the PE class and read in a file.

The dir() Command Provides a Glimpse of Methods

```
>>> dir(pefile.PE)
['_IMAGE_BASE_RELOCATION_ENTRY_format_', '_IMAGE_BASE_RELOCATION_format_', '_IMAGE_BOUND_FORWARDER_REF_format_']

_c', '_q', '_format', '_et_ct_ab_e', '_get_h_s_h', '_l_c', '_n', '_o_c_a_
'_le_', '_lt_', '_module_', '_ne_', '_new_', '_parse_', '_reduce_', '_reduce_ex_', '_repr_',
'_setattr_', '_sizeof_', '_str_', '_subclasshook_', '_unpack_data_', '_weakref_', 'adjust_FileAlignment'
'adjust_SectionAlignment', 'close', 'dump_dict', 'dump_info', 'dword_align', 'full_load', 'generate_checksum',
'get_bytes_from_data', 'get_data', 'get_data_from_dword', 'get_data_from_qword', 'get_data_from_word', 'get_dword
_at_rva', 'get_dword_from_data', 'get_dword_from_offset', 'get_imphash', 'get_import_table', 'get_memory_mapped_ima
ge', 'get_offset_from_rva', 'get_overlay', 'get_overlay_data_start_offset', 'get_physical_by_rva', 'get_qword_at_r
va', 'get_qword_from_data', 'get_qword_from_offset', 'get_resources_strings', 'get_rich_header_hash', 'get_rva_fro
m_offset', 'get_section_by_offset', 'get_section_by_rva', 'get_string_at_rva', 'get_string_from_data', 'get_string
_u_at_rva', 'get_warnings', 'get_word
>>> help(pefile.PE.is_exe)
Help on function is_exe in module pefile:

is_exe(self)
    Check whether the file is a standard executable.

This will return true only if the file has the IMAGE_FILE_EXECUTABLE_IMAGE flag
set and the IMAGE_FILE_DLL not set and the file does not appear to be a driver
either.
```

We can return to the help menu to read more about the methods and attributes of the PE class. Alternatively, we can view a summary of this information by typing `dir(pefile.PE)`. An excerpt of this output is shown on this slide.

The `dir()` command output is a bit cluttered, but a quick scan of the text hints at the module's capabilities. For example, we see references to `get_impash`, `get_overlay`, and `is_exe`. If we want more information about a specific method, we can use the `help()` function.

By the way, functions that begin and end with double underscores are considered “magic methods”. They are basically reserved and can be safely ignored for now.

Pefile: Load a Program

- Within a Python interactive shell, type these commands to get started:

```
>>> import pefile
>>> target = pefile.PE("C:\\Windows\\notepad.exe")
>>> target.get_imphash()
'670212bd5fae78855c331eddeffdd4eb'
>>> target.is_exe()
True
```

- When an executable is loaded, we can begin calling methods to learn about the target program and explore its structure.
- It's helpful to have an interactive shell open while writing a script to test code and review any available documentation.

This slide demonstrates how to use the pefile module to load an executable and begin using available methods.

Extract Information about Imported DLLs and Functions

`help(pefile.PE)` output describes how to access the various directories entries, including one associated with the program's imports.

```
Directory entries will be available as attributes (if they exist):  
(no other entries are processed at this point)
```

```
DIRECTORY_ENTRY_IMPORT (list of ImportDescData instances)  
DIRECTORY_ENTRY_EXPORT (ExportDirData instance)  
DIRECTORY_ENTRY_RESOURCE (ResourceDirData instance)  
DIRECTORY_ENTRY_DEBUG (list of DebugData instances)  
DIRECTORY_ENTRY_BASERELOC (list of BaseRelocationData instances)
```

```
DI >>> help(pefile.ImportDescData)
```

```
DI Help on class ImportDescData in module pefile:
```

```
class ImportDescData(DataContainer)  
|   ImportDescData(**args)  
|  
|   Holds import descriptor information.  
|  
|   dll:      name of the imported DLL  
|   imports:  list of imported symbols (ImportData instances)  
|   struct:   IMAGE_IMPORT_DESCRIPTOR structure
```

```
>>> for item in target.DIRECTORY_ENTRY_IMPORT:  
...     print(item.dll)  
...
```

```
b'KERNEL32.dll'  
b'GDI32.dll'  
b'USER32.dll'  
b'api-ms-win-crt-string-l1-1-0.dll'  
b'api-ms-win-crt-runtime-l1-1-0.dll'  
b'api-ms-win-crt-private-l1-1-0.dll'  
b'api-ms-win-core-com-l1-1-0.dll'  
b'api-ms-win-core-shlwapi-legacy-l1-1-0.dll'  
b'api-ms-win-shcore-obsolete-l1-1-0.dll'  
b'api-ms-win-shcore-path-l1-1-0.dll'  
b'api-ms-win-shcore-scaling-l1-1-1.dll'  
b'api-ms-win-core-rtlsupport-l1-1-0.dll'  
b'api-ms-win-core-errorhandling-l1-1-0.dll'  
b'api-ms-win-core-processthreads-l1-1-0.dll'  
b'api-ms-win-core-processord-l1-1-0.dll'
```

Although the `pefile` module includes several function to quickly extract information like an import table hash, obtaining other information may require a bit more work.

For example, let's discuss how to extract an executable's imported DLLs and functions. This is useful data we often consult when performing static file analysis.

The screenshots on this slide show how we can navigate `pefile` documentation and perform testing to get access to the list of imported DLLs. In this case, we learn that the imports are accessible via the `DIRECTORY_ENTRY_IMPORT` attribute, which is a list of `ImportDescData` instances. Reviewing documentation on the `ImportDescData` class reveals that the "dll" attribute contains the name of the imported DLL.

The code on the right iterates over the `DIRECTORY_ENTRY_IMPORT` list and prints out the "dll" attribute for each `ImportDescData` object. The output is a list of imported DLLs.

Iterate over ImportData Instances to Print DLLs and Functions

```
>>> help(pefile.ImportDescData)
Help on class ImportDescData in module pefile:

class ImportDescData(DataContainer)
|   ImportDescData(**args)
|
|   Holds import descriptor information.
|
|   dll:      name of the imported DLL
|   imports:  list of imported symbols (ImportData instances)
|   struct:   IMAGE_IMPORT_DESCRIPTOR structure
```

```
>>> help(pefile.ImportData)
Help on class ImportData in module pefile:

class ImportData(DataContainer)
|   ImportData(**args)
|
|   Holds imported symbol's information.
|   ordinal:  Ordinal of the symbol
|   name:     Name of the symbol
|   bound:    If the symbol is bound, this contains
|             the address.
```

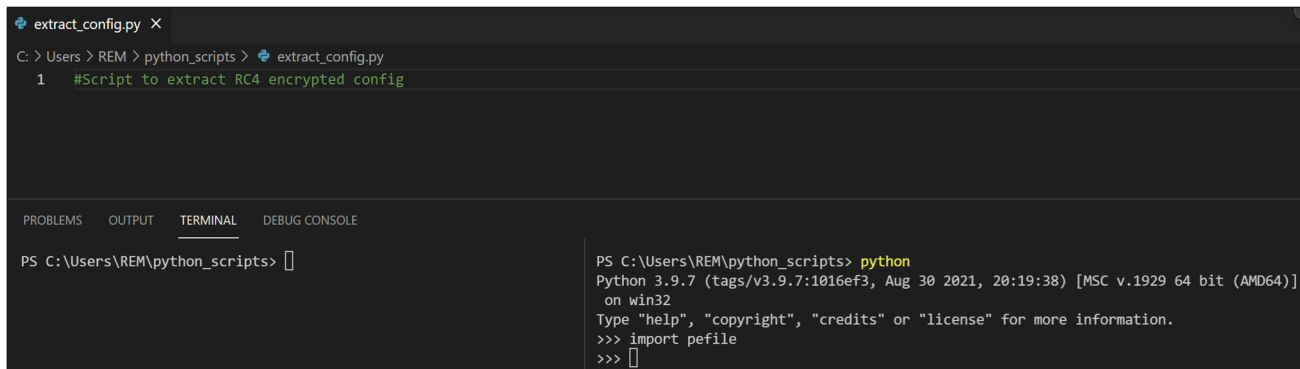
```
>>> for item in target.DIRECTORY_ENTRY_IMPORT:
...     print(item.dll)
...     for function in item.imports:
...         print(function.name)
...
b'KERNEL32.dll'
b'GetProcAddress'
b'CreateMutexExW'
b'AcquireSRWLockShared'
b'DeleteCriticalSection'
b'GetCurrentProcessId'
b'GetProcessHeap'
b'GetModuleHandleW'
```

We can supplement the output on the previous slide by adding the names of functions imported by each DLL. To obtain this additional detail, observe that each `ImportDescData` object contains an attribute “imports” that is a list of `ImportData` instances. Documentation on `ImportData` shows a “name” attribute that specifies the function name.

The code on the right makes use of this detail to iterate over the list of function names to print each one to the terminal.

With a brief introduction to `pefile` behind us, let’s return to building our configuration extractor.

VS Code: Include a Text Editor, Terminal, and Python Shell



The screenshot shows the Visual Studio Code (VS Code) interface. At the top, a file named `extract_config.py` is open in the editor. The file path is `C:\Users\REM\python_scripts\extract_config.py`. The script content is as follows:

```
1 #Script to extract RC4 encrypted config
```

Below the editor, the **TERMINAL** tab is active. It contains two panels. The left panel shows a PowerShell prompt at `C:\Users\REM\python_scripts>`. The right panel shows the output of running `python`, which displays the Python 3.9.7 version information and the prompt `>>>`. The command `>>> import pefile` has been entered, and the prompt `>>>` is visible again.

This organization allows us to both run our script and interactively explore `pefile` as needed to parse our target file.

Let's get started with building our config extractor. We will also take some time to get more familiar with the `pefile` Python module. Within VS Code, create a new file and call it `extract_config.py`. We will also use two terminal windows below the script file. On the left, we have our terminal prompt where we can view the output of running our script. This will help us check for errors and assess if our script produces the expected output. On the right, launch a Python interactive shell. Rather than diving straight into creating a script, the interactive shell is a great way to learn about available modules and perform quick testing.

Within the interactive shell, load `pefile` with the command `import pefile`. As a reminder, `pefile` is not included with a default Python install. It was installed within your Windows VMs with the command `pip3 install pefile`.

Our Script Will Import Several Python Modules

- We need to import several libraries:

```
import pefile
import argparse
from Crypto.Cipher import ARC4
```

- The pycryptodome module implements many cryptographic algorithms including RC4, Salsa20, ChaCha20, and various ECC curves.

```
cipher = ARC4.new(key)
decrypted_data = cipher.decrypt(data)
print(decrypted_data)
```

- Our script must extract the key and data.

Our script needs to import several Python modules to perform its work, including pefile, argparse, and pycryptodome.

As a reminder, the “import” statement in Python finds the specified module and provides access to the module’s functionality. The “import” statement allows access to all functions within a specified module. If you only need access to a subset of functions within a module, consider the “from” statement. As shown on this slide, our script will include the statement:

```
from Crypto.Cipher import ARC4
```

This statement imports functions from ARC4.py located in the Crypto (i.e., pycryptodome) package under the Cipher subdirectory. A “package” is a collection of related modules. You can find ARC4.py within your VM at C:\Users\REM\AppData\Local\Programs\Python\Python39\Lib\site-packages\Crypto\Cipher\ARC4.py.

Next, We Specify Arguments and Load the Target Program

- Arguments will specify the target executable and an optional output file for the extracted config:

```
parser = argparse.ArgumentParser(description="Config extractor")
parser.add_argument("-f", "--file", help="File for config extraction.", required=True)
parser.add_argument("-o", "--output", help="Output file for config.", required=False)
args = parser.parse_args()
```

- Then, we can load the specified target program.

```
target = pefile.PE(args.file)
```

Next, we will use `argparse` to specify our command line arguments. As a reminder, we want to include arguments to specify the input file (i.e., the target binary) and an output file. We can make the output file option by including `required=False` when we call `parser.add_argument()`.

The Sections Attribute Is a List of `IMAGE_SECTION_HEADER` Structures That Contain the “Name” Attribute

```
PS C:\Users\REM\python_scripts> python .\extract_config.py
-f C:\Users\REM\Desktop\file_and_more\file.exe
[IMAGE_SECTION_HEADER]
0x1C8 0x0 Name: .text
0x1D0 0x8 Misc: 0xA2D4
0x1D0 0x8 Misc_PhysicalAddress: 0xA2D4
0x1D0 0x8 Misc_VirtualSize: 0xA2D4
0x1D4 0xC VirtualAddress: 0x1000
0x1D8 0x10 SizeOfRawData: 0xA400
0x1DC 0x14 PointerToRawData: 0x400
0x1E0 0x18 PointerToRelocations: 0x0
0x1E4 0x1C PointerToLinenumbers: 0x0
0x1E8 0x20 NumberOfRelocations: 0x0
0x1EA 0x22 NumberOfLinenumbers: 0x0
0x1EC 0x24 Characteristics: 0x60000020
[IMAGE_SECTION_HEADER]
0x1F0 0x0 Name: .rdata
```

- The initial outputs show byte strings.
- We convert to UTF-8 with `decode()` and remove null bytes with `rstrip()`.

```
14 for section in target.sections:
15     print(section.Name)

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

PS C:\Users\REM\python_scripts> python .\extract_config.py
-f C:\Users\REM\Desktop\file_and_more\file.exe
b'.text\x00\x00\x00'
b'.rdata\x00\x00'
b'.data\x00\x00\x00'
b'.s7bz\x00\x00\x00'
b'.reloc\x00\x00'

15 for section in target.sections:
16     section_str = section.Name.decode().rstrip("\x00")
17     print(section_str)

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

PS C:\Users\REM\python_scripts> python .\extract_config.py
-f C:\Users\REM\Desktop\file_and_more\file.exe
.text
.rdata
.data
.s7bz
.reloc
```

Run the script from the command line to specify arguments.

