

Lab 3: Basic PE Static Analysis

ITSC 303: Malware Analysis

Table of Contents

[Lab Outcomes 4](#_Toc31272650)

[Background Reading 4](#_Toc31272651)

[Introduction 5](#_Toc31272652)

[1.0 Examining a New Sample 6](#_Toc31272653)

[2.0 Getting Started with Ghidra 8](#_Toc31272654)

[2.1 Opening the Sample in Ghidra 8](#_Toc31272655)

[2.2 Closing the Sample 16](#_Toc31272656)

[2.3 Using the Main Function 17](#_Toc31272657)

[2.4 Marking Up a Function 19](#_Toc31272658)

[2.5 Continuing to Reverse the Main Function 22](#_Toc31272659)

[2.6 Reverse Engineering FUN\_401160 27](#_Toc31272660)

[2.7 Reverse Engineering FUN\_401000 29](#_Toc31272661)

[2.8 Reverse Engineering FUN\_401190 33](#_Toc31272662)

[2.9 Reverse Engineering FUN\_4012C0 38](#_Toc31272663)

[2.10 Reverse Engineering FUN\_401440 44](#_Toc31272664)

[References 46](#_Toc31272665)

Malware Analysis

Lab 3: Basic PE Static Analysis

Lab Outcomes

This lab will focus on the following outcomes:

* Use skills from the previous lab.
* Understand the basics of opening and navigating Windows PE analysis software Ghidra
* Learn how to add comments and rename entities Ghidra
* Understand how Win32 API functions appear in Ghidra disassembly and decompilation
* Import and use Win32 API structures and enumerations Ghidra
* Reverse engineer functions in Ghidra disassembly views
* Identify and reverse engineer obfuscation techniques used by malware authors.
* Use Ghidra to aid in reverse engineering.
* Understand the features and functionality of an entire malware sample using reverse engineering.

Background Reading

* *Practical Malware Analysis* by Michael Sikorski and Andrew Honig
  + Chapter 1: Basic Static Techniques
  + Chapter 4: A Crash Course in x86 Disassembly
  + Chapter 6: Recognizing C Code Constructs in Assembly

Introduction

Malware is often developed in C/C++ and distributed as Windows Portable Executable (PE) files. The job of a malware analyst is to determine the exact functionality of malware. During the course of incident response situations, disassembly and decompilation may not be an option, as it is time consuming and in high-pressure situations like a security breach, it can be easy to lose sight of important functionality. Where this effort becomes most useful is during root cause analysis efforts, where a more in-depth analysis is conducted, and any findings are included as part of the major incident report. If analyzing malware for fun or in a less time-constrained scenario, consider taking some time to disassemble the sample to fully understand.

Once the functionality is known, effective remediation efforts can be made and the extent of the damage can be understood.

Reverse engineering a PE file requires disassembling the machine code in the file. The freely available Ghidra, greatly aids in the efficient and accurate reverse engineering of PE files. In addition to disassembling the machine code to assembly code, Ghidra can include metadata about the Win32 API (structure and enumeration definitions) that aids in understanding Win32 API uses, and even decompile assembly back to high-level source code for your ease of analysis. If reviewing the source code is easier than the assembly for you, feel free to use that functionality for the remainder of this lab. Ghidra also allows for extensive annotation and markup to help the reverse engineer decipher malware’s functionality.

This lab involves reverse engineering the entire functionality of a malware sample using Ghidra.

1. Examining a New Sample

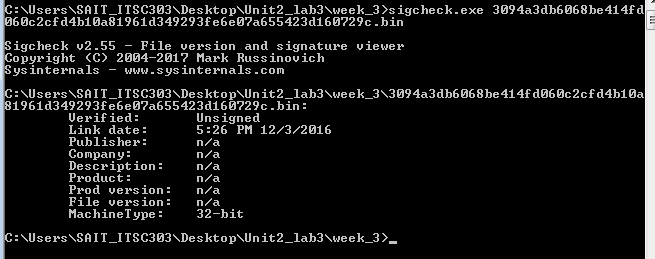
You will examine the file called **WinDiagService.exe,** located in the Unit 2, Week 3 folder of your samples section. For the purposes of this lab, assume that the file was collected as part of an incident response investigation of a suspected malware infection. The infected computer was seen communicating with the URL hxxp://aruddheksl.ru/news.php.

Your task is to verify that the file is indeed malware, to determine if it is possible that it generated the traffic communicating with hxxp://aruddheksl.ru, and to determine its capabilities. You will use Hiew, strings.exe and sigcheck.exe to get a sense of what the binary is capable of.

1. First, using **sha256sum.exe**, calculate the SHA-256 sum of the file, and then rename the file to **<sha-256>.bin**. Among other benefits, this ensures you don’t accidently run the binary.

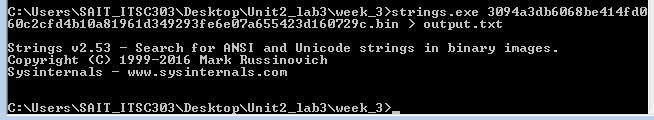


1. Using **sigcheck.exe**, check if the file is signed. Is it signed?



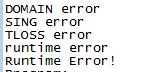
**Answer:** Yes / No (Highlight one)

1. Using **strings.exe**, dump the strings to a text file, examine them, and then answer the following questions. (Refresher: strings.exe *full path to WinDiagService.exe* > *full path to store output.txt file*)



**Questions**

1. List at least three interesting strings. Don’t include names of Win32 API functions. Briefly describe what each string could be used for or what it tells you about the sample. (Consider strings that are useful for identification purposes within a malware incident)
2. String 1:



**There seems to be a lot of errors happening near the beginning of the file, which is very strange behavior.**

1. String 2:



**This file references the days of the week, months of the year and time A LOT, which is very strange for a file like this.**

1. String 3:



**This is a very suspicious URL that it is communicating with**

1. Open the sample in Hiew or another file analysis tool (Filealyzer, HexView, etc). Based on the strings you dumped before and on how the code appears in Hiew, is the sample packed?

**I used PEveiw and no, it is not packed as you can read the strings**

Yes / No (Highlight one)

1. Answer the following questions based on the imports of the function.
   1. Does the sample import any APIs related to reading or writing files on disk? If so list the API functions

**WriteFile**

* 1. Does the sample import any APIs related to reading or writing from the registry? If so list the API functions
  2. Does the sample import any APIs related to communicating over the Internet? If so list the API functions

**InternetOpenA**

**InternetOpenUrlA**

**InternetReadFile**

**InternetCloseHandle**

* 1. Does the sample import any APIs related to cryptography? If so list the API functions

**I could not find any**

* 1. What is the virtual address of the entry point of the program? (You can manually do the addition or use Hiew.)

**0x401FC0**

* 1. Based on your analysis of the sample’s strings, signing information, and imported APIs, give a brief summary of what you know about the sample at this point.

