

Lab 4: Basic PE Static Analysis

ITSC 303: Malware Analysis

Table of Contents

[Lab Outcomes 3](#_Toc22024170)

[Background Reading 3](#_Toc22024171)

[Introduction 3](#_Toc22024172)

[1.0 Digging More Deeply into the Command and Control Loop 4](#_Toc22024173)

[1.1 Using the Ghidra Decompiler 4](#_Toc22024174)

[1.2 Documenting the Exact Format of Each Command 7](#_Toc22024175)

[References 10](#_Toc22024176)

Malware Analysis

Lab 4: Basic PE Static Analysis

Lab Outcomes

This lab will focus on the following outcomes:

* Use the skills learned from previous labs.
* Understand how to generate pseudo source code in Ghidra
* Compare pseudo C code in Ghidra with assembly.
* Determine the exact command structure for a malware sample.

Background Reading

* *Practical Malware Analysis* by Michael Sikorski and Andrew Honig
  + Chapter 1: Basic Static Techniques
  + Chapter 6: Recognizing C Code Constructs in Assembly

Introduction

Decompilation, the translation from machine code to C source code, is another tool in the malware analyst’s tool box. While the decompiled C source code is not always as readable as assembly language, nor is it often very close to the original programmer-created C/C++ code, it can help in analyzing malware.

Malware often passes data between components on an infected system or between the infected system and the command and control server. Precise knowledge of these data structures is often required to fully analyze a sample or piece together information from an infected system.

In this lab, you will analyze and precisely document the data structures used in the command and control loop.

Understanding the command and control loop will allow you to better identify potentially infected machines, as well as provide some containment insight during an incident response phase. Disassembly and Decompilation take considerable amounts of time and effort, so during a live incident you may not be using this stage for initial identification and containment efforts due to time constraints. You will however want to conduct decompilation steps during root cause analysis phases, and prior to completing the report, as you may find additional indicators of compromise here, as well as further identifying how the malware took control of the system.

1. Digging More Deeply into the Command and Control Loop

## 1.1 Using the Ghidra Decompiler

In the last lab, you analyzed the **3094a3db6068be414fd060c2cfd4b10a81961d349393fe 6e07a65423d160729c.bin** file. In this lab, you will take a more detailed look at the command and control loop. Reading and understanding the functionality of command and control loops using Ghidra can be made more efficient if you take advantage of Ghidra’s decompiler. The below graphic is from IDA Pro, but the logic remains sound.

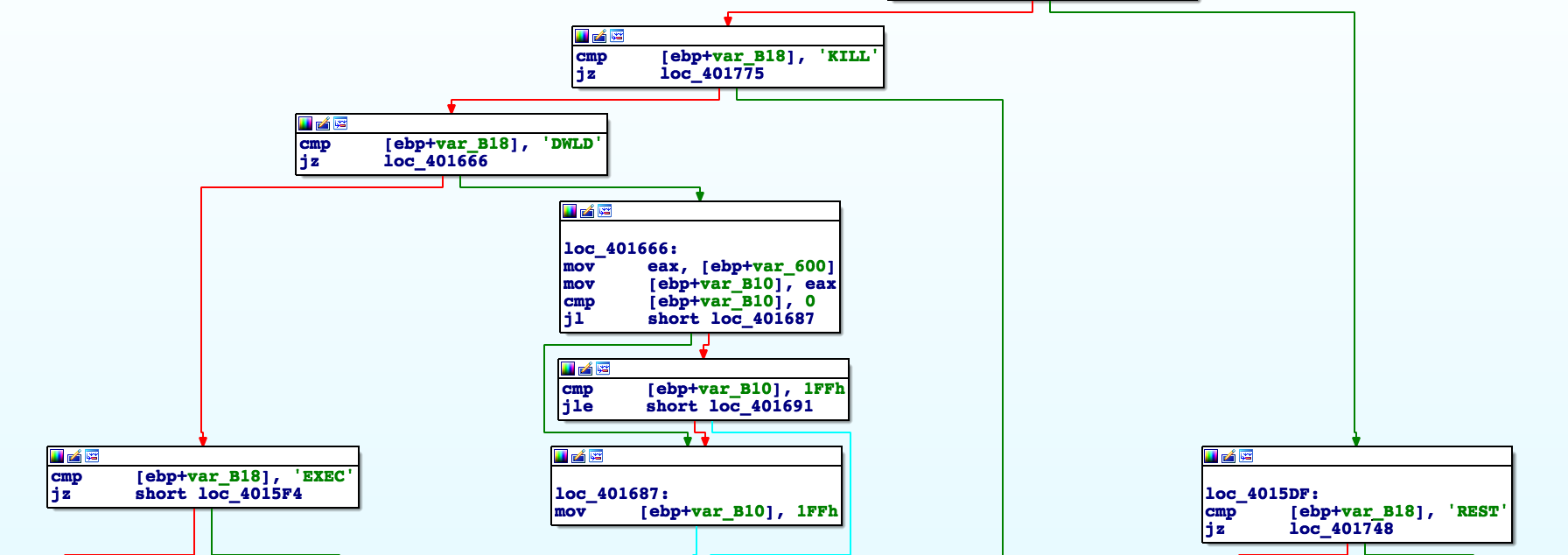
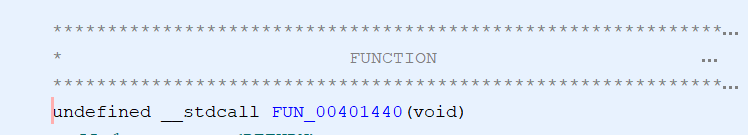