To execute the script on the previous slide and pass arguments, you will need to run it from the command line within the terminal window. The output lists `IMAGE_SECTION_HEADER` structures. Each structure includes, among other attributes, the “Name” of the section. Let’s update our script to print out these names (see top-right screenshot).

The new output prints out the section names as expected, but the format requires explanation. Each section is preceded by “b” because the data is a “binary string”. If you printed the type for these values using the built-in `type()` function, the output describes the values as `<class 'bytes'>`. To convert this data to a string type, we can apply the `decode()` function. When using `decode()`, you can specify the appropriate encoding. UTF-8 is the default, so we do not need to specify the encoding in this case. An alternative to using the `decode()` function is to convert the type with the code `str(section.Name, 'utf-8')`. However, `decode()` is more common when operating on byte data. For more information on the `decode()` method, see <https://for710.com/decode>.

Because the byte data includes null bytes, it is necessary to remove those values. We can accomplish this with the string `rstrip()` method (see <https://for710.com/rstrip>).

Compare Similar Executables to Identify the Section of Interest

- Brief static analysis indicates all files have a section with an unusual section name, but the name varies.
- In our script, we could check for section names that are non-standard.

c:\users\rem\desktop\file_and_more\file.exe	property	value	value	value	value	value
indicators (33) *	name	.text	.rdata	.data	.s7bz	.reloc
virusotal (warning)	md5	AFB8170E2FD...	CEE02166B388E5...	8E933246B6D8...	42BDA57785CD...	8E13393D944...
c:\users\rem\desktop\file_and_more\propsys.e: ^	property	value	value	value	value	value
indicators (27) *	name	.text	.rdata	.data	.xl7f	.reloc
virusotal (warning)	md5	546A09CEB867...	5BE5CE9A1FD70...	7EF6B1D7AE03...	C980ED955344A...	37ECA68CD79...
c:\users\rem\desktop\file_and_more\smphost.exe	property	value	value	value	value	value
indicators (27) *	name	.text	.rdata	.data	.cfg	.reloc
virusotal (warning)	md5	77E6C3808FF2...	4D3434703AF49E...	50F2E5B47C4C...	66A29D2FD42660...	BA6DAD4B69...

Our script must identify the appropriate section within the target PE file to extract the encrypted configuration data. We know the encrypted content is in the section named ".s7bz". One approach is to iterate over the sections within the target file (i.e., file.exe) until we find a section named ".s7bz". However, this section name may change across similar samples, and if the name changes, our script will not work.

The best-case scenario is that our script successfully extracts the configuration data from file.exe *and* similar files (we will discuss how to find similar files more in the next section). Therefore, when deciding how to identify the anomalous section, it is helpful to collect several similar malware samples. This slide shows section data for file.exe and two samples that have similar functionality when compared to file.exe. All three examples have a single anomalous section, but the section name varies. Within our script, perhaps we could iterate over sections and identify the one with an uncommon section name. Let's try this approach.

Create a List of “Good” Sections to Compare Against

- To identify the anomalous section, we define a list that includes multiple string values.
- If a section name is *not* in the list, identify it for further processing.
- Our updated script correctly identifies the single unusual section.

```
15 good = [".text", ".rdata", ".data", ".reloc"]
16
17 for section in target.sections:
18     section_str = section.Name.decode().rstrip("\x00")
19     if section_str not in good:
20         print(section_str)
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
PS C:\Users\REM\python_scripts> python .\extract_config.py
-f C:\Users\REM\Desktop\file_and_more\file.exe
.s7bz
```

In the code on this slide, we first define a list of standard Windows executable section names. Then, when we iterate over the list of sections, we check each section name against our list. If a section name is not in the list, it must be the anomalous section that contains the encrypted configuration data.

Clarify Where the RC4 Key and Encrypted Data are Located

If we browse to the section name `.s7bz` within Ghidra, we find several labeled locations including one that corresponds to the encrypted config.

The screenshot displays the Ghidra interface with the `RC4` recipe selected. The `Input` pane shows a hex string: `5F 8E A2 05 BC 3D 65 F9 F5 C5 A8 A9 90 20 10 B7 69 EC 2E 09 01 65 2F B9 62 82 E4 2B A9 6F 2B 44 35 CC A5 97 8C 51 8F 79 B5 A3 19 9C C8 A7 08 C7 02 63 F6`. The `Output` pane shows a JSON object: `{ "pk": "/SvNLPYVd04yhjQWFntNHZ0bsHYzZDzRIF+Hj kQuTmE=", "pid": "33",`. The `Headers` pane shows the `.s7bz` section selected. The `Disassembly` pane shows the following assembly code:

```
0041e028 ?? 5Eh ^
0041e029 ?? 8Eh
0041e02a ?? A2h
0041e02b ?? 05h
0041e02c ?? BCh
0041e02d ?? 82h
```

The `5Eh` value is highlighted with a red box. A red arrow points to the `FUN_00401a73` label, which is annotated with the text: `FUN_00401a73 calls the RC4 function`. The `Comments` pane shows the following code:

```
byte * __stdcall FUN_00401a73
byte *
EAX:4 <RETURN>
FUN_00401a73
00401a75 JZ LAB_00401ac2
00401aa7 JZ LAB_00401ac2
00401aa9 PUSH ESI
00401aaa PUSH dword ptr [DAT_0041e024]
00401ab0 PUSH EDI=>DAT_0041e028
00401ab1 PUSH 0x20
00401ab3 PUSH DAT_0041e000
00401ab8 CALL cipher_rc4
```

Before proceeding with our script, we must understand the structure of the anomalous PE section and where the encrypted data resides. Within Ghidra, when we jump to the PE section of interest, we find several labeled locations. One labeled location corresponds to the encrypted data. There are three more labeled locations, which we will discuss shortly.

Associate Two Other Labeled Locations with the RC4 Key and Encrypted Data Size

The HW breakpoint is hit within the RC4 decryption function.

Assembly code snippet (004059FC):

```

push ebp
mov ebp, esp
sub esp, 104
push ebx
push esi
xor esi, esi
push edi
mov eax, esi
mov byte ptr ss:[ebp+eax-104], 0
inc eax
cmp eax, 100
jnb file.405A0C
mov edi, esi
xor edx, edx
mov bl, byte ptr ss:[ebp+edi-104]
movzx ecx, bl
div dword ptr ss:[ebp+c]
mov eax, dword ptr ss:[ebp+8]
movzx eax, byte ptr ds:[edx+eax]
add eax, esi

```

Registers (00401A8B):

```

EAX 0439F4A0
EBX 00231000
ECX 000066CC
EDX 0439F401

```

Memory dump (0041E000):

```

0041E000 59 4E 73 38 51 44 36 51 78 57 62 77 51 63 74 44 59s8QD6QX
0041E010 34 74 38 68 34 64 71 40 48 4D 62 41 54 44 34 68 4t8k4dqMK
0041E020 0E B0 27 66 CC 66 00 00 5E 8E A2 05 BC 82 3D 65 . "fif..A
0041E030 F9 F5 C5 A8 A9 90 20 10 B7 69 EC 2E 09 01 65 2F 06A'0.

```

Call stack (00401A8B):

```

00401A8B . call <file>
00401AC0 . mov eax, esi
00401AC2 . pop esi
00401AC3 . pop edi
00401AC4 . ret
00401AC5 . push ebp
00401AC6 . mov ebp, esp
00401AC8 . sub esp, 28
00401ACB . push ebx
00401ACC . call <file>
00401AD1 . mov ebx, eax
00401AD3 . xor eax, eax

```

Hardware breakpoint (dword, read/write) at file.0041E000 (0041E000)!

Based on our analysis in Section 2 (see the slide in Section 2 titled “The HW Breakpoint Is Hit within the RC4 Decryption Function”), we can associate two of the remaining three labeled locations with the RC4 key and the encrypted data size.

Rename Labels for Clarity, Leaving One Unknown Label

rc4_key			
0041e000	??	59h	Y
0041e001	??	4Eh	N
0041e002	??	73h	s
0041e003	??	38h	8
0041e004	??	51h	Q
0041e005	??	44h	D
0041e006	??	36h	6
0041e007	??	51h	Q
0041e008	??	78h	x
0041e009	??	57h	W
0041e00a	??	62h	b
0041e00b	??	77h	w
0041e00c	??	51h	Q
0041e00d	??	63h	c
0041e00e	??	74h	t
0041e00f	??	44h	D
0041e010	??	34h	4
0041e011	??	74h	t
0041e012	??	38h	8

DAT_0041e020			
0041e020	??	0Eh	
0041e021	??	B0h	
0041e022	??	27h	'
0041e023	??	66h	f

encrypted_size			
0041e024	undefi...	000066CCh	

encrypted_data			
0041e028	??	5Eh	^
0041e029	??	8Eh	
0041e02a	??	A2h	
0041e02b	??	05h	
0041e02c	??	BCh	
0041e02d	??	82h	

FUN_00401a73			
00401a73	PUSH	EDI	
00401a74	PUSH	dword ptr [encrypted_size]	
00401a7a	MOV	EDI, encrypted_data	
00401a7f	PUSH	EDI=>encrypted_data	
00401a80	PUSH	0x0	
00401a82	CALL	FUN_00405846	
00401a87	ADD	ESP, 0xc	
00401a8a	CMP	EAX, dword ptr [DAT_0041e020]	
00401a90	JZ	LAB_00401a96	
00401a92	XOR	EAX, EAX	
00401a94	POP	EDI	
00401a95	RET		

Some code analysis is necessary to understand the remaining labeled data.

Within the PE section of interest, there is one remaining labeled data location. To investigate, let's review the code that references this location.

At 401a8a, the data is compared to the contents of EAX. If the value in EAX is not equal to the 32-bit value at 41e020, the function returns. We know EAX often stores the return value of a function, and we see a function call shortly before this function: at 401a82, FUN_00405846 is executed. Interestingly, arguments passed to FUN_00405846 include the encrypted data size and a pointer to the encrypted data. Let's jump to FUN_00405846 for a closer look.

A Hexadecimal Value within FUN_00405846 Suggests CRC32

```

FUN_00405846
00405846  PUSH  EBP
00405847  MOV   EBP, ESP
00405849  MOV   ECX, dword ptr [EBP + param_1]
0040584c  MOV   EDX, dword ptr [EBP + param_3]
0040584f  NOT   ECX
00405851  TEST  EDX, EDX
00405853  JZ     LAB_00405880
00405856  SHL   EAX, 1
00405868  AND   EAX, 0x1
0040586b  NOT   EAX
0040586d  INC   EAX
0040586e  AND   EAX, 0xedb88320
00405873  XOR   ECX, EAX
00405875  SUB   EDI, 0x1
00405878  JNZ   LAB_00405864
0040587a  TEST  EDX, EDX
0040587c  JNZ   LAB_0040585a
    
```

0xedb88320

Search: All Maps Videos Shopping News More

<https://github.com/Michaelangel007/crc32>

Michaelangel007/crc32: CRC32 Demystified - GitHub

The reverse polynomial, 0xEDB88320, where the bits are reversed. The CRC algorithm comes in two forms: Normal initialization checks the top bit and shifts left, ...

<https://lxp32.github.io/docs/a-simple-example-crc32...>

A simple example: CRC32 calculation – The LXP32 Processor

Recipe

From Hex

Delimiter: Space

CRC-32 Checksum

Input

length: 78947
lines: 1

```

5e 8e a2 05 bc 82 3d 65 f9 f5 c5 a8 a9 90 20 10 b7 69 ec
2e 09 01 65 2f b9 62 82 e4 2b a9 6f 2b 44 35 cc a5 97 8c
51 8f 79 b5 a3 19 9c c8 a7 08 c7 02 63 f6 ec 87 56 60 bc
cd 28 92 eb 5c d7 4d 01 7f 5a 82 7c 25 63 01 3f 8f b5
67 cd 3b 6d 90 38 69 55 ca 53 cb 11 f4 ce f1 11 3a 9c c3
    
```

Output

time: 24ms
length: 8
lines: 1

6627b00e

Address	Value	Label
0041e020	??	0Eh
0041e021	??	B0h
0041e022	??	27h
0041e023	??	66h

FOR710 | Reverse-Engineering Malware: Advanced Code Analysis

Within FUN_00405846 we see code that is not immediately identifiable. It includes XOR, SUB, and AND instructions, which may indicate the function implements an algorithm. Within the function, also observe the value 0xedb88320. Online research of this hexadecimal value reveals it is associated with the CRC32 algorithm. Considering the arguments passed to the function, FUN_00405846 likely performs an integrity check of the encrypted data before proceeding with decryption. Using CyberChef, we can confirm the four-byte value located at 41e020 is the CRC32 checksum of the encrypted data.