**As far I can tell, it seems to copy information from the host and it also communicates with a suspicious URL.**

1. Getting Started with Ghidra

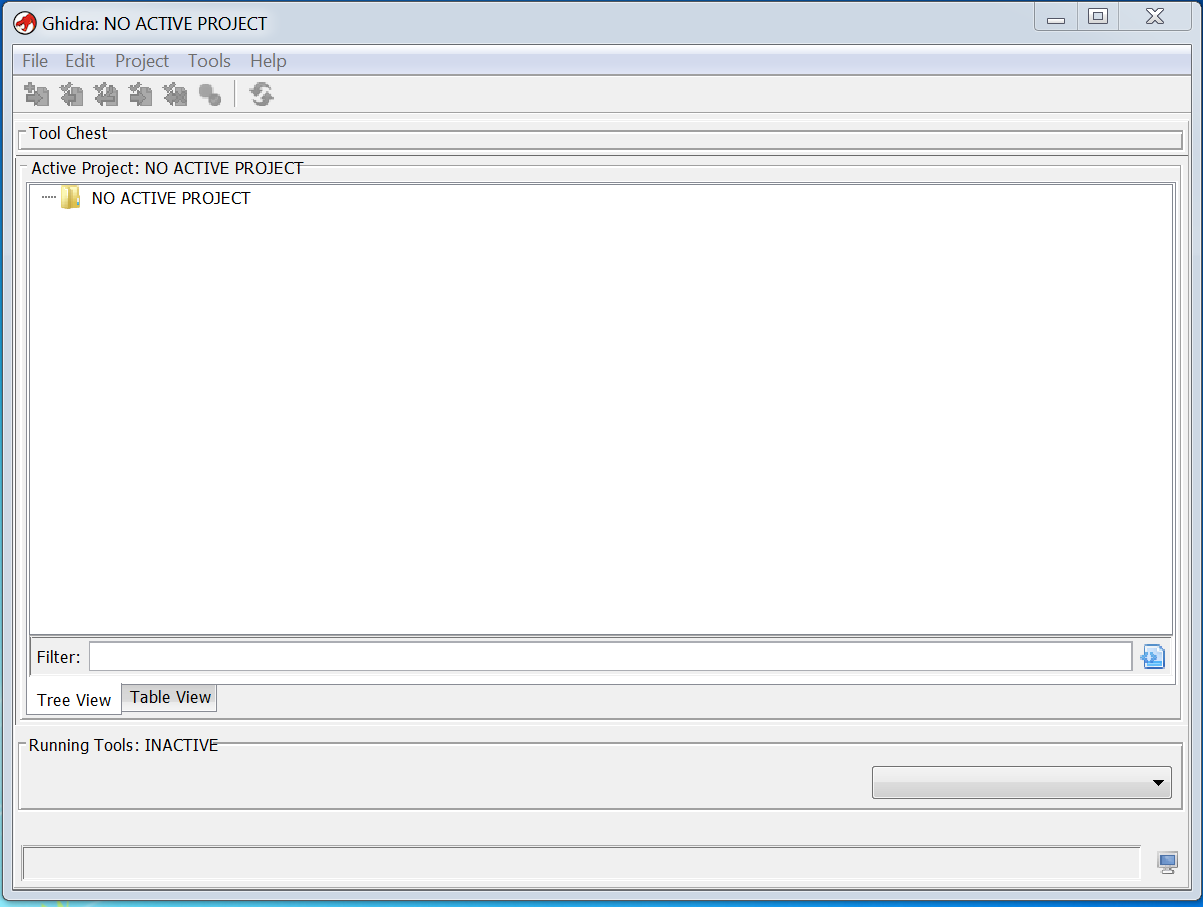
In this section, you will use Ghidra to further examine the file. Ghidra disassembles the file and provides a graph-based view of the functions to reverse engineer its capabilities precisely.

## Opening the Sample in Ghidra

* 1. Launch Ghidra using the ghidraRun.bat script located on your VM in your toolset folder under RE\disassembler\ghidra\_9.1-BETA\_DEV

This may take some time, as it’s a Windows batch script that is opening the tool under a Java environment.

* 1. A project window will open and should display that No Active Project is in progress.



* 1. Click on the NO ACTIVE PROJECT folder and the File button will be clickable. Click File, and New Project.