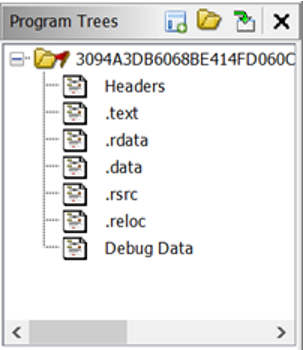
Figure 1: Portion of a Command and Control Loop

Source: IDA Pro, 2016. Reproduced and used in accordance with the fair dealing provisions in section 29 of the Canadian Copyright Act for the purposes of education, research or private study. Further distribution may infringe copyright.

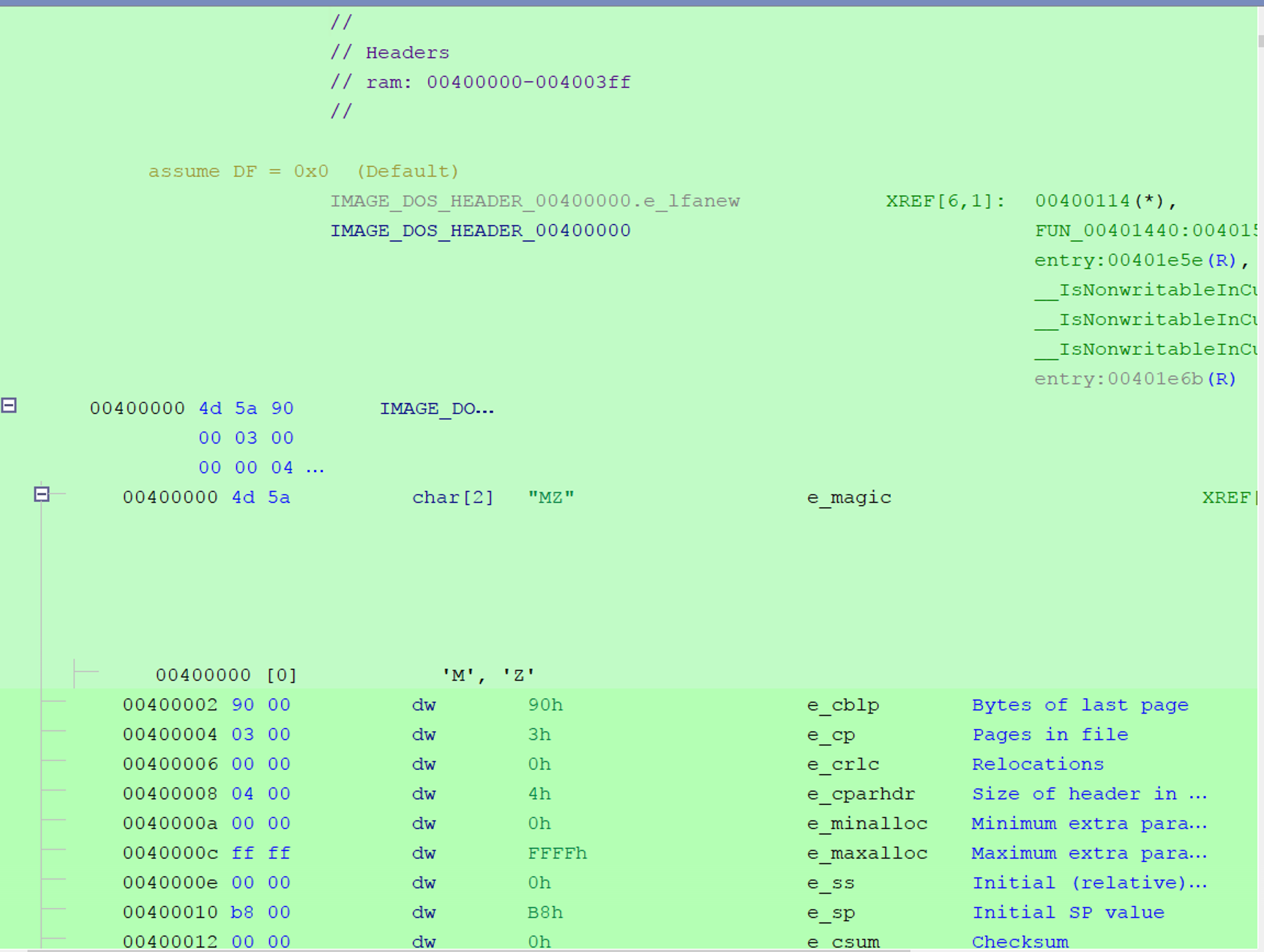
1. Open the sample in Ghidra and return to the command and control function, which can be found at 0x401440.