38

© 2022 Anuj Soni

Technet24

Use the Slice Operator for Data Extraction

Addresses	Hex
0041e000	59 4e 73 38 51 44 36 51 78 57 62 77 51 63 74 44
0041e010	34 74 38 6b 34 64 71 4d 4b 4d 62 41 54 44 34 6b
0041e020	0e b0 27 66 cc 66 00 00 5e 8e a2 05 bc 82 3d 65
0041e030	f9 f5 c5 a8 a9 90 20 10 b7 69 ec 2e 09 01 65 2f
0041e040	b9 62 82 e4 2b a9 6f 2b 44 35 cc a5 97 8c 51 8f

RC4 Key (32 bytes)

Encrypted Data CRC32 Checksum (4 bytes)

Encrypted Data Size (4 bytes, little endian)

Encrypted Data (variable)

- `data[start:stop]` #start through stop-1
- `data[start:]` #start through end of array
- `Data[:stop]` #beginning through stop-1

```

17 for section in target.sections:
18     section_str = section.Name.decode().rstrip("\x00")
19     if section_str not in good:
20         section_data = section.get_data()
21
22         key = section_data[0:32]
23         print("Key: " + key.hex())
24
25         data_size = section_data[36:40]
26         data_size_int = int.from_bytes(data_size, "little")
27         print("Encrypted data size: " + str(data_size_int))
28
29         data = section_data[40:40+data_size_int]

```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```

PS C:\Users\REM\python_scripts> python .\extract_config.p
y -f C:\Users\REM\Desktop\file_and_more\file.exe
Key: 594e73385144365178576277516374443474386b3464714d4b4d
62415444346b
Encrypted data size: 26316

```

Based on our recent static code analysis, we can now accurately identify the structure of the PE section where the encrypted configuration resides. This section also contains the RC4 key, a CRC32 checksum of the encrypted data, and the size of the encrypted data.

In our Python script, we can use the slice operator to extract these individual components as shown in the code snippet. Another approach to parsing binary data is to use the Python struct library. While we will not use this approach in class, you can read more about this library here: <https://for710.com/pystruct>.

The code on this slide also demonstrates the use of the `get_data()` method. When run against a section, this method returns the data contained within the section. This is the data we slice as needed to specify the key, data size, and encrypted data.

This slide also uses the integer method `from_bytes()`. This function returns the integer represented by an array of bytes (for more information, see <https://for710.com/frombytes>).

Use the Python json Module for Beautification

extract_config.py > ...

```
1 #Script to extract RC4 encrypted config
2
3 import argparse
4 import pefile
5 from Crypto.Cipher import ARC4
6 import json ←
```

```
decrypted_str = decrypted_data.decode().rstrip("\x00")
config_data = json.loads(decrypted_str)
with open(args.output, "w") as f:
    f.write(json.dumps(config_data, indent=4))
```

```
config.txt
1 {
2     "pk": "/SvNLPYVd04yhjQWFntNHZ0bsHYz2DzRIF+HjkQuTmE=",
3     "pid": "33",
4     "sub": "331",
5     "dbg": false,
6     "fast": true,
7     "wipe": true,
8     "whl": {
9         "fld": [
10            "boot",
11            "msocache",
12            "programdata",
13            "$windows.ws",
14            "tor browser",
15            "program files (x86)",
16            "$recycle.bin",
17            "google",
18            "$windows.bt",
19            "intel",
20            "windows",
21            "windows.old",
22            "mozilla",
23            "system volume information",
24            "perflogs",
25            "application data",
26            "appdata",
27            "program files"
28        ]
29    }
30 }
```

As mentioned on the previous slide, the configuration is in JSON format. We can use the Python json module to beautify the output and add indentation as shown on the right side of this slide.

The `json.load()` method converts the decrypted content to a Python object. The `json.dump()` method generates a JSON formatted string with the specified indentation.

To view the complete config extractor script, see the file `section3.1_extract_config.py` in the `Malware\Section3` folder in your Windows VMs.

To learn more about the Python json module, see <https://for710.com/python-json>.

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 3

- Python for Malware Analysis
 - **Lab 3.1: Automating Config Extraction with Python**
- Malware Analysis with DBI Frameworks
 - Lab 3.2: Automate Payload Extraction with Frida
- Automating Analysis with Ghidra
 - Lab 3.3: Scripting with Ghidra

This page intentionally left blank.

Lab 3.1: Background Topics: pefile.PE.dump_info()

```
>>> help(pefile.PE.dump_info)
Help on function dump_info in module pefile:

dump_info(self, dump=None, encoding='ascii')
    Dump all the PE header information into human readable string.

>>> with open("output.txt", "w") as f:
...     f.write(target.dump_info())
```

```
1 -----DOS_HEADER-----
2
3 [IMAGE_DOS_HEADER]
4 0x0      0x0  e_magic:          0x5A4D
5 0x2      0x2  e_cblp:          0x90
6 0x4      0x4  e_cp:            0x3
7 0x6      0x6  e_crlc:          0x0
8 0x8      0x8  e_cparhdr:       0x4
9 0xA      0xA  e_minalloc:      0x0
10 0xC      0xC  e_maxalloc:     0xFFFF
11 0xE      0xE  e_ss:            0x0
12 0x10     0x10 e_sp:            0xB8
13 0x12     0x12 e_csum:          0x0
14 0x14     0x14 e_ip:            0x0
15 0x16     0x16 e_cs:            0x0
16 0x18     0x18 e_lfarlc:        0x40
17 0x1A     0x1A e_ovno:          0x0
18 0x1C     0x1C e_res:           0x0
19 0x24     0x24 e_oemid:          0x0
20 0x26     0x26 e_oeminfo:       0x0
21 0x28     0x28 e_res2:          0x0
22 0x3C     0x3C e_lfanew:        0xD0
```

```
242
243 [IMAGE_RESOURCE_DIRECTORY]
244 0xF800    0x0  Characteristics:      0x0
245 0xF804    0x4  TimeDateStamp:        0x0 [Thu Jan
246 0xF808    0x8  MajorVersion:         0x4
247 0xF80A    0xA  MinorVersion:         0x0
248 0xF80C    0xC  NumberOfNamedEntries: 0x0
249 0xF80E    0xE  NumberOfIdEntries:    0x1
250 Id: [0x539] (-)
251 [IMAGE_RESOURCE_DIRECTORY_ENTRY]
252 0xF810    0x0  Name:                    0x539
253 0xF814    0x4  OffsetToData:              0x80000018
254 [IMAGE_RESOURCE_DIRECTORY]
255 0xF818    0x0  Characteristics:      0x0
256 0xF81C    0x4  TimeDateStamp:        0x0 [Thu
257 0xF820    0x8  MajorVersion:         0x4
258 0xF822    0xA  MinorVersion:         0x0
259 0xF824    0xC  NumberOfNamedEntries: 0x0
260 0xF826    0xE  NumberOfIdEntries:    0x1
261 Id: [0x7A69]
262 [IMAGE_RESOURCE_DIRECTORY_ENTRY]
263 0xF828    0x0  Name:                    0x7A69
264 0xF82C    0x4  OffsetToData:              0x80000030
265 [IMAGE_RESOURCE_DIRECTORY]
266 0xF830    0x0  Characteristics:      0x0
267 0xF834    0x4  TimeDateStamp:        0x0
268 0xF838    0x8  MajorVersion:         0x4
269 0xF83A    0xA  MinorVersion:         0x0
270 0xF83C    0xC  NumberOfNamedEntries: 0x0
271 0xF83E    0xE  NumberOfIdEntries:    0x1
272 \--- LANG [0,0] [LANG_NEUTRAL,SUBLANG_NEUTRAL]
273 [IMAGE_RESOURCE_DIRECTORY_ENTRY]
274 0xF840    0x0  Name:                    0x0
275 0xF844    0x4  OffsetToData:              0x48
276 [IMAGE_RESOURCE_DATA_ENTRY]
277 0xF848    0x0  OffsetToData:              0x12058
278 0xF84C    0x4  Size:                    0x149F
279 0xF850    0x8  CodePage:                0x0
280 0xF854    0xC  Reserved:                 0x0
```

The next few slides provide additional background that is helpful for Lab 3.1.

The walk-through in this module discussed malware that embeds an encrypted configuration in a section. If the encrypted configuration is stored as a resource, however, accessing this data programmatically is a bit more complicated due to the structure of the resource section.

We can use the `pefile dump_info()` method to learn more about the structure of the resource section. This method prints all PE header information. This volume of data is best reviewed in a text editor, so the commands on this slide write the output to a file.

Our goal is to access the `IMAGE_RESOURCE_DATA_ENTRY` structure, which contains an offset to the resource data in `winbio.exe`.

Lab 3.1: Background Topics: Access Resources with pefile (I)

```
Directory entries will be available as attributes (if they exist):  
(no other entries are processed at this point)
```

```
DIRECTORY_ENTRY_IMPORT (list of ImportDescData instances)  
DIRECTORY_ENTRY_EXPORT (ExportDirData instance)  
DIRECTORY_ENTRY_RESOURCE (ResourceDirData instance)  
DIRECTORY_ENTRY_DEBUG (list of DebugData instances)  
DIRECTORY_ENTRY_BASERLOC (list of BaseRelocationData instances)  
DIRECTORY_ENTRY_TLS  
DIRECTORY_ENTRY_BOUND_IMPORT (list of BoundImportData instances)
```

```
>>> help(pefile.ResourceDirData)  
Help on class ResourceDirData in module pefile:  
  
class ResourceDirData(DataContainer)  
| ResourceDirData(**args)  
|  
| Holds resource directory information.  
|  
| struct: IMAGE_RESOURCE_DIRECTORY structure  
| entries: list of entries (ResourceDirEntryData instances)
```

```
>>> help(pefile.ResourceDirEntryData)  
Help on class ResourceDirEntryData in module pefile:  
  
class ResourceDirEntryData(DataContainer)  
| ResourceDirEntryData(**args)  
|  
| Holds resource directory entry data.  
|  
| struct: IMAGE_RESOURCE_DIRECTORY_ENTRY structure  
| name: If the resource is identified by name this  
| attribute will contain the name string. None  
| otherwise. If identified by id, the id is  
| available at 'struct.Id'  
| id: the id, also in struct.Id  
| directory: If this entry has a lower level directory  
| this attribute will point to the  
| ResourceDirData instance representing it.  
| data: If this entry has no further lower directories  
| and points to the actual resource data, this  
| attribute will reference the corresponding  
| ResourceDataEntryData instance.  
| (Either of the 'directory' or 'data' attribute will exist,  
| but not both.)
```

Similar to the approach we used to access a program's IAT, our journey to access resource data begins with a directory entry. On the top-left of this slide, the screenshot shows an excerpt of output from `help(pefile.PE)`. This explains that resources are accessible through the `DIRECTORY_ENTRY_RESOURCE` attribute. This attribute is an instance of a `ResourceDirData` object.

If we query `help()` to understand the `ResourceDirData` class, we find it includes an attribute named "entries" that is a list of `ResourceDirEntryData` instances.

If we query `help()` to understand the `ResourceDirEntryData` class, we see it includes four attributes. The data attribute points to the actual resource data, but only if there are no lower directories. If there are subdirectories, those are accessible via the directory attribute, which points to additional `ResourceDirData` objects. This means we must traverse any subdirectories before accessing the resource data. As highlighted on this slide, only one of the directory or data attributes may exist.

When we finally reach the resource data, it will be an instance of the `ResourceDataEntryData` class.

Lab 3.1: Background Topics: Access Resources with pefile (2)

```
class ResourceDataEntryData(DataContainer)
| ResourceDataEntryData(**args)
|
| Holds resource data entry information.
|
| struct: IMAGE_RESOURCE_DATA_ENTRY structure
| lang: Primary language ID
| sublang: Sublanguage ID
```

```
>>> help(pefile.PE.get_data)
Help on function get_data in module pefile:

get_data(self, rva=0, length=None)
    Get data regardless of the section where it lies on.