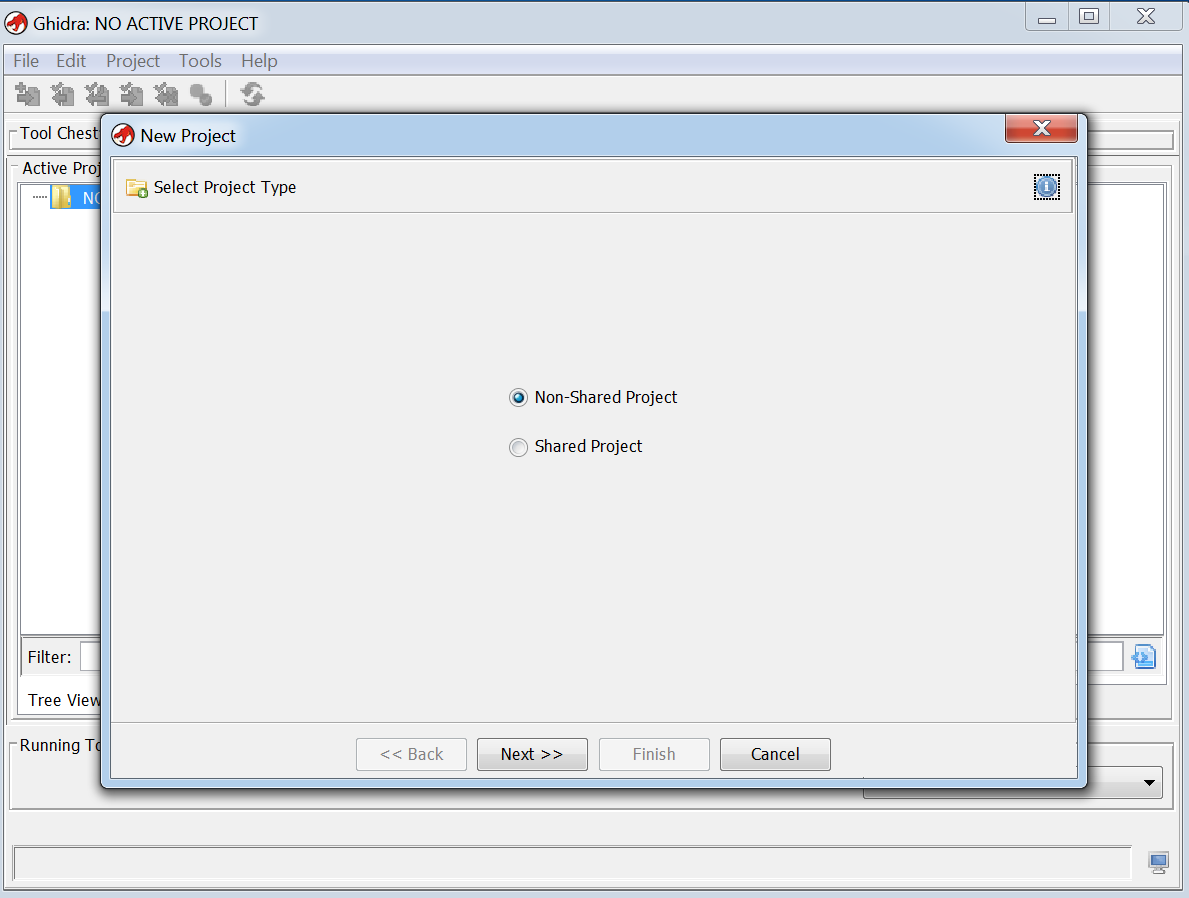


Figure 1: Ghidra’s start a new project window

Source: Ghidra, 2020. 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. Ensure that Non-Shared Project is selected, and click **Next>>**
  2. Select a file location, and a unique name for your project according to figure 2, and click **Finish**

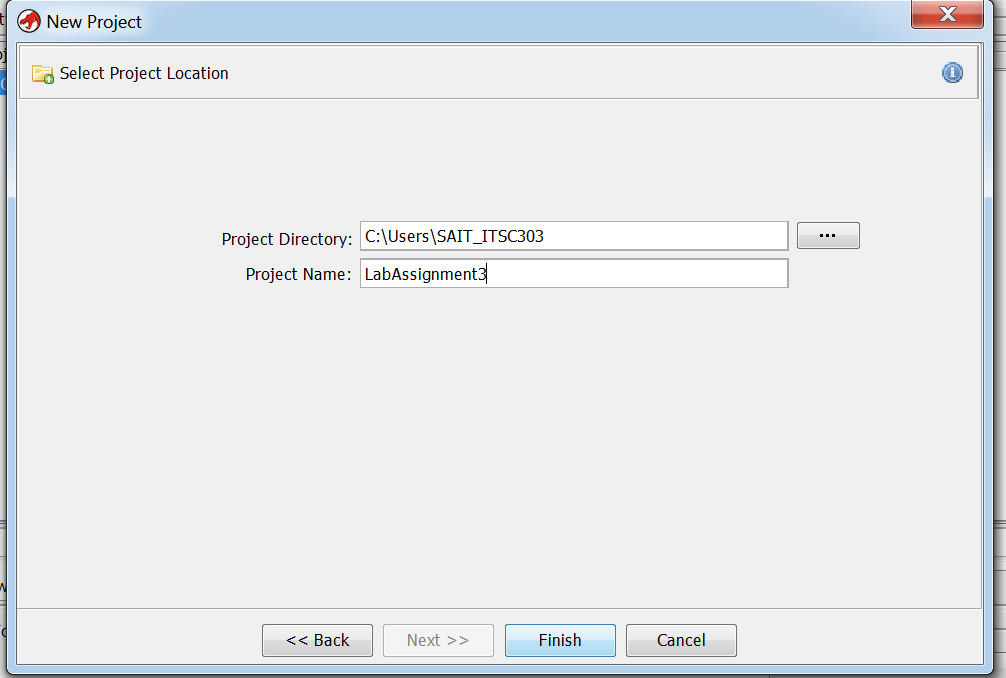


Figure 2: Ghidra New Project Window

Source: Ghidra, 2020. 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. From here, you will have a new project initiated and ready for analysis. Click on the named Project in your screen, and press the green “dragon” icon. This will open up the Ghidra interface. *This may take a few minutes, just let it run*