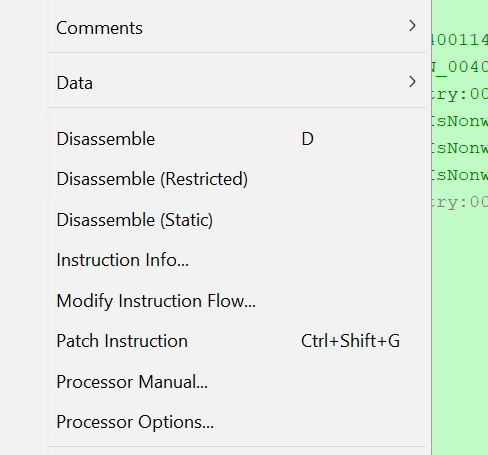
1. During the previous lab, you changed the DWORDs, which represent command and control commands, to their ASCII equivalents. For example, the DWORD 44574C44h becomes DWLD.
2. Ghidra will decompile the functions as they appear on your view, however to kick start the process a little faster complete the following steps:
   1. In the program trees section, double click the root of the binary you’re analyzing:



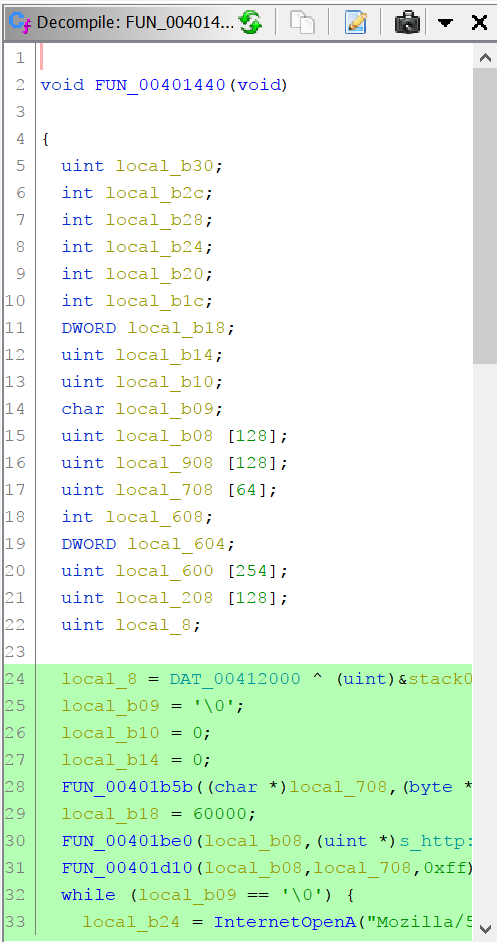
* 1. Inside the disassembly view, click anywhere inside the window, and press CTRL+A to highlight the whole program



* 1. Right click, and select Disassemble



* 1. Ghidra will go through and disassemble the entirety of the program. When you access the function, you will see the decompiled source code listed on the right hand side of the screen



Source: Ghidra v9.1, 2019. Reproduced and used in accordance with the fair dealing provisions in section 29 of the Canadian Copyright Act for the purposes of education, research or private study. Further distribution may infringe copyright.