    Given a RVA and the size of the chunk to retrieve, this method
    will find the section where the data lies and return the data.
```

```
-----Resource directory-----
241
242
243 [IMAGE_RESOURCE_DIRECTORY]
244 0xF800 0x0 Characteristics: 0x0
245 0xF804 0x4 TimeDateStamp: 0x0 [Thu Jan
246 0xF808 0x8 MajorVersion: 0x4
247 0xF80A 0xA MinorVersion: 0x0
248 0xF80C 0xC NumberOfNamedEntries: 0x0
249 0xF80E 0xE NumberOfIdEntries: 0x1
250 Id: [0x539] (-)
251 [IMAGE_RESOURCE_DIRECTORY_ENTRY]
252 0xF810 0x0 Name: 0x539
253 0xF814 0x4 OffsetToData: 0x80000018
254 [IMAGE_RESOURCE_DIRECTORY]
255 0xF818 0x0 Characteristics: 0x0
256 0xF81C 0x4 TimeDateStamp: 0x0 [Thu
257 0xF820 0x8 MajorVersion: 0x4
258 0xF822 0xA MinorVersion: 0x0
259 0xF824 0xC NumberOfNamedEntries: 0x0
260 0xF826 0xE NumberOfIdEntries: 0x1
261 Id: [0x7A69]
262 [IMAGE_RESOURCE_DIRECTORY_ENTRY]
263 0xF828 0x0 Name: 0x7A69
264 0xF82C 0x4 OffsetToData: 0x80000030
265 [IMAGE_RESOURCE_DIRECTORY]
266 0xF830 0x0 Characteristics: 0x0
267 0xF834 0x4 TimeDateStamp: 0x0
268 0xF838 0x8 MajorVersion: 0x4
269 0xF83A 0xA MinorVersion: 0x0
270 0xF83C 0xC NumberOfNamedEntries: 0x0
271 0xF83E 0xE NumberOfIdEntries: 0x1
272 \--- LANG [0,0] [LANG_NEUTRAL, SUBLANG_NEUTRAL]
273 [IMAGE_RESOURCE_DIRECTORY_ENTRY]
274 0xF840 0x0 Name: 0x0
275 0xF844 0x4 OffsetToData: 0x0
276 [IMAGE_RESOURCE_DATA_ENTRY]
277 0xF848 0x0 OffsetToData: 0x12058
278 0xF84C 0x4 Size: 0x149F
279 0xF850 0x8 CodePage: 0x0
280 0xF854 0xC Reserved: 0x0
```

Continuing our analysis from the previous slide, we can query `help()` for more information on the `ResourceDataEntryData` class. There, we observe a “struct” attribute. This attribute contains the `IMAGE_RESOURCE_DATA_ENTRY` structure we saw in our earlier `dump_info()` output (highlighted on the slide). This structure contains `OffsetToData` and `Size` fields. Passing these values to `get_data()` will return the resource data we desire.

Lab 3.1

Automating Config Extraction with Python

Please begin Lab 3.1 now.

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 3

- Python for Malware Analysis
 - Lab 3.1: Automating Config Extraction with Python
- **Malware Analysis with DBI Frameworks**
 - Lab 3.2: Automate Payload Extraction with Frida
- Automating Analysis with Ghidra
 - Lab 3.3: Scripting with Ghidra

This page intentionally left blank.

Malware Analysis with DBI Frameworks

This module discusses how to use Dynamic Binary Instrumentation (DBI) frameworks to help automate malware analysis.

Malware Analysis with DBI Frameworks: Module Objectives

- Gain familiarity with the concept of DBI frameworks.
- Understand how DBI-based tools can help automate reverse engineering.
- Use a DBI-based tool to monitor API calls.
- Use a DBI framework's Python bindings to script a common malware analysis workflow.

This slide describes the objectives of this module.

DBI Frameworks: Run a Program and Interact with Its Internals

- This includes hooking functions, observing API calls, assessing function inputs/outputs, and modifying instructions and data during execution.
- DBI-based tools are often used to assess proprietary programs, evaluate performance, and discover vulnerabilities.
- DBI frameworks are available for both desktop and mobile OSs and most include well documented APIs to facilitate tool development.



Malware reverse engineers perform dynamic code analysis to inspect a program during execution. This typically involves using a debugger to monitor a suspect process. A complementary approach is to interrogate a running process using Dynamic Binary Instrumentation (DBI) frameworks. While a debugger allows you to attach to a process, DBI techniques allow you to inject and execute code within a process to examine its internals.

Well-known DBI frameworks include DynamoRIO, Intel's Pin, and Frida. These frameworks are often used to assess proprietary programs and evaluate program performance, but they can also be applied to accelerate malware analysis. They allow analysts to hook functions to observe API calls, assess their inputs and outputs, and even modify instructions and data during execution. DBI frameworks target both desktop and mobile operating systems (i.e., Windows, macOS, GNU/Linux, iOS, Android™, and QNX) and provide well-documented APIs to facilitate tool development.

DBI Framework Capabilities Complement the RE Process

- Malware reverse engineers debug programs to inspect code and data during execution.
- DBI frameworks can help automate the debugging process.
- With DBI-based tools, we can monitor APIs of interest and automatically extract argument and return values.
- We can use DBI framework capabilities to automate common RE tasks including deobfuscating code, dumping payloads, and decrypting data.

DBI frameworks are versatile, and one application is to automate common malware analysis tasks. Reverse engineers often find themselves monitoring functions, observing their inputs and outputs, and taking further action based on what is discovered. DBI-based tools can play a key role in automating these tasks, including deobfuscating code, dumping payloads, and decrypting data.

Frida Background

- While there are several options to choose from, we will pursue the Frida DBI framework to automate our reverse engineering efforts.
- Frida is a free and open-source DBI framework we can use to rapidly automate malware analysis workflows.
- Frida injects JavaScript into a running program to monitor and/or modify function arguments and return values.
- Frida includes helpful command line tools to get started immediately, but the framework's power is best experienced using its Python bindings.
- The framework is available in your VM, but in another environment you would install Frida with the command: `pip3 install frida-tools`.

This module demonstrates how to use Frida to automate reverse engineering workflows. We will discuss Frida's key features and explain the core components of a Frida Python script. With this knowledge, analysts can rapidly build custom tools to perform binary analysis.

Frida is a free and open-source software created by Ole André V. Ravnås. It allows analysts to inject JavaScript into programs to observe, intercept, and modify the inputs and outputs of function calls during execution. It works on a variety of desktop and mobile operating systems. Frida provides command line tools for those who want immediate access to its benefits, but the framework's functionality and flexibility are best experienced using the available Python bindings.

Frida requires a Python 3 install on a Windows, macOS, or GNU/Linux operating system. To install Frida, run the following command from an internet-connected machine: `pip3 install frida-tools`.

Python 3 and Frida are already installed within your VMs.

Frida-Trace Provides Benefits of the Frida Framework with Minimal Coding

- With frida-trace, you can quickly intercept APIs of interest.
- Use this command line format:
 - `frida-trace -f <target> -i <module Name>!<functions(s)>`
 - Example: `frida-trace -f run.exe -i KERNEL32.DLL!CreateFile*`
- **-f**: Target file to execute
- **-i**: Function(s) to intercept
- *****: Wildcard to monitor multiple APIs, but this may include *too* many
- Including the module name is optional but recommended.

Frida-trace is one of several command-line tools in the Frida framework that has clear benefits for malware analysis. Malware analysts often spend time tracing API calls; this tool helps automate tracing by allowing analysts to display and process the inputs and outputs of a specified function. Frida is not an emulator framework, which means it executes the target program. For this reason, frida-trace should be used in an isolated environment when performing malware analysis.

To spawn a process and begin tracing function calls, use the following command line format: `frida-trace -f <program name> -i <MODULE NAME>!<Function(s) to monitor>`.

To view the help output, use the “-h” command line flag. You can also visit <https://for710.com/fridatrace> for more information.

There are numerous APIs worth tracing for malware analysis. For example, consider this command:
`frida-trace -f run.exe -i KERNEL32.DLL!CreateFile*`

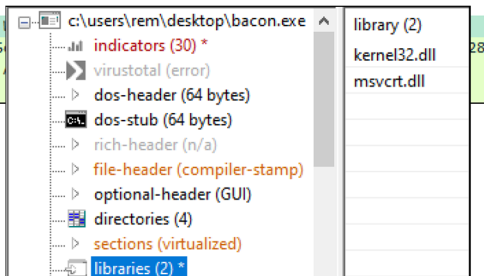
This command line will launch run.exe and monitor calls to any API that begins with “CreateFile” in kernel32.dll. Key details about this command line to note are:

- The `-f` flag specifies the target file to spawn.
- Use `-i` flag specifies the function to monitor.
- Including the module name is optional but yields more concise and targeted output. The module name must be typed in all capital letters.
- The asterisk (*) is a wildcard, and this is helpful to monitor both the wide character and ANSI versions of Windows APIs (e.g., `CreateFileW` and `CreateFileA`) or groups of APIs (e.g., all APIs that begin with “Crypto”). However, it is generally best to be as specific as possible to avoid an overwhelming amount of information from too many API calls.

Example: Use frida-trace to Assess Libraries Loaded at Runtime

No.	Time	Source	Destination	Protocol	Length	Info
77	3.284498	192.168.151.2	46.173.214.102	TCP	66	51542 → 80 [SYN] Seq=0
78	3.284731	46.173.214.102	192.168.151.2	TCP	66	80 → 51542 [SYN, ACK] S
79	3.284766	192.168.151.2	46.173.214.102	TCP	54	51542 → 80 [ACK] Seq=1
80	3.284872	192.168.151.2	46.173.214.102	HTTP	404	GET /cm HTTP/1.1

When crafting frida-trace commands, specify APIs of interest to reduce noise.



```
PS C:\Users\REM\Desktop> frida-trace -f bacon.exe -i KERNEL32.DLL!LoadLibraryA -i KERNEL32.LoadLibraryW
Instrumenting...
LoadLibraryA: Loaded handler at "C:\\Users\\REM\\Desktop\\__handlers__\\KERNEL32.DLL\\LoadLibraryA.js"
Started tracing 1 function. Press Ctrl+C to stop.
/* TID 0xa74 */
1047 ms LoadLibraryA()
1047 ms LoadLibraryA()
1047 ms LoadLibraryA()
1047 ms LoadLibraryA()
```

Let's use frida-trace to take a closer look at a malware sample. The screenshots on this slide show analysis of a sample named bacon.exe with SHA-256 hash 866b2c7dc08682cc174c58bf5cfc6ca44af7696b28b8191e7ff003d97639934c. You can find this file in the Malware\\Section3 folder within your VMs.

Brief behavioral analysis of bacon.exe indicates an HTTP request. However, looking at the imports, we see only two imported DLL with functions that have no obvious capability to communicate across a network. For example, we might expect to see ws2_32.dll or wininet.dll in the list of imported DLLs, two modules that provide networking functionality. This likely indicates additional libraries are loaded at runtime with APIs resolved during execution. We can use frida-trace to quickly evaluate if LoadLibrary variants (i.e., LoadLibraryW or LoadLibraryA) are called to load additional DLLs. We could monitor a larger group of LoadLibrary API variants using "LoadLibrary*", but this results in too much noise. Some trial and error is involved in writing targeted commands, but in general, it is best to be as specific as possible when choosing APIs of interest.

The command line from this screen is run from the PowerShell command prompt, but you can use the traditional command prompt as well.

According to the frida-trace output, LoadLibraryA is called by the program. This is a good starting point, but we need more detailed output. What DLL do these calls load? To answer this question, we need to modify the "handler" referenced in the Frida output.

Modify LoadLibraryA.js to Print Out the Loaded Library

```
HMODULE LoadLibraryA(  
    LPCSTR lpLibFileName  
);
```

```
onEnter(log, args, state) {  
    log('LoadLibraryA(): ' + args[0].readAnsiString());  
},
```

lpLibFileName

The name of the module. This can be either a library module (a .dll file) or an executable

```
PS C:\Users\REM\Desktop> frida-trace -f bacon.exe -i KERNEL32.DLL!LoadLibraryA -i KERNEL32.LoadLibraryW  
Instrumenting...  
LoadLibraryA: Loaded handler at "C:\Users\REM\Desktop\__handlers__\KERNEL32.DLL\LoadLibraryA.js"  
Started tracing 1 function. Press Ctrl+C to stop.  
/* TID 0x171c */  
1047 ms LoadLibraryA(): KERNEL32.dll  
1047 ms LoadLibraryA(): ADVAPI32.dll  
1047 ms LoadLibraryA(): WININET.dll  
1047 ms LoadLibraryA(): WS2_32.dll
```

The Microsoft documentation for `LoadLibraryA` explains that the first argument points to the file or device to be created or opened. We can direct Frida to output this argument by modifying `onEnter` as shown on this slide. The `readAnsiString()` method reads a null-terminated ANSI string from the pointer passed as the first argument. Other methods to read strings from a pointer include `readCString()`, `readUtf8String()`, and `readUtf16String()`. It's worth noting that `readAnsiString()`, which is used in this example, only works within the context of the Windows operating system. Again, browse to <https://for710.com/fridanativeptr> for more detail on available methods.

If we re-run the earlier `frida-trace` command line, the output now shows which DLLs are loaded during execution. Among the four DLLs are `wininet.dll` and `ws2_32.dll`, two modules that contain networking functionality. You will have an opportunity to use a similar command when monitoring calls to `GetProcAddress` in the upcoming lab.

Example: Monitor Crypto-Related API Calls in Ransomware