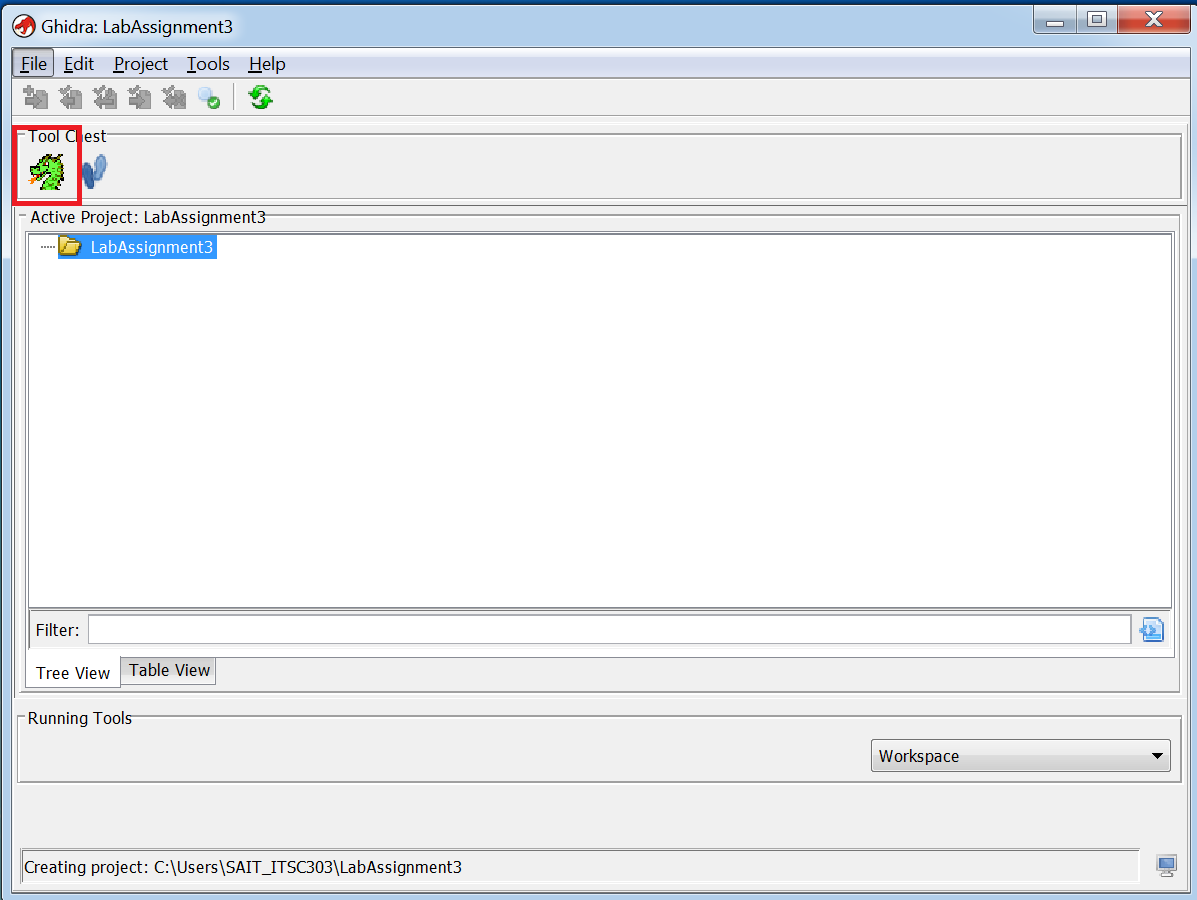
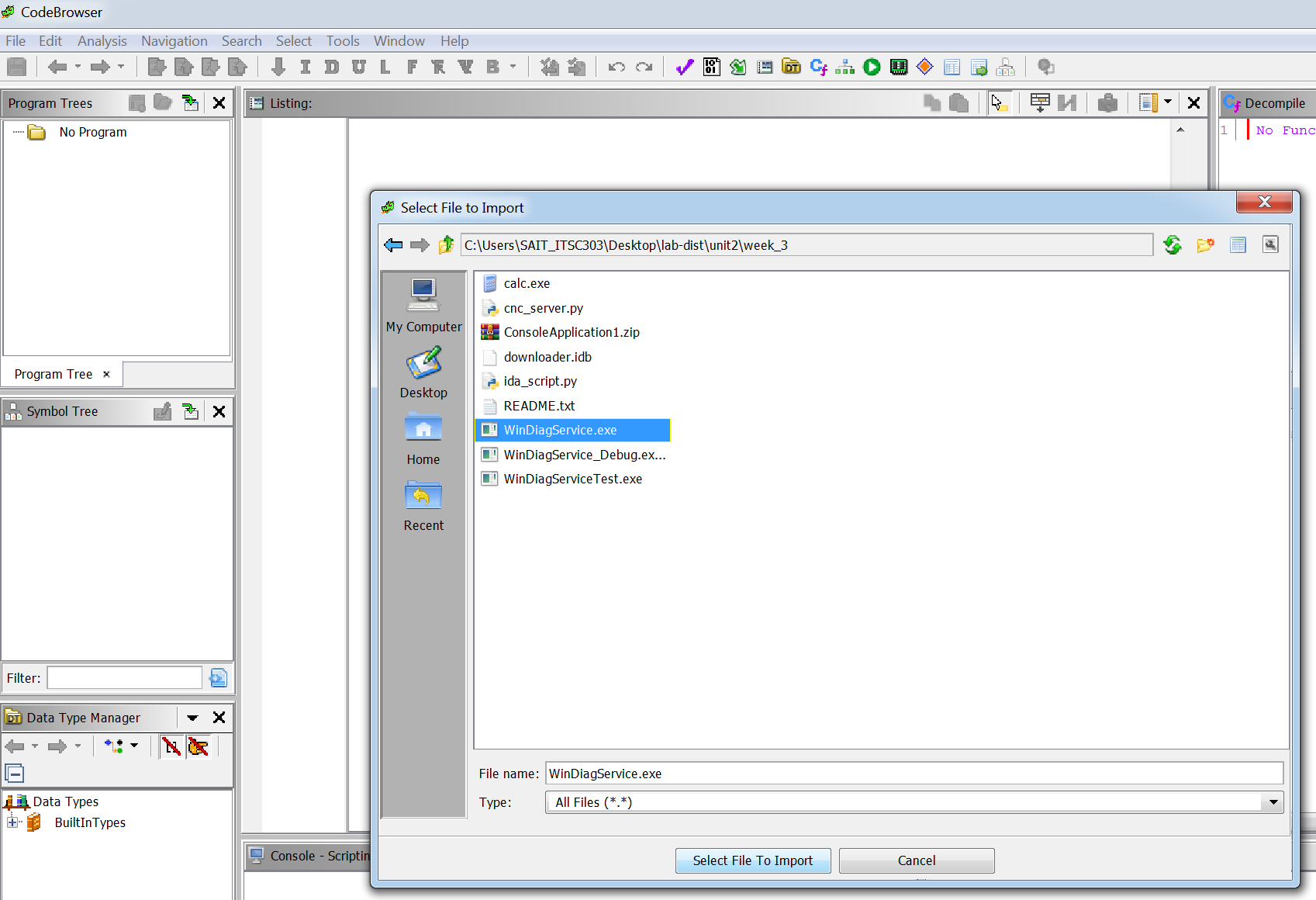


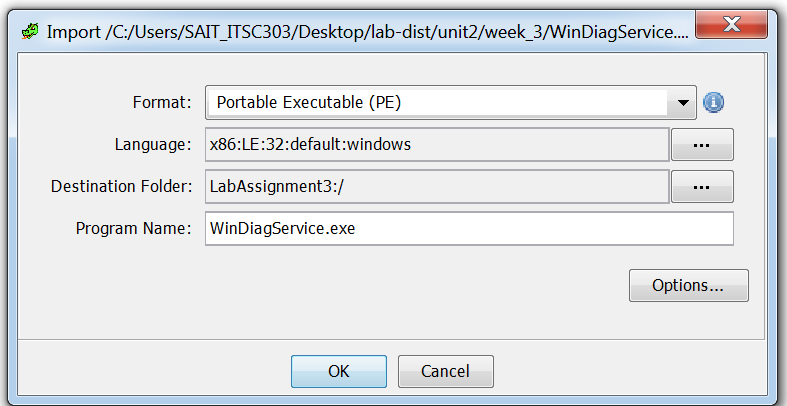
Figure 3: Ghidra Interface Showing New Project Created

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.

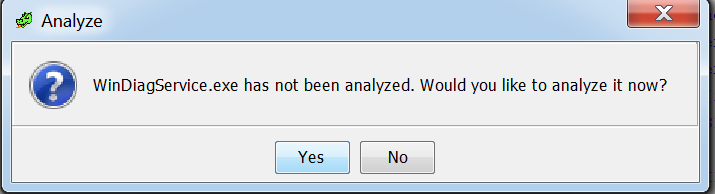
* 1. The Ghidra interface will open up, and once loaded, click File in the upper left corner, then select Import. This will pop open a directory menu which will allow you to navigate to your sample. Navigate to WinDiagService.exe on the Unit 2 Week 3 folder, and press Select File to Import