1. Compare the pseudocode output to the assembly code. Trace execution flow between the assembly code and the pseudo code. Do you find one easier to follow than the other? (This isn’t a trick question. Many experienced malware analysts find assembly easier to follow.)

Pseudo code is much easier to follow for me than assembly

1. You can rename variables in the decompiled view in the same way you can in the assembly view. Notice that most (if not all) of your changes to the assembly view carry over to the decompiled source code.

## 1.2 Documenting the Exact Format of Each Command

Consider the common command “copy,” used to copy a file from one location to another. Most operating systems implement a form of the copy command as follows:

copy from to

The mnemonic “copy” identifies the command (usually a verb). The copy command needs at least two command line arguments, what to copy (from) and where to copy it (to). It doesn’t make sense with only one argument. In addition, you know when the command verb ends and the first argument begins because they are delimited by a space.

Within a program, commands (or messages) are passed by way of functions (usually on the stack or in registers). Commands or messages between programs are passed using a data structure agreed to by the sender and the receiver.

You need to identify the exact format for each of the four commands that the malware sample uses. You don’t have a pcap of the malware’s communication, so use only the assembly code and the decompiled code.

The first table for the REST command is filled out for you:

|  |  |  |
| --- | --- | --- |
| **Command: REST** | | |
| **Offset** | **Data** | **Description** |
| 0x4015DF | 0x54534552 (REST) | The command verb used to instruct the command and control loop to sleep |
| 0x401769 | 32bit signed integer | The number of milliseconds to sleep, the value must be >=0 and <0x100000. If the value is outside of this range, the value 60000 (0xea60) is used |

**Goal:** Fill out the tables for the remaining three commands.

**Notes:**

* Not all parts of the command may appear at fixed offsets.
* Delete offsets from the table that are not used by the command.
* By using these tables, you should be able to create valid commands for the malware.
* Each command may have fewer elements than you have boxes on the page. Simply delete the rows you don’t need or add more as you require. The number of rows under each command is **NOT** an indication of how many elements you need to uncover.
* Some thing to keep in mind. String functions may require buffer space for processing, and often data structures such as arrays can be useful for loading contents and storing them in order.

|  |  |  |
| --- | --- | --- |
| **Command: KILL** | | |
| **Offset** | **Data** | **Description** |
| 0x004015AE | 0x4b494c4c  Kill in little endian | KILL means to kill (end) a process forcibly |
| 0x00401775 | byte local\_b09  Unsigned Byte | It is a variable that is an unsigned byte, which if it did not get set to 1 then it accesses a internet connection, ie. It downloads something |
| 0x004015AE | uint local\_608  Unsigned Int | A variable that is an unsigned integer, it is used to check against KILL to set local\_b09 to 1, it is used later to check against DWLD to set variable local\_b14 equal to variable local\_604 |

|  |  |  |
| --- | --- | --- |
| **Command: DWLD** | | |
| **Offset** | **Data** | **Description** |
| 0x004015BE | 0x444C5744  DWLD in little endian | DWLD means, download (as best I can tell) |
| 0x004015BE | uint local\_608  unsigned int | In this line it is checked against DWLD and if it is it sets variable local\_b14 to be equal to local\_604 |
| 0x0040166C | DWORD local\_b14  Long unsigned int | Assigning this variable to be equal to that of local\_604 |
| 0x0040166C | DWORD local\_604  Long unsigned int | Being used to assign to variable local\_b14 |

|  |  |  |
| --- | --- | --- |
| **Command: EXEC** | | |
| **Offset** | **Data** | **Description** |
| 0x004015CE | 0x43455845  EXEC in little endian | EXEC is a command that spawns another process |
| 0x004015CE | uint local\_608  Unsigned int | Compared against EXEC |
| 0x004015FA | size\_t local\_b10  Unsigned int | This variable is set to be equal to the value of the variable local\_604 |
| 0x004015FA | DWORD local\_604  Int | This variable is used to set its value into the variable in local\_b10 |

References

Ghidra. (2019). National Security Agency. [Computer software].Retrieved from <https://ghidra-sre.org>