```
PS C:\Users\REM\Desktop> frida-trace --decorate -f proc.exe -i Crypt*
Instrumenting...
CryptDestroyHash: Auto-generated handler at "C:\\Users\\REM\\Desktop\\__handlers__\\ADVAPI32.dll\\CryptDestroyHash.js"
CryptDestroyKey: Auto-generated handler at "C:\\Users\\REM\\Desktop\\__handlers__\\ADVAPI32.dll\\CryptDestroyKey.js"
CryptGetKeyParam: Auto-generated handler at "C:\\Users\\REM\\Desktop\\__handlers__\\ADVAPI32.dll\\CryptGetKeyParam.js"
CryptDecrypt: Auto-generated handler at "C:\\Users\\REM\\Desktop\\__handlers__\\ADVAPI32.dll\\CryptDecrypt.js"
CryptGetDefaultProviderA: Auto-generated handler at "C:\\Users\\REM\\Desktop\\__handlers__\\ADVAPI32.dll\\CryptGetDefaultProviderA.js"
CryptSetProviderA: Auto-generated handler at "C:\\Users\\REM\\Desktop\\__handlers__\\ADVAPI32.dll\\CryptSetProviderA.js"
Started tracing 39 functions. Press Ctrl+C to stop.
/* TID 0xda8 */
1516 ms CryptAcquireContextW() [ADVAPI32.dll]
1532 ms CryptImportKey() [ADVAPI32.dll]
1532 ms CryptDecrypt() [ADVAPI32.dll]
1532 ms CryptDestroyKey() [ADVAPI32.dll]
1532 ms CryptImportKey() [ADVAPI32.dll]
1532 ms CryptDecrypt() [ADVAPI32.dll]
1532 ms CryptDestroyKey() [ADVAPI32.dll]
```

– *decorate (-d)* adds the module name to the onEnter log statement.

This slide demonstrates the use of frida-trace with a different malware sample. In the screenshots on this slide, we analyzed a ransomware sample named proc.exe with SHA-256 hash d0f212d9f64ed0590919966ba837beb1de95f38d88476406b18490cd4322f21e. You can find this file in the Malware\\Section3 folder within your VMs.

As we discussed in Section 2, ransomware may use the Microsoft CryptoAPI to perform encryption and decryption. With the command on this slide, we can quickly assess if this sample uses the CryptoAPI. In this case, we used the “--decorate” flag, which adds the name of the API module to the log output. A review of the output confirms this malware uses the CryptoAPI, and the call to CryptDecrypt might pique our interest. What content is decrypted?

Example: Modify a CryptDecrypt Handler to View Decrypted Content (I)

```
/*
 * Auto-generated by Frida. Please modify to match the signature of CryptDecrypt.
 * This stub is currently auto-generated from manpages when available.
 *
 * For full API reference, see: https://frida.re/docs/javascript-api/
 */
{
  /**
   * Called synchronously when about to call CryptDecrypt.
   * @this {object} - Object allowing you to store state for use in onLeave.
   * @param (function) log - Call this function with a string to be presented to the user.
   * @param (array) args - Function arguments represented as an array of NativePointer objects.
   * For example use args[0].readUtf8String() if the first argument is a pointer to a C string encoded as UTF-8.
   * It is also possible to modify arguments by assigning a NativePointer object to an element of this array.
   * @param (object) state - Object allowing you to keep state across function calls.
   * Only one JavaScript function will execute at a time, so do not worry about race-conditions.
   * However, do not use this to store function arguments across onEnter/onLeave, but instead
   * use "this" which is an object for keeping state local to an invocation.
   */
  onEnter(log, args, state) {
    log('CryptDecrypt()');
    this.decData = args[4];
  },
  /**
   * Called synchronously when about to return from CryptDecrypt.
   * See onEnter for details.
   * @this {object} - Object allowing you to access state stored in onEnter.
   * @param (function) log - Call this function with a string to be presented to the user.
   * @param (NativePointer) retval - Return value represented as a NativePointer object.
   * @param (object) state - Object allowing you to keep state across function calls.
   */
  onLeave(log, retval, state) {
    log('Decrypted data: ' + this.decData.readAnsiString());
  }
}
```

```
BOOL CryptDecrypt(
  HCRYPTKEY hKey,
  HCRYPTHASH hHash,
  BOOL Final,
  DWORD dwFlags,
  BYTE *pbData,
  DWORD *pdwDataLen
);
```

Use *this* to reference arguments from onLeave.

We can modify the `CryptDecrypt` handler file to print the decrypted content. The fifth argument passed to `CryptDecrypt` specifies the location in memory of the decrypted data. To access this argument from `onLeave`, we use “`this`” in both `onEnter` and `onLeave` when referring to the `decData` (i.e., pointer to decoded data) variable.

Example: Modify a CryptDecrypt Handler to View Decrypted Content (2)

```
Windows PowerShell
PS C:\Users\REM\Desktop> frida-trace -f proc.exe -i CryptDecrypt -o out.txt

out.txt
1 344 ms CryptDecrypt()
2 344 ms Decrypted data:
  Kernel32.dll;Wow64DisableWow64FsCache;
  dvapi32.dll;CreateProcessWithNoWindow;
3 344 ms Decrypted data: Deal with the ransom!
4
5 Decryption service is paid
6
7 Also from your servers file
  to our cloud storage

out.txt
19 COMMUNICATION METHOD:
20
21 To decrypt your computer, you need to download the TOR browser at
  https://www.torproject.org/download/
22
23 Install it and visit our website for further action
  http://paymen45oxzpnouz.onion/
24
25 359 ms Decrypted data:
  ntdll.dll;NtQueryObject;NtQuerySystemInformation;RtlGetVersion;Kernel32.dll;
  ;GetFinalPathNameByHandleW;QueryFullProcessImageNameW;
26 359 ms Decrypted data: sc delete vmickvpexchange
27 sc delete vmicguestinterface
28 sc delete vmicshutdown
29 sc delete vmicheartbeat
```

After modifying the CryptDecrypt handler, we can run frida-trace to review decrypted content. Note the use of the “-o” command line flag to create a log file. The decrypted content includes references to DLLs, APIs, ransom note text, and commands.

Keep in mind that, in the case of ransomware, generating an output file might be problematic if the text file is immediately encrypted.

Frida-Python

- A Python script will give us more control over execution and make it easier to troubleshoot issues.
- We can intercept multiple APIs and store arguments and variables across function calls to automate analysis.
- Our script will:
 - Accept a target executable on the command line.
 - Execute the program and attach to the spawned process.
 - Include JavaScript, which includes functions to intercept and onEnter/onLeave code.
 - Inject the JavaScript into the target process.
 - Wait to receive any messages from the target process.

Frida-trace is a great way to initially benefit from the framework. However, writing our own script using Frida's Python bindings provides more control and flexibility over API monitoring. In the next part of this module, we will discuss how to harness Frida-Python to automate more complex aspects of our reverse engineering workflow. The goal is to create a script that will perform the following items:

- Accept a target executable on the command line.
- Execute the program.
- Attach to the spawned process.
- Specify JavaScript code to inject into the target process. This will hook `VirtualAlloc` and `VirtualProtect` and include code to send messages from within the target process to the Python process.
- Inject the JavaScript code into the target process.
- Resume the process.
- Wait to receive any messages from the target process.

A Practical Application of Frida's Python Bindings

The screenshot displays a debugger interface for analyzing the file `c:\users\rem\desktop\bacon.exe`. The left-hand pane shows the file's structure, with the `imports (57)` section expanded to reveal the `VirtualProtect` function. The top right pane shows the assembly code for the `VirtualProtect` function, with a breakpoint set on the `jmp qword ptr ds:[<&VirtualProtect>]` instruction. The bottom right pane shows a memory dump at address `0000000001400000`, which contains the string `MZARUH.ãh.ì...H` and the text `This program cannot be run in DOS mode`.

Let's return to `bacon.exe` and continue our analysis of this file. This analysis will demonstrate an opportunity to automate a common reverse engineering workflow using Frida's Python bindings.

Among `bacon.exe`'s imports is `VirtualProtect`, and this API interests us in malware analysis because it is used to change the permissions (e.g., read, write, execute) of a memory region. If a region in memory is updated to be executable, it might include code that was unpacked, decoded, and/or decrypted at runtime. For a similar reason, `VirtualAlloc` is a good API to investigate. You will have an opportunity to trace that API in the upcoming lab.

A common next step is to transition to a debugger to assess what content is present in memory when `VirtualProtect` is called. To this end, we load `bacon.exe` into `x64dbg` and set a breakpoint on `VirtualProtect`. Then, we run the program, and each time we encounter a call to `VirtualProtect`, we view its arguments. The first of four arguments passed to `VirtualProtect` specifies the starting address of the memory region whose permissions will change. The third parameter specifies the new permission to be applied (Microsoft refers to this as the "memory protection constant", listed here: <https://for710.com/memory-protection-constants>).

The first time we encounter `VirtualProtect`, observe the protection constant of `0x20`, which includes executable permissions. If we dump the starting address to the dump window, we see a region beginning with "MZ" and the string "This program cannot be run in DOS mode", two indications that this content might be a Windows executable.

Dump the Memory Region to Confirm a Windows Executable

Dump 1

Dump 2

Dump 3

Dump 4

Address	Hex
0000000000140000	4D 5A 41 52 55 48 89
0000000000140010	8D
0000000000140020	FF
0000000000140030	F9
0000000000140040	0E
0000000000140050	69

Memory Map

CPU

Log

Notes

Address	Size	Ir
0000000000140000	0000000000040000	
00000000001C0000	000	Follow in Disassembler
00000000001C5000	000	Follow in Dump
0000000000200000	000	Dump Memory to File
00000000003BC000	000	

c:\users\rem\Desktop\bacon_0000000000140000.bin

indicators (50) *

virusotal (error)

dos-header (64 bytes)

imports (201) *

exports (ReflectiveLoader)

exceptions (680)

tls-callbacks (n/a)

relocations (726)

resources (n/a)

strings (4088)

debug (n/a)

manifest (n/a)

property

value

md5

29DA1E86F69F7982C7D0427E09D5EFE6

sha1

7B0C4E9208C4CF29DDA068A235CE0997B9B6A6B9

sha256

7D784D8D9F39F3706C7145739178D15010078645DB43B7215E29148

size without overlay

100000 (bytes)

entropy

6.365

imphash

n/a

signature

n/a

entry-point

48 89 5C 24 08 48 89 74 24 10 57 48 83 EC 20 49 8B F8 8B DA 48 8B F1 83 FA 01 75 05 E8 CF 6C 00 00

file-version

n/a

description

n/a

file-type

dynamic-link-library

cpu

64-bit

To confirm the memory region contains a Windows executable, we follow it in the memory map and dump it to disk. Finally, we can load the dumped binary into PeStudio, which confirms this file is a 64-bit DLL.

Automate the Recent Analysis with a Frida-Python Script

Our script should automate our debugging workflow:

- Monitor calls to the VirtualProtect API.
- Evaluate if the memory region specified by the first argument contains a Windows executable (e.g., “MZ”).
- Dump the data if there is a match.
- Optional: Check if the protection constant includes executable permissions.

This slide lists the activities we want our Frida script to automate.

Key Components to a Frida-Python Script (2)

Next, the code below specifies the JavaScript code to be injected:

```
script = session.create_script("""
    Module.load('KERNEL32.DLL');
    var vpFunctionAddress = Module.getExportByName("KERNEL32.DLL",
        "VirtualProtect");

    Interceptor.attach(vpFunctionAddress,
    {
        onEnter: function (args) {
            console.log("\\\\nFunction called");
        },
        onLeave: function (retval) {
            console.log("\\\\nFunction returned");
        }
    });
""")
```

After the target process is launched, we must specify the JavaScript to be injected into the target process. In Python, we execute `session.create_script()` and provide the JavaScript code.

The JavaScript includes several key functions:

- `Module.load()`: Loads the specified module.
- `Module.getExportByName()`: Returns the address of the specified API, which is necessary to intercept calls.
- `Interceptor.attach()`: Intercepts calls to the specified address (returned from `Module.getExportByName`) and specifies the previously discussed `onEnter` and `onLeave` functions.
- `Console.log()`: Logs messages to the console during execution.

Key Components to a Frida-Python Script (3)

Finally, the script injects the JavaScript into the target process, resumes the process, and prepares to read any log messages:

```
script.load()  
frida.resume(pid)  
sys.stdin.read()  
session.detach()
```

Lastly, the script loads the injected code into the target process, resumes the process, and receives any logged messages.

Build upon the Basic Script to Dump a Windows Executable (I)

First, we add JavaScript try...catch statements to detect any errors.

```
try {  
    Module.load('KERNEL32.DLL');  
} catch (err) {  
    console.log(err);  
}  
  
try {  
    var vpExportAddress = Module.getExportByName("KERNEL32.dll", "VirtualProtect");  
} catch (err) {  
    console.log(err);  
}
```

To generate a custom script that dumps a Windows executable referenced by VirtualProtect, we must customize the basic script template we just discussed. The code discussed in the upcoming slides is available in your Malware\Section3 folder in the file named section3.2_dump_mz_vp.py.

First, let's add some error detection in case a module does not load properly, or a function is not intercepted as expected. We can use JavaScript try...catch statements for this as shown on the slide above. This is not absolutely necessary, and it does add some bloat to the code, but it helps troubleshoot issues and can save a lot of time.

Build upon the Basic Script to Dump a Windows Executable (2)

- Then, we capture relevant VirtualProtect arguments and log them.
- In addition, we print a hexdump of the memory region referenced.