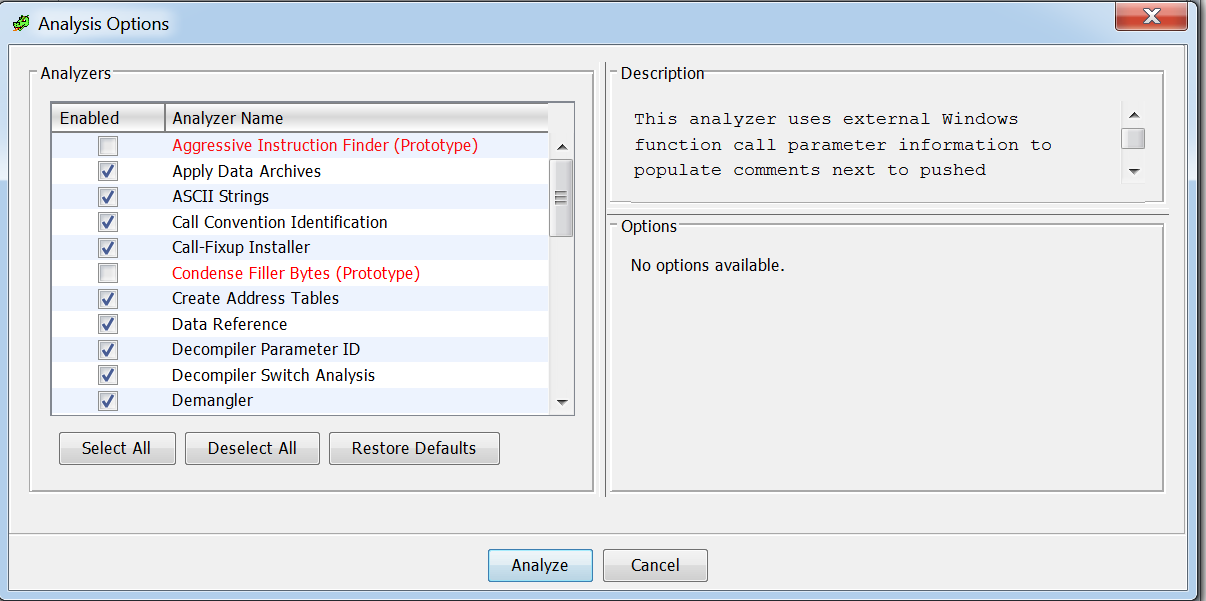
* 1. The import sample screen will pop up, confirming your options. Press OK and the popup will close. This will also take some time to complete, so let it run until finished



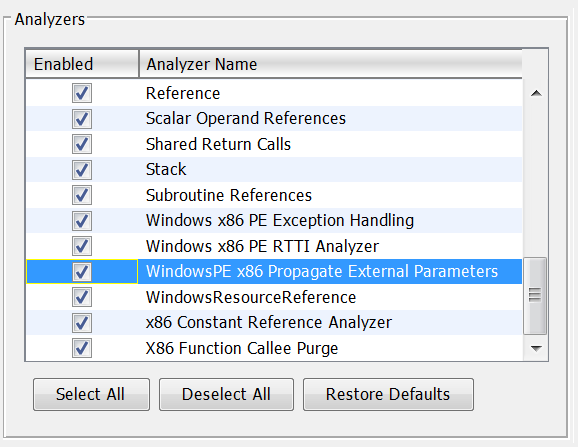
* 1. Once the sample is loaded, you’ll be prompted by Ghidra asking if you would like to analyze the sample now. Select Yes.



* 1. A new window will pop up and ask for you to select some options for analysis. Ensure the options with red text are left unchecked in accordance with the below figure:

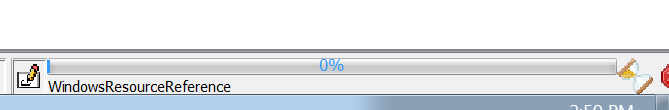


Scroll down until you see the WindowsPE x86 Propagate External Parameters, and ensure it is checked. This will allow you to extend the Windows libraries into your code analysis, which can provide some valuable context into what the function you’re analyzing is actually doing.



Once you’ve selected the necessary boxes, click Analyze, and give it some time to process entirely. You may receive some errors that state a file was missing or unable to be located (possible with PDB files), click OK and carry on

* 1. Now the sample is loaded into Ghidra and you should be able to see a lot of interesting elements necessary for analysis efforts. Take some time to find each of the following windows in the Ghidra interface:
     + Program Tree: Exhibits program headers used by the binary
     + Symbol Tree: Shows all imports, exports, functions, classes, and other useful program logic
     + Listing Window: Shows assembly code for the program
     + Decompiler: Provides decompiled source code.
  2. Click anywhere inside the Listing Window, and press CTRL + A to highlight all of the assembly code
  3. Inside the Listing Window right-click and select the Disassemble option. This will go through and disassemble all of the code for you. The bottom right hand corner of your Ghidra view will have the status of the disassembly. Ensure this is completed before conducting further analysis. See the figures below:



**This means the disassembly is still in progress, do not conduct analysis at this point**

****

**This indicates the disassembly is complete and your analysis can continue**

* 1. Now when you click on instructions within the Listing Window, the Decompiler window should begin to present the relevant source code for your analysis efforts as well. Take a minute to move about the interface and see how this works. Clicking on the function header will often present you with the entire source code for your review.

## Closing the Sample

If you opt to close Ghidra and power down your machine, simply save your work either with the blue save icon in the upper left corner, or my selecting File-> Save. If you want to make additional copies of your work in Ghidra, simply select Save As. If you attempt to close Ghidra without saving, you should be prompted to save your work, however do not rely on this entirely, as it may not necessarily prompt you every time.

## Using the Main Function

Even if the main function is open, knowing how to jump to specific addresses in Ghidra is useful.

1. To jump to an address, press the **G** key
2. Type **0x4017D0** in the *Jump address* box and click **OK**.

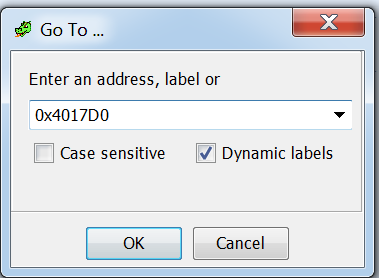


Figure 9: Jump Address Box

Source: Ghidra 2020. 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.

The information in Figure 10 appears.

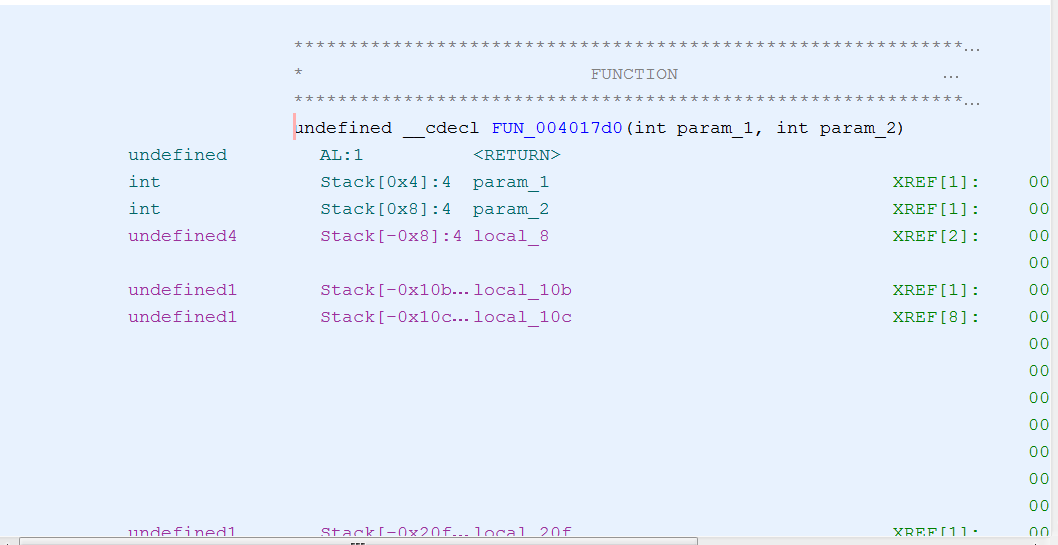


Figure 10: Main Function

Source: Ghidra, 2020. 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. On the right hand side in the Decompiler window, you should see some source code derived from the assembly.