```
var vpAddress = args[0];
var vpSize = args[1].toInt32();
var vpProtect = args[2];

console.log("\nVirtualProtect called!");
console.log("\tAddress: " + vpAddress);
console.log("\tSize: " + vpSize);
console.log("\tNew Protection: " + vpProtect)

console.log("\n" + hexdump(vpAddress));
```

When performing manual debugging of `bacon.exe`, we paused at each call to `VirtualProtect` and reviewed its arguments. Specifically, we dumped the starting address to a dump window and checked the new permissions applied. In our Frida script, we need to access and print out this information using the code shown. We also access the size of the region, which will be useful when it comes time to dump data to disk.

Note our use of the function `toInt32()`, which converts the second argument (i.e., the size of the region) to an integer. We also take advantage of the `hexdump()` JavaScript function to print both hexadecimal and ASCII output located at the specified memory region.

Build upon the Basic Script to Dump a Windows Executable (3)

Finally, check if the first two bytes are “MZ”; and dump the contents if a match is found.

```
if (vpAddress.readAnsiString(2) == "MZ"){  
  
    var someBinData = vpAddress.readByteArray(vpSize);  
  
    var filename = vpAddress + "_mz.bin";  
    var file = new File(filename, "wb");  
    file.write(someBinData);  
    file.flush();  
    file.close();  
  
    console.log("\\nDumped file: " + filename);  
}
```

This code uses the Frida function `readAnsiString()` to read the first two bytes of the memory region and check if they match “MZ”. If so, it uses the function `readByteArray()` to read the memory contents and dump the file to disk. Note that the argument passed to `readByteArray()` uses the size of the region to read in the appropriate number of bytes.

Script Output

```

PS C:\Users\REM\Desktop> python dump_mz_vp.py bacon.exe

VirtualProtect called!
Address: 0x3310000
Size: 260608
New Protection: 0x20

0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
03310000 4d 5a 41 52 55 48 89 e5 48 81 ec 20 00 00 00 48 MZARUH...H...H
03310010 8d 1d ea ff ff ff 48 89 df 48 81 c3 f4 63 01 00 .....H...H...c...
03310020 ff d3 41 b8 f0 b5 a2 56 68 04 00 00 00 5a 48 89 ..A...Vh...ZH.
03310030 f9 ff d0 00 00 00 00 00 00 00 00 00 f8 00 00 00 .....
03310040 0e 1f ba 0e 00 b4 09 cd 21 b8 01 4c cd 21 54 68 .....!..L..!Th
03310050 69 73 20 70 72 6f 67 72 61 64 20 63 61 6a 6a 6f is program cba...
03310060 74 20 62 65 20 72 75 6d 6f 64 65 2e 0d 0d
03310070 6d 6f 64 65 2e 0d 0d
03310080 8c 6b 6e 52 c8 0a 00
03310090 ae e4 d2 01 50 0a 00
033100a0 39 cc cf 01 e1 0a 00
033100b0 39 cc cd 01 c2 0a 00
033100c0 c8 0a 01 01 14 0a 00
033100d0 ae e4 ca 01 c9 0a 00
033100e0 52 69 63 68 c8 0a 00
033100f0 00 00 00 00 00 00 00
Dumped file: 0x3310000_mz.bin
  
```

This script could be modified to search for and dump shellcode.

property	value
md5	799B47D9451E59DDAA8E2FD0F7A17522
sha1	A2C9C53B4FBD5A9E7D5E559E632639E5F8B5202E
sha256	5BFCEDAB08E72CB27D0E1F83F666A04109E69488387681E9C5E718CBA5730F0
description	n/a
file-type	dynamic-link-library
cpu	64-bit
subsystem	GUI
compiler-stamp	0x5DE8F1AD (Thu Dec 05 12:01:49 2019)
debugger-stamp	n/a

Running the script produces the output shown above. Loading the dumped content into PeStudio confirms it is a 64-bit DLL. Comparing this recently dumped executable with the file we dumped using a debugger reveals that the executable produced via Friday-Python does not have an overlay. This is because our script uses the size passed to VirtualProtect to determine how many bytes to read and dump. The result is that our Frida-Python script is more precise in its approach to dumping deobfuscated content.

To view the complete script, see the file `section3.2_dump_mz_vp.py` in the `Malware\Section3` folder in your Windows VMs.

This example script could also be modified to search for and dump shellcode. Instead of searching for “MZ” at the beginning of a memory region, the script could search for common shellcode bytes. We discussed common shellcode bytes in Section 1 of this course.

To view a Frida script that detects and dumps shellcode, see this blog post written by Anuj Soni: <https://for710.com/bbfrida>.

During Malware Analysis, We Monitor Memory Regions to Identify Interesting Content

The screenshot displays a debugger interface for analyzing the file `c:\users\rem\desktop\bacon.exe`. The left sidebar shows the file's structure, including indicators (31), headers (DOS, rich, file, optional), directories (4), sections (virtualized), libraries (2), and imports (57). The middle pane shows the function list for `VirtualAlloc`, with the instruction pointer (RIP) at `00007FFF7B528500` and the instruction `jmp qword ptr ds:[&VirtualAlloc]`. The right pane shows the register values for RAX, RBX, and RCX, and a dump window for Dump 1 through Dump 5 and Watch 1, displaying memory addresses, hex values, and ASCII representations.

Address	Hex	ASCII
0000000000140000	4D 5A 41 52 55 48 89 E5 48 81 EC 20 00 00 00 48	MZARUH.ãH.ì...H
0000000000140010	8D 1D EA FF FF FF 48 89 DF 48 81 C3 F4 63 01 00	..ëÿÿÿH.ßH.ÃôC..
0000000000140020	FF D3 41 B8 F0 B5 A2 56 68 04 00 00 00 5A 48 89	ÿôA,ðµçvh...ZH.
0000000000140030	F9 FF D0 00 00 00 00 00 00 00 00 00 00 00 00	ÿÿð.....ø....
0000000000140040	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	..º...í!..Lí!Th
0000000000140050	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno

In the last example, we identified a Windows executable when looking at analyzing calls to `VirtualProtect`. Sometimes, a more productive approach is to monitor multiple regions allocated via `VirtualAlloc`. For example, when we performed some initial static file analysis of `bacon.exe`, we noted it imported `VirtualAlloc` in addition to `VirtualProtect`. If we reviewed calls to `VirtualAlloc` within a debugger, we could dump each region allocated to a dump window and observe if any interesting data appears at that location. Eventually, we would identify the same Windows executable.

Using `VirtualAlloc` as a pivot into the code instead of `VirtualProtect` is sometimes necessary. The challenge with this approach, however, is that it requires us to track multiple regions in memory and repeatedly evaluate if any execute content was placed in one of the allocated regions. Let's explore this approach.

One Approach to Tracking Multiple Allocated Regions

```
Module.load('KERNEL32.DLL');
var vaExportAddress = Module.getExportByName("KERNEL32.dll", "VirtualAlloc");

var memRegions = [];

Interceptor.attach(vaExportAddress,
{
    onEnter: function (args) {
        this.vaSize = args[1].toInt32();

        for(var i = 0; i < memRegions.length; i++)
        {
            var firstBytes = memRegions[i].memBase.readAnsiString(2);

            if (firstBytes == "MZ")
            {
                //Generate hexdump
                //Dump content to disk
            }
        }

    },
    onLeave: function (retval) {
        memRegions.push({memBase:ptr(retval), memSize:this.vaSize});
    }
});
```

To track multiple allocated regions, we could create an array of memory regions to monitor. We could add to this array when a new region is allocated and check previously allocated regions for evidence of code or data. This automates the common malware analysis workflow of monitoring multiple regions in memory during execution.

To track multiple sections, we could use the approach described on this screen. For brevity, previous discussed code is referred to in comments but not included. Try...catch statements are also excluded.

This code declares an array `memRegions` that tracks allocated memory regions. Whenever `VirtualAlloc` returns, it pushes (i.e., adds) to the array. Each element has two properties: the starting address of the region allocated (`memBase`) and the size of the region allocated (`memSize`). We use `ptr()` to convert the return value to a pointer because it is an address. We also refer to the size via “this” because the size is referenced as an argument when the function is first called, and not when it returns.

With this code, anytime `VirtualAlloc` is called, previously allocated regions will be evaluated to determine if they begin with the bytes “MZ”. If those bytes are found, the regions are dumped as described earlier in this module.

You’ll have an opportunity to implement this approach in the upcoming lab.

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 3

- Python for Malware Analysis
 - Lab 3.1: Automating Config Extraction with Python
- Malware Analysis with DBI Frameworks
 - **Lab 3.2: Automating Payload Extraction with Frida**
- Automating Analysis with Ghidra
 - Lab 3.3: Scripting with Ghidra

This page intentionally left blank.

Lab 3.2

Automating Payload Extraction with Frida

Please begin Lab 3.2 now.

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 3

- Python for Malware Analysis
 - Lab 3.1: Automating Config Extraction with Python
- Malware Analysis with DBI Frameworks
 - Lab 3.2: Automate Payload Extraction with Frida
- **Automating Analysis with Ghidra**
 - Lab 3.3: Scripting with Ghidra

This page intentionally left blank.

Automating Analysis with Ghidra

This page intentionally left blank.

Scripting with Ghidra: Module Objectives

- Explore how to automate code analysis workflows within Ghidra.
- Use Ghidra's built-in Python interpreter to explore provided APIs.
- Understand best practices for developing Ghidra Python scripts.
- Write a Ghidra Python script to automate analysis.
- Our focus is on script development and not the use of third-party scripts.

This slide describes the objectives of this module.

Extend Ghidra's Functionality with Plugins, Analyzers, Extensions, and Scripts

- This course focuses on developing scripts for rapid prototyping.
- Scripts can access disassembly and decompiler output.
- Ghidra supports scripting using both Java and Python—we will continue our focus on Python.
- Python is possible via Jython, which provides complete access to Ghidra's Java API.
- Jython only implements Python 2.7, but we can still accomplish our automation goals despite this shortcoming.

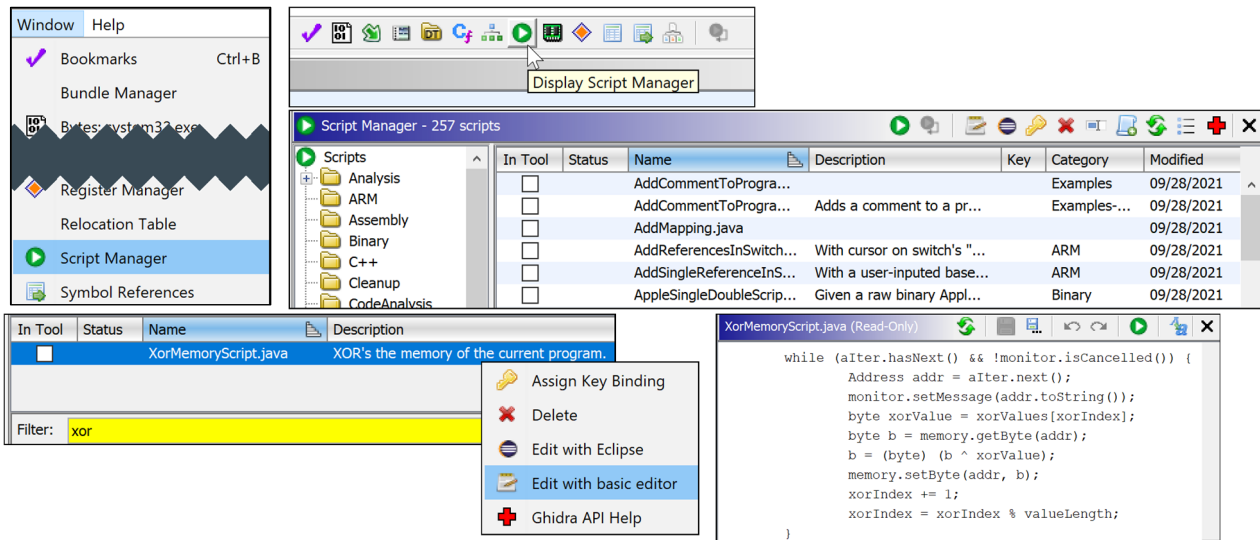


Ghidra can be extended by developing plugins, analyzers, extensions (also referred to as “modules”), and scripts. This module will focus on script development which is relatively lightweight and quick compared to other options mentioned.

Ghidra supports script development using both Java and Python. Due to our focus on Python during the last two modules and the general popularity of the language, it makes sense to build on our existing Python knowledge and use Ghidra's Python scripting capabilities. Python support is provided via Jython, a Java implementation of Python. Jython implements Python 2.7, and there are no public plans for Python 3 support. As a result, any Python development within Ghidra must be supported by Python 2.

Ghidra has hundreds of built-in scripts available to use and review. Although most built-in scripts are written in Java, the API is very similar to that provided via Python. Reading and understanding these scripts is an excellent starting point to understand how to write your own scripts within Ghidra.

Ghidra Includes Hundreds of Functional Built-In Scripts



To access Ghidra's Script Manager, you can browse to Window > Script Manager. Alternatively, you can click the green "Play" button located within the toolbar at the top of the Code Browser. The Script Manager contains hundreds of built-in scripts that are ready to use and learn from as we develop our own scripts. These scripts serve as excellent starting points for writing your own scripts. Most provided scripts are written in Java, but as mentioned earlier, Ghidra's scripting APIs are the same regardless of which language you use for script development.

To search for a script, use the Filter field at the bottom of the Script Manager. For example, if we filter by "xor" as shown on this slide, we arrive at a script that performs the XOR operation against each byte in a specified memory location. To view or edit the script, right-click on it and choose "Edit with basic editor". Do not worry if the code in the excerpt on this slide is unclear. We will cover key methods and attributes later in this module.

If you want to try running the scripts and Python commands discussed in the upcoming slides, you will need to load an executable for analysis. For this discussion, we will use system32.exe from Section 2 located at Malware\Section2\system32.zip in your VMs.

Launch the Python Interpreter Via Window > Python

- The Python terminal defaults to an instance of the GhidraScript, which extends the Flat Program API.

The screenshot shows the Ghidra Python Interpreter window on the left and the Ghidra API documentation on the right. The interpreter window has a title bar 'Python - Interpreter' and a menu bar with 'Help'. The main text area shows the following content:

```
Python Interpreter for Ghidra
Based on Jython version 2.7.2 (v2.7.2:925a3cc3b49d, Mar 13 2015)
[OpenJDK 64-Bit Server VM (Oracle Corporation)]
Press 'F1' for usage instructions
>>> help()
Searching API for ghidra.app.script.GhidraScript...
#####
class GhidraScript
    extends ghidra.program.flatapi.FlatProgramAPI
```

The 'Help' menu is open, showing options: 'Contents', 'Ghidra API Help' (selected), 'User Agreement', and 'About Ghidra'. The API documentation on the right shows the 'CLASS' tab selected, displaying the 'GhidraScript' class hierarchy:

```
Package ghidra.app.script
Class GhidraScript
    java.lang.Object
    ghidra.program.flatapi.FlatProgramAPI
    ghidra.app.script.GhidraScript
```

- Help is available in the API documentation.
- Also use tab completion in the interpreter, `help()`, `dir()`, and `type()`

The best way to start learning about Python scripting within Ghidra is to dive straight in. To test command and explore various methods, we can use the built-in Python interpreter. You can access the interpreter by browsing to Window > Python from the menu bar.

When typing commands in the Python interpreter, you are operating within an instance of the `GhidraScript` class. This class *extends* the `FlatProgramAPI`, which means the `GhidraScript` instance has all the methods and properties of the `FlatProgramAPI` we discussed earlier.

In addition to accessing API documentation using the Firefox shortcuts discussed on the previous slide, you can also browse to Help > Ghidra API help from the menu bar of the Python interpreter.

To learn more about the Ghidra API, use the built-in API help. In addition, take advantage of the Python `help()`, `dir()`, and `type()` functions.

Flat Program API dir() Output

```
dir()
['MAX_REFERENCES_TO', '_builtins_', '_doc_', '_name_', '_package_', 'addEntryPoint', 'addInstructionXref',
'analyze', 'analyzeFile', 'analyzeReferences', 'asDouble', 'asByte', 'asGhidraVersion', 'asInt', 'asLong', 'asReference',
'getAnalysisOptionsAndValues', 'getAnalysisOptionDescriptions', 'getAnalysisOptionDescription', 'getAnalysisOptions',
'getAnalysisOptionDescriptions', 'getBookmarks', 'getBytes', 'getCategory', 'getCodeUnitFormat',
'getCurrentAnalysisOptionsAndValues', 'getCurrentProgram', 'getDataAfter', 'getDataAt', 'getDataBefore',
'getDataContaining', 'getDataTypes', 'getDefaultLanguage', 'getDemangled', 'getDouble', 'getEOLComment',
'getEOLCommentAsRendered', 'getEquate', 'getEquates', 'getFirstData', 'getFirstFunction', 'getFirstInstruction',
'getFloat', 'getFragment', 'getFunction', 'getFunctionAfter', 'getFunctionAt', 'getFunctionBefore',
'getFunctionContaining', 'getGhidraVersion', 'getGlobalFunctions', 'getInstructionAfter', 'getInstructionAt',
'getInstructionBefore', 'getInstructionContaining', 'getInt', 'getLanguage', 'getLastData', 'getLastFunction',
'getLastInstruction', 'getLong', 'getMemoryBlock', 'getMemoryBlocks', 'getMonitor', 'getNamespace', 'getPlateComment',
'getPlateCommentAsRendered', 'getPostComment', 'getPostCommentAsRendered', 'getPreComment', 'getPreCommentAsRendered',
'getProgramFile', 'getProjectRootFolder', 'getReference', 'getReferencesFrom', 'getReferencesTo', 'getRepeatableComment',
'getRepeatableCommentAsRendered', 'getScriptAnalysisMode', 'getScriptArgs', 'getScriptName', 'getShort', 'getSourceFile',
'getTitle', 'setBuiltins', 'setCurrentSelection', 'setDouble', 'setEOLComment', 'setFloat', 'setInt', 'setLong', 'setPlateComment',
'setPostComment', 'setPotentialPropertiesFileLocations', 'setPreComment', 'setPropertyFile', 'setPropertyFileLocation',
'setReferencePrimary', 'setRepeatableComment', 'setScriptArgs', 'setServerCredentials', 'setShort', 'setSourceFile',
'setToolStatusMessage', 'show', 'sourceFile', 'start', 'state', 'toAddr', 'toHexString', 'toString', 'writer']
```

Because the built-in Python interpreter defaults to an instance of the `GhidraScript` class, you can view its attributes and methods by typing the `dir()` command. This slide shows an excerpt of available attributes and methods.

Access the Program API with the currentProgram Object

```
>>> help(currentProgram)
```

```
Searching API for ghidra.program.database.ProgramDB...
```

```
#####  
class ProgramDB
```

```
    extends ghidra.framework.data.DomainObjectAdapterDB
```

```
    'getBookmarkManager', 'getChangeSet', 'getChangeStatus', 'getChanges', 'getClass', 'getCodeManager', 'getCompiler',  
'getCompilerSpec', 'getConsumerList', 'getContentHandler', 'getCreationDate', 'getCurrentTransaction', 'getDBHandle',  
'getDataTypeManager', 'getDefaultPointerSize', 'getDescription', 'getDomainFile', 'getEquateTable', 'getExecutableFormat',  
'getExecutableMD5', 'getExecutablePath', 'getExecutablesSHA256', 'getExternalManager', 'getFunctionManager',  
'getGlobalNamespace', 'getImageBase', 'getIntRangeMap', 'getLanguage', 'getLanguageID', 'getListing', 'getLock',  
'getMaxAddress', 'getMemory', 'getMetadata', 'getMinAddress', 'getModificationNumber', 'getName', 'getNamespaceManager',  
'getOptions', 'getOptionsNames', 'getProgramContext', 'getProgramUserData', 'getRedoName', 'getReferenceManager',  
'getRegister', 'getRegisters', 'getRelocationTable', 'getStoredVersion', 'getSymbolTable', 'getSynchronizedDomainObjects',  
'getTreeManager', 'getUndoName', 'getUndoStackDepth', 'getUniqueProgramID', 'getUsrPropertyManager', 'globalNamespace',  
'hasGlobalNamespace', 'hasTempAddress', 'hasTempCode', 'hasTempData', 'hasTempSymbol', 'hasTempText', 'hasTempValue',  
'isReadOnly', 'isReadOnlyCode', 'isReadOnlyData', 'isReadOnlySymbol', 'isReadOnlyText', 'isReadOnlyValue', 'isReadOnlyValue',  
'setExecutableFormat', 'setExecutableMD5', 'setExecutablePath', 'setExecutableSHA256', 'setImageBase', 'setLanguage',  
'setName', 'setObjChanged', 'setPropertyChanged', 'setPropertyRangeRemoved', 'setRegisterValuesChanged', 'setTemporary',  
'sourceArchiveAdded', 'sourceArchiveChanged', 'startTransaction', 'storedVersion', 'symbolAdded', 'symbolChanged',  
'symbolTable', 'synchronizedDomainObjects', 'tagChanged', 'tagCreated', 'temporary', 'toString', 'treeManager', 'undo',  
'undoLock', 'undoName', 'undoStackDepth', 'uniqueProgramID', 'unlock', 'usrPropertyManager', 'wait']
```

To access the Program API, refer to the `currentProgram` object. The `currentProgram` object refers to the target program loaded within Ghidra. Type `help(currentProgram)` or `dir(currentProgram)` to view available methods and attributes.

When Scripting within Ghidra, Consider These Types of Content

- Functions:
 - Arguments
 - Return values
 - Function references
- Instructions:
 - Mnemonics
 - Operands
- Addresses:
 - Function location
 - Instruction location
 - Data location
 - Address references
- Comments: Adding notes

To help script an analysis workflow, consider which of the above components are used and in what order.

When performing an analysis workflow, we commonly work with the content listed on this slide. As we attempt to automate an analysis workflow, these are the same components we want to reference. The Ghidra Scripting interface provides access to each of these objects.

Ghidra APIs Provide Methods Associated with Functions

- Flat API:

- `getFirstFunction()` : Returns the first function in the program.
- `getFunctionContaining(address)` : Get the function containing an address.

- Program API:

- Identify and iterate through all functions:

```
fm = currentProgram.getFunctionManager().getFunctions(True)
for function in fm:
    ep_addr = function.getEntryPoint()
    fn_body = function.getBody()
    fn_name = function.getName()
```

Function analysis is critical to performing effective code analysis. This slide lists examples of methods in both the Flat and Program API that allow us to interact with and learn about functions.

Two helpful functions within the Flat API include:

- `getFirstFunction()` returns the first function in the program. To clarify, the “first” function is the function at the lowest virtual address, not the first function that will be executed.
- `getFunctionContaining()` returns the address of the function containing the address passed to the function.

While the Flat API provides many key methods, it lacks functions to perform some common reverse engineering workflows like reviewing function references. For this, we must use the Program API. The code excerpt at the bottom of this slide demonstrates Program API methods we can use to iterate over function references. Specifically, `currentProgram.getFunctionManager().getFunctions(True)` returns an “iterator” that includes functions within the program. An “iterator” is simply an object with multiple values that we can iterate over.

The for loop in the code excerpt queries each function in the program for key information:

- `getEntryPoint()` : Returns the address of the first instruction in the function.
- `getBody()` : Returns the set of addresses that comprise the function. You can call `contains()` against the returned value to determine if a specified address is contained within a function.
- `getName()` : Returns the name of a function.

For additional detail, search for any of these methods in the API documentation.

Referencing Addresses in a Ghidra Python Script

- Addresses may be passed to a function as an argument or returned from a function after execution.
- `toAddr()` : converts offset (long, int, or string) to an Address object.
- This code iterates through references to an address:

```
decoding_fn = toAddr(0x54a3d0)
fn_refs = getReferencesTo(decoding_fn)

for ref in fn_refs:
    from_addr = ref.getFromAddress()
```

During code analysis, we interact with addresses often. Addresses specify where an instruction, function, or data are located in memory. Addresses are also passed to functions as arguments, and they may be return values. For this reason, it is important we understand how to interact with addresses in a Ghidra Python script.

To convert a value to an address, we use the `toAddr()` function. This function can take a long (64-bits), int (32-bits), or a string. Examples of acceptable formats include:

```
toAddr(0x54a3d0)
toAddr("0x54a3d0")
toAddr("54a3d0")
```

During malware analysis, we often evaluate references to an address, such as the starting address of a function. To accomplish this task, first specify the starting address of a function or obtain the address via the Program API method `getEntryPoint()` shown on the previous slide. Then, use the Flat API method `getReferencesTo()` to return an array of references to the specified address. Finally, use a for loop to iterate through each reference and call `getFromAddress()` to acquire the address of the reference. An example of this code is shown on the slide.

An Address is an instance of the `GenericAddress` class:

```
file:///C:/Users/REM/AppData/Local/Ghidra/GhidraAPI_javadoc/10.0.4/api/ghidra/program/model/addresses/GenericAddress.html
```

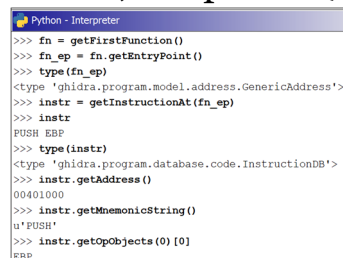
Accessing Instructions in a Ghidra Script

- The Flat API includes several methods that return an instruction:

<code>getInstructionAfter(Address address)</code>	Returns the instruction defined after the specified address or null if no instruction exists.
<code>getInstructionAfter(Instruction instruction)</code>	Returns the instruction defined after the specified instruction or null if no instruction exists.
<code>getInstructionAt(Address address)</code>	Returns the instruction at the specified address or null if no instruction exists.
<code>getInstructionBefore(Address address)</code>	Returns the instruction defined before the specified address or null if no instruction exists.
<code>getInstructionBefore(Instruction instruction)</code>	Returns the instruction defined before the specified instruction or null if no instruction exists.

- Using an instruction, we can get the address, mnemonic, or operand(s).

- `getAddress()` : Return the instruction address.
- `getMnemonicString()` : Return the mnemonic.
- `getOpObjects()` : Takes the index of the operand and returns an array with the operand value (use `toString()` to convert to a string).