**Note:** The function is named **main** and the arguments are named as you would expect for the main function of a C program (e.g., argc, argv\*\*, envp\*\*).

1. To verify the address is correct, click on the function header in your Listing Window, and observe the lower section under the Console window

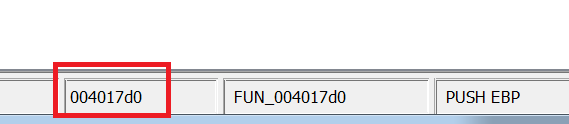


Figure 11: Current Address

Source: Ghidra, 2020. 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.

**Note:** Remember that 0x4017D0 is not the address of the entry point you calculated using Hiew above.

1. Jump to the address of the entry point you calculated above.

A function call is displayed: **\_\_\_security\_init\_cookie**

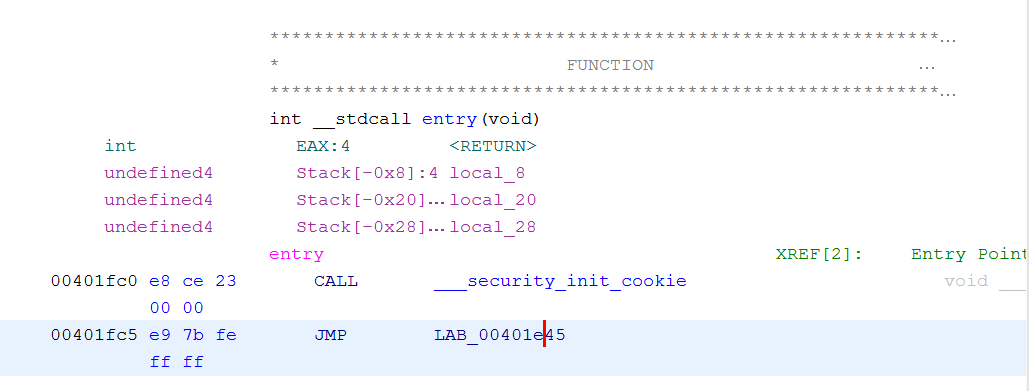


Figure 12: Entry Point

Source: Ghidra, 2020. 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.

The code before the main function is created by the compiler. The code the compiler generates sets up the execution environment and initializes certain security features, in this case, stack protection. Typically, you don’t need to worry about this compiler-generated code. However, the programmer can control certain aspects of this generated code, and it is possible to place logic before the main function. For example, malware has been known to hide logic before main, but this is not the case for the current sample.

In this this sample, Ghidra was able to find and name the main function because it recognized the compiler-generated code and knows where the main function will be called in this code. However, Ghidra may not always be able to determine where the main function is located, and you must do so manually.

In cases where Ghidra can’t find the main function, remember that the entry point is likely not the main function and that there may be programmer-controlled logic before the main function.

## Marking Up a Function

Before you start reverse engineering the main function, jump back to the address of main, **0x4017D0**.

Ghidra allows you to add comments to any line. Looking at the beginning of the main function, you see the following instructions:

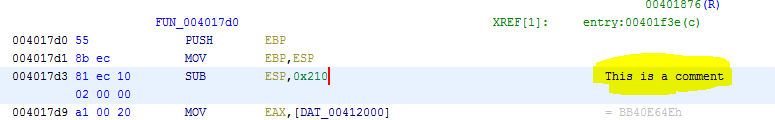
push ebp

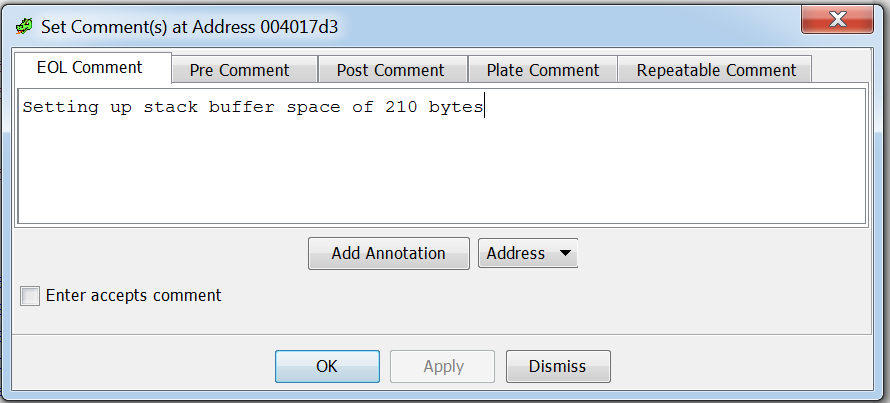
mov ebp, esp

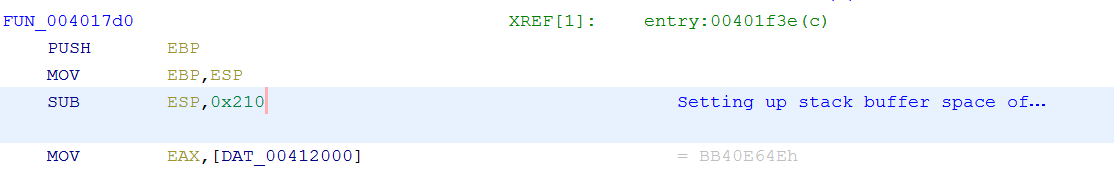
sub esp, 210

These are typical stack frame initialization functions. In this case, 0x210 bytes of space are being reserved on the stack for local function variables.

1. Add a comment to the main function describing these three lines by placing your cursor on the line to which you want to add a comment, right-clicking, mousing over the Comment menu, and selecting EOL comment. This will spawn a new window, where you can type in your comment. When you’re finished, press the OK button, and a new comment will appear at the end of the line of assembly.







**Figure 13: Main Function with Comment**

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.

**Note:** To change or delete comments, select the comment with your cursor, right click and select the Comments menu, and you will be able to see the Delete Comment option.