```
Python - Interpreter
>>> fn = getFirstFunction()
>>> fn_ep = fn.getEntryPoint()
>>> type(fn_ep)
<type 'ghidra.program.model.address.GenericAddress'>
>>> instr = getInstructionAt(fn_ep)
>>> instr
PUSH EBP
>>> type(instr)
<type 'ghidra.program.database.code.InstructionDB'>
>>> instr.getAddress()
00401000
>>> instr.getMnemonicString()
u'PUSH'
>>> instr.getOpObjects(0) [0]
EBP
```

This slide discusses methods to acquire an instruction and probe the instruction for its address, mnemonic, or operand(s).

As a reminder, we loaded the file `system32.exe` into Ghidra for this discussion. This file has the SHA-256 hash `eecc969ba17e924093821a7c862da03f8668abe833042b6bd023fbe75fa2e0e8`. You can find it in on your VM desktop at `Malware\Section2\system32.zip`.

Acquiring the address and mnemonic is relatively straightforward, as shown in the example on this slide. Using the method to acquire the operands, however, may be a bit confusing and requires further explanation. The method that returns an instruction's operand is `getOpObjects()`, and the argument passed to it is the index of the operand (i.e., zero for the first operand, one for the second operand). This method returns an array with only one value. Therefore, to obtain the first operand of an instruction stored in the variable "instr", we type `instr.getOpObjects(0) [0]`. Note that the returned operand value is not a string—to convert it to a string, use the `toString()` method. Using the example on this slide, we could acquire the string with the command `instr.getOpObjects(0) [0].toString()`.

For more information on methods related to instructions, search the API documentation for "ghidra.program.database.code.InstructionDB".

Operating on Data at a Specified Address

- `getDataAt (Address)` : Returns data at a specified address.
- `getValue ()` : Returns the value of the data item.
- Use `try` and `except` clauses to fail gracefully if data is not defined.

```
Python - Interpreter
>>> data = getDataAt(toAddr("411ea0"))
>>> data
unicode u"SOFTWARE\\keys_data\\data"
>>> type(data)
<type 'ghidra.program.database.code.DataDB'>
>>> value = data.getValue()
>>> value
u'SOFTWARE\\keys_data\\data'
>>> type(value)
<type 'unicode'>
```

```
try:
    value = data.getValue()
except:
    print("No defined data at this location")
```

When performing manual code analysis, we often encounter pointers to data. For example, an instruction may have an operand that points to a string. We then double-click on the pointer to view the corresponding data, which may be a string. How can we programmatically perform this same action?

The Ghidra API provides two helpful methods. First, use the Flat API's `getDataAt ()` function to return the data at a specified address. Then, to acquire the data content in a format that is easier to work with, call `getValue ()` against the previously returned data. In the case that a pointer points to a Unicode string (as shown in the example on this slide), `getValue ()` will return the data value as a Unicode string.

Note that if there is no defined data at a specified address, while `getDataAt ()` will return successfully, `getValue ()` will generate an exception. For that reason, it is recommended to use `try` and `except` clauses in Python. If the code in the `try` clause generates an error, the `except` clause will be executed, where you can print a detailed message.

Adding Comments and Bookmarks

boolean	<code>setEOLComment(Address address, java.lang.String comment)</code>	Sets a EOL comment at the specified address
boolean	<code>setPlateComment(Address address, java.lang.String comment)</code>	Sets a PLATE comment at the specified address
boolean	<code>setPostComment(Address address, java.lang.String comment)</code>	Sets a POST comment at the specified address
boolean	<code>setPreComment(Address address, java.lang.String comment)</code>	Sets a PRE comment at the specified address

Bookmark	<code>createBookmark(Address address, java.lang.String category, java.lang.String note)</code>	Creates a NOTE book mark at the specified address.
-----------------	--	--

During code analysis, we know it is important to document our findings. One way to log observations is to insert comments within the Listing view. An analyst could also use Ghidra's Bookmark feature to add similar notes. The Flat API methods on this slide allow users to add various types of comments or a bookmark.

Renaming Functions

- The Program API's `setName()` method can rename a specified function:

```
void setName(java.lang.String name, SourceType source) Set the name of this function.
```

- This method's second argument specifies a symbolic constant:

Enum Constants	
Enum Constant	Description
ANALYSIS	The object's source indicator for an auto analysis.
DEFAULT	The object's source indicator for a default.
IMPORTED	The object's source indicator for an imported.
USER_DEFINED	The object's source indicator for a user defined.

```
Python - Interpreter
>>> fn = getFirstFunction()
>>> fn
FUN_00401000
>>> from ghidra.program.model.symbol import SourceType
>>> fn.setName("first_func", SourceType.USER_DEFINED)
>>> fn
first_func
```

In addition to inserting comments, we also rename functions during code analysis to create more meaningful function names. The Program API's `setName()` method can help with this task. The first argument passed to this method is the new function name, and the second argument is a symbolic constant. Symbolic constant options are shown on the bottom-left of this slide. To find the documentation within your VM, copy and paste this location into your browser within one of the Windows VMs:

file:///C:/Program Files

(x86)/Ghidra/docs/GhidraAPI_javadoc/api/ghidra/program/model/symbol/SourceType.html

To reference one of the symbolic constant options, we must import them from `ghidra.program.model.symbol.SourceType`. The Python statements on the right side of this slide show a sequence of statements that renames the first function in the target program.

Script Example: Identifying Functions with ARX Operations

```
script.py
#TODO Find functions that perform ARX operations.
#@author Anuj Soni
#@category FOR710
#@keybinding
#@menupath
#@toolbar

fm = currentProgram.getFunctionManager().getFunctions(True)
for function in fm:
    if function.getName()[3:] != "FUN":
        continue

    contains_add = False
    contains_nonzero_xor = False
    contains_rot = False

    ep = function.getEntryPoint()
    instr = getInstructionAt(ep)
    fn_body = function.getBody()

    while (instr != None and fn_body.contains(instr.getAddress())):
        operation = instr.getMnemonicString()
        if operation == "ADD":
            contains_add = True
        elif operation == "ROL" or operation == "ROR":
            contains_rot = True
        elif operation == "XOR":
            op1 = instr.getOpObjects(0)
            op2 = instr.getOpObjects(1)
            if op1 != op2:
                contains_nonzero_xor = True

        instr = getInstructionAfter(instr)

    if contains_rot and contains_add and contains_nonzero_xor:
        print("Function " + function.getName() + " contains ARX operations")
```

```
Console - Scripting
script.py> Running...
Function FUN_004034f0 contains ARX operation(s)
script.py> Finished!
```

```
FUN_004034f0
004034f0 PUSH EBP
004034f1 MOV EBP, ESP
004034f3 SUB ESP, 0xcc
004034f9 CMP dword ptr [EBP + param_4], 0x0
004034fd PUSH EBX
004034fe PUSH EDI
```

```
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
```

This slide discusses an example of a complete Ghidra Python script. This purpose of this script is to identify functions with ARX (i.e., add, rotate, xor) operations. As we discussed in Section 2, functions with ARX operations may implement an algorithm of interest. When this script is run against system32.exe, it identifies FUN_004034f0, and a closer look at this function's code confirms it implements the Salsa symmetric encryption algorithm. In fact, we discussed this same function in Section 2.

To take a closer look at the code on this slide, see the file `section3.3_ghidra_arx.py` in the `Malware/Section3` folder in your Windows VMs.

Ghidra's Headless Mode

- In headless mode, you can batch process multiple files.
- Note that to output content to a log file, you must use `println()` instead of `print()` in the Python script.
- Use the format:
 - "C:\Program Files (x86)\Ghidra\support\analyzeHeadless.bat"
`<project_location> <project_name> -import <target file> -postScript <full_path_script> -log <process_log> -scriptlog <script_log> -overwrite`

Ghidra provides a headless mode which allows you to control Ghidra via the command line. This may be helpful for processing multiple files without opening each within the GUI. For example, if you are confident in a Python script and its output, you could consider running the script against multiple files using Ghidra's headless mode.

To execute a script in headless mode you must launch `analyzeHeadless.bat`. This .bat file is located at `C:\Program Files (x86)\Ghidra\support\analyzeHeadless.bat`

The format of the command line is shown on this slide. Note that the `<project location>` directory must already exist.

For additional documentation on this feature, copy and paste these locations into Firefox's address bar within the Windows VM:

- [file:///C:/Program Files \(x86\)/Ghidra/docs/GhidraClass/Intermediate/HeadlessAnalyzer.html](file:///C:/Program Files (x86)/Ghidra/docs/GhidraClass/Intermediate/HeadlessAnalyzer.html)
- [file:///C:/Program Files \(x86\)/Ghidra/support/analyzeHeadlessREADME.html](file:///C:/Program Files (x86)/Ghidra/support/analyzeHeadlessREADME.html)

To execute our ARX script on the previous slide, use the command:

```
"C:\Program Files (x86)\Ghidra\support\analyzeHeadless.bat" projectdir projectname -import system32.exe -postScript C:\Users\REM\ghidra_scripts\script.py -log headlesslog.txt -scriptlog scriptlog.txt -overwrite
```

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 3

- Python for Malware Analysis
 - Lab 3.1: Automating Config Extraction with Python
- Malware Analysis with DBI Frameworks
 - Lab 3.2: Automate Payload Extraction with Frida
- Automating Analysis with Ghidra
 - **Lab 3.3: Scripting with Ghidra**

This page intentionally left blank.

Lab 3.3 Background: Search program.exe for Functions That Perform XOR Operations

```
#@author Anuj Soni
#@category FOR710
#@keybinding
#@menupath
#@toolbar

from ghidra.program.model.lang import OperandType
fm = currentProgram.getFunctionManager().getFunctions(True)
for function in fm:
    if function.getName()[:3] != "FUN":
        continue
    contains_nonzero_xor = False
    ep = function.getEntryPoint()
    instr = getInstructionAt(ep)
    fn_body = function.getBody()

    while (instr != None and fn_body.contains(instr.getAddress())):
        operation = instr.getMnemonicString()
        if operation == "XOR":
            op1 = instr.getOpObjects(0)
            op2 = instr.getOpObjects(1)
            if op1 != op2 and (instr.getOperandType(0) == OperandType.SCALAR or instr.getOperandType(1) == OperandType.SCALAR):
                contains_nonzero_xor = True
            instr = getInstructionAfter(instr)

    if contains_nonzero_xor:
        print("Function " + function.getName() + " contains an XOR operation of interest.")
```

```
Console - Scripting
section3.3_ghidra_nonzeroXOR.py> Running...
Function FUN_00408130 contains an XOR operation of interest.
Function FUN_004093f0 contains an XOR operation of interest.
Function FUN_00410950 contains an XOR operation of interest.
Function FUN_00455d30 contains an XOR operation of interest.
Function FUN_0046alb9 contains an XOR operation of interest.
section3.3_ghidra_nonzeroXOR.py> Finished!
```

```
LAB_00409490
00409490 MOV     AL, byte ptr [ESI]
00409492 LEA     ECX=>local_54, [EBP + -0x44]
00409495 XOR     AL, 0x8
00409497 ADD     AL, 0x3
00409499 XOR     AL, 0x54
0040949b MOVZX   EAX, AL
0040949e PUSH    EAX
0040949f CALL    FUN_0040c280
004094a4 INC     ESI
004094a5 CMP     ESI, EDI
004094a7 JNZ     LAB_00409490
```

Earlier in this module, we created a Ghidra Python script to identify functions with add, rotate, and xor (ARX) operations. We could generalize this script and update it to *only* search for xor operations given how frequently this instruction appears in encoding and decoding functions. We might also consider including a check to ensure one xor operand is a scalar (i.e., a number). This is another common occurrence in functions that encode or decode data. A script that includes these updates is called section3.3_ghidra_nonzeroXOR.py, and it is included in the Malware\Section3 folder.

If we run this script against program.exe (located in Malware\Section3), it outputs several functions of interest. This exercise will explore just one of these functions, FUN_004093f0. If we double-click on this function and review its code, we observe the loop shown on this slide, which contains several mathematical operations, including two XORs.

Lab 3.3

Scripting with Ghidra

Please begin Lab 3.3 now.

Scripting with Ghidra: Module Objectives, Revisited

- ✓ Explore how to automate code analysis workflows within Ghidra.
- ✓ Use Ghidra's built-in Python interpreter to explore provided APIs.
- ✓ Understand best practices for developing Ghidra Python scripts.
- ✓ Write a Ghidra Python script to automate analysis.

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 3

- Python for Malware Analysis
 - Lab 3.1: Automating Config Extraction with Python
- Malware Analysis with DBI Frameworks
 - Lab 3.2: Automate Payload Extraction with Frida
- Automating Analysis with Ghidra
 - Lab 3.3: Scripting with Ghidra

This page intentionally left blank.