Looking at the next few instructions you see:

mov eax, \_\_\_security\_cookie

xor eax, ebp

mov [ebp+var\_4], eax

These instructions create a stack canary, or stack cookie, to help mitigate against buffer overflow attacks. The compiler automatically adds these instructions at the beginning of most functions.

The cookie value is calculated into eax and then placed on the stack at [ebp+local\_8]. Ghidra automatically generates the name **local\_8** (your name may be slightly different). Any time this stack cookie is used, it will be referred to as local\_8.

1. To give local\_8 a more meaningful name, place your cursor on local\_8 and press **L** or right click and select **Edit Label**.
2. Give the variable the new name **stack\_cookie**.

The main function now looks like what’s shown in Figure 14.

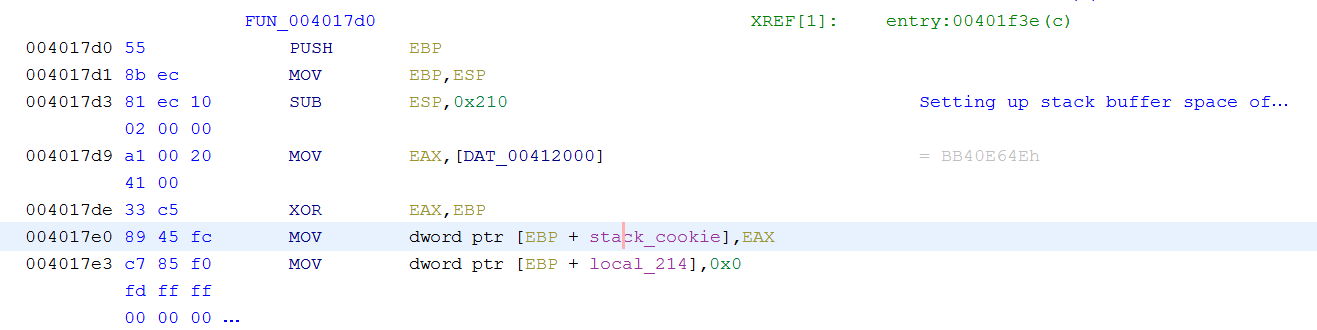


Figure 14: Main Function with Renamed Variable

Source: Ghidra, 2020. 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. To see other locations where the stack\_cookie variable is used, place your cursor on the **stack\_cookie** variable and press the **CTRL+Shift+F**

Ghidra displays cross references, as shown in Figure 15.

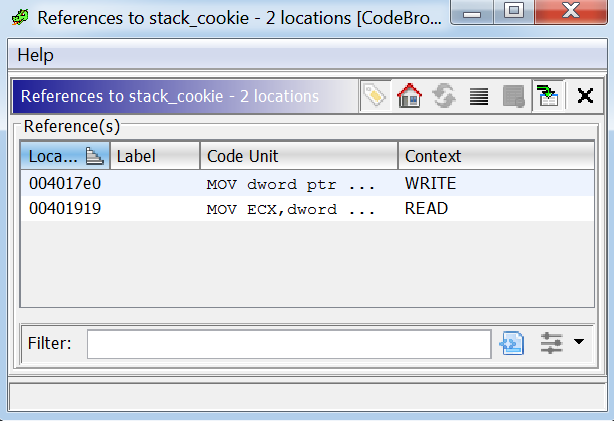


Figure 15: Cross References to stack\_cookie

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.

Figure 15 shows that there are two uses of the variable stack\_cookie: one 0x10 bytes from the beginning of the main function (the location you looked at) and a second reference 0x149 bytes after the beginning of the main function.

1. To jump to the second reference to stack\_cookie, select it from the list and click **OK**.

You can see where the stack\_cookie is checked at the end of the function. The stack cookie is copied from the stack to the register ecx, and the function called FUN\_00401927 is called.

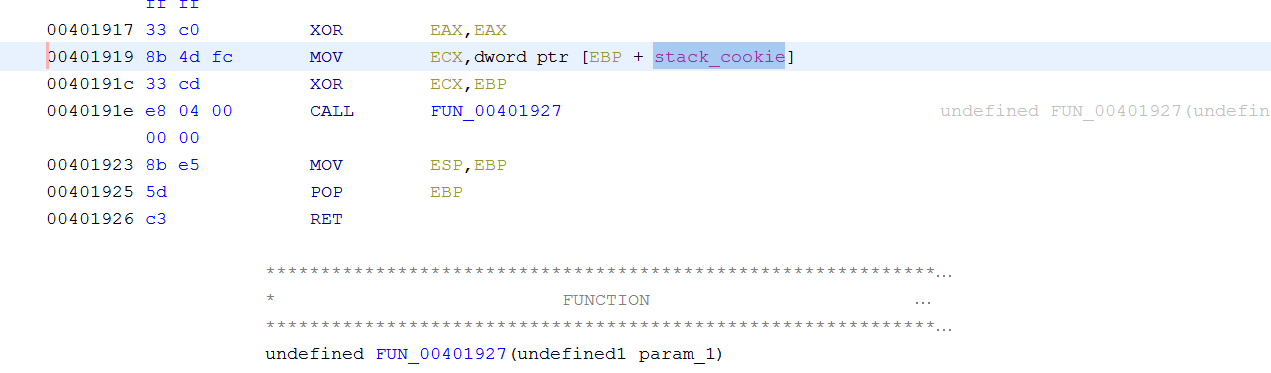


Figure 16: The stack cookie is used

Source: Ghidra, 2020. 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.

You will constantly use the three actions covered in this section when looking at malware in Ghidra:

* + Adding comments
  + Renaming things (e.g., variables, functions, offsets)
  + Jumping to cross references of variables

Practise adding lots of comments and renaming everything you can to make the assembly more understandable.

**NOTE: As stated, we are unable to complete past 2.4 as we do not have access to the right tools.**

## Continuing to Reverse the Main Function

Although it isn’t necessary to add a comment to every line, it’s a good idea to add a comment to explain what is going on in the function. Rename variables as well (if you’re not sure what they’re being used for, give them temporary names).

1. Examine the main function up to the first jump instruction.

jl lab\_401857

1. Add comments and rename variables, as required. The function before markup should look like Figure 17.

**Note:** Don’t scroll past this screen until you have finished marking up your function.

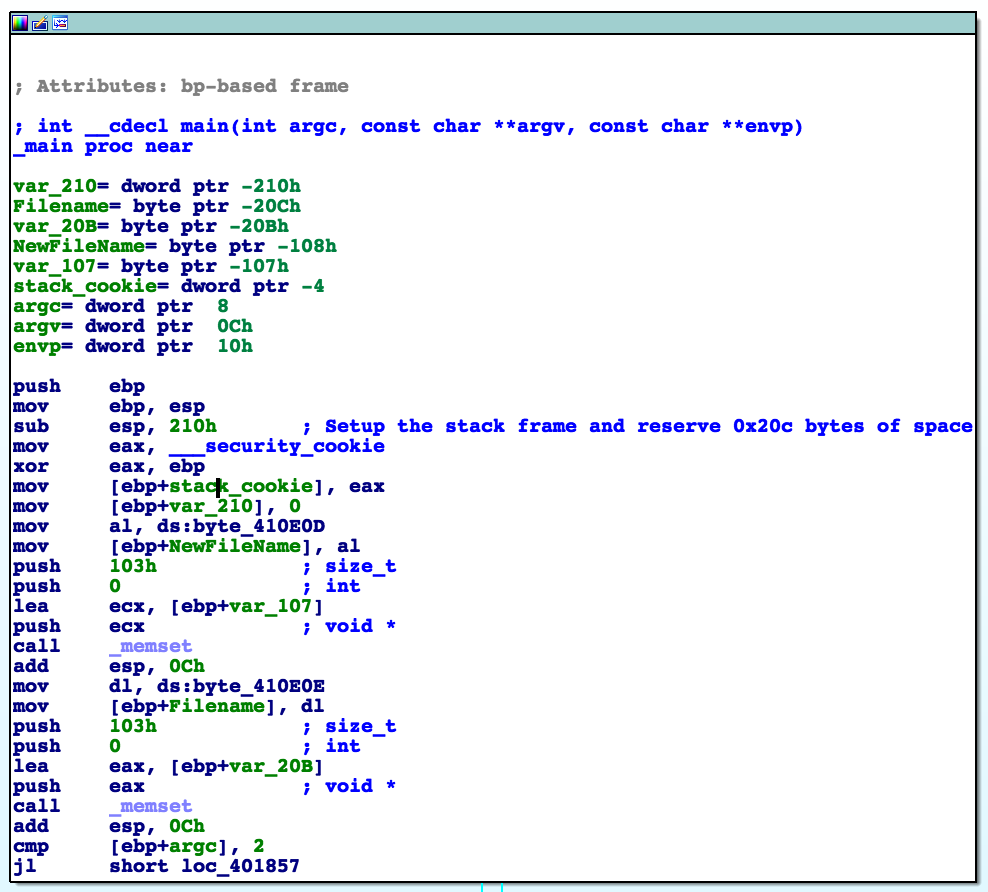


Figure 17: Main Function Before Markup

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. Compare your marked up function to the function in Figure 18.

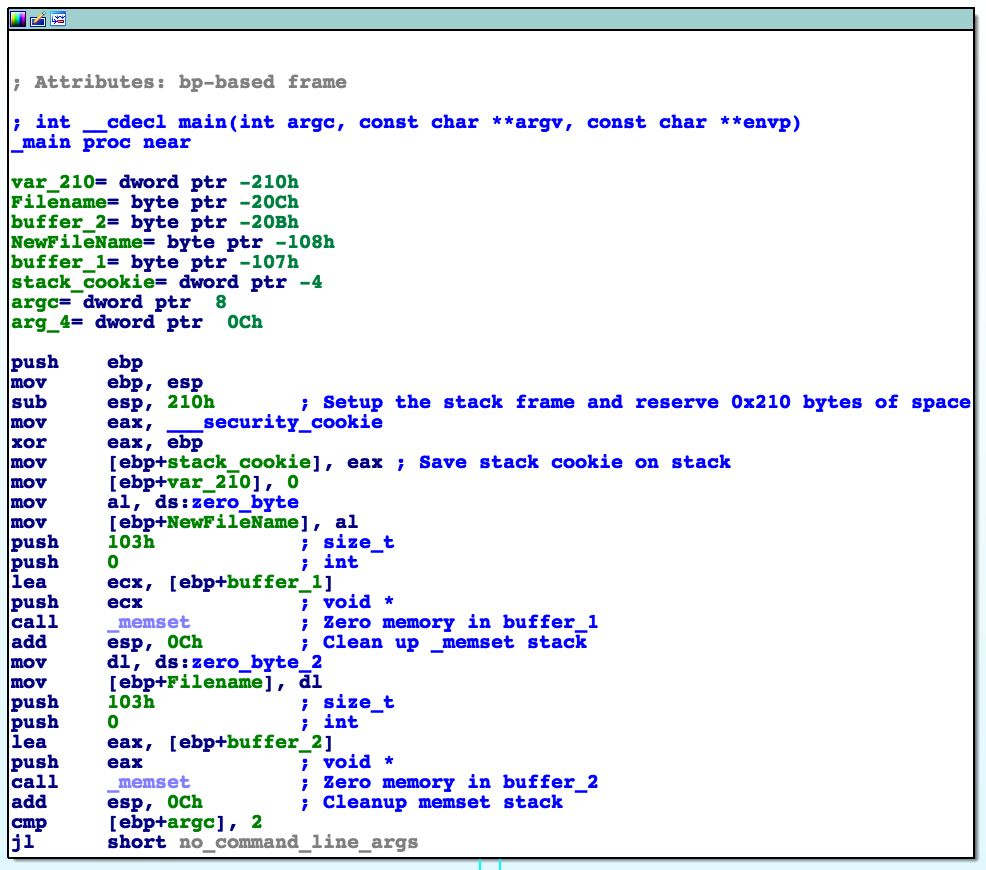


Figure 18: Main Function After Markup

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.

**Notes:**

* The variables buffer\_1 and buffer\_2 appear to be some kind of container, because the memory for each was zeroed out. They may later hold strings or some other data. You can rename them when you know more about them.
* The value of argc is checked. This value controls whether the jump is taken. If the value of argc is lower than 2, the jump is taken.
* The value of argc is the count of command line arguments passed to the program. The program name is always the first argument, so the value of argc should always be at least 1. If the value of argc is 2 or more, then the user has passed at least one command line argument.
* Jump locations can also be renamed. In this case it was renamed to no\_command\_line\_args.

You have established that this program can accept one or more command line arguments. Now examine how the program uses its command line arguments by looking at the code that is executed if the jump is not taken.



Figure 19: Code When the Jump Isn’t Taken

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.

The strncpy function is defined as follows:

char \*strncpy(char \*strDest, const char \*strSource, size\_t count);

Remember that arguments are pushed in reverse order to the argument mapping, as follows:

* + size\_t -> 0xff
  + char \*strSource -> dword ptr [esi+4]
  + char \*strDest -> offset s\_HttpAruddheksl

The value offset s\_HttpAruddheksl means data at an offset in the current file. Double-click **s\_HttpAruddheksl** to jump to where it is defined.

Notice that s\_HttpAruddheksl is defined in the .data section. It is a null terminated string (the URL) that the sample was observed communicating with. The name “s\_HttpAruddheksl” was generated by Ghidra from the data in the string.

1. Rename this variable (for now, name it **url**). Follow the same procedure as you used for renaming variables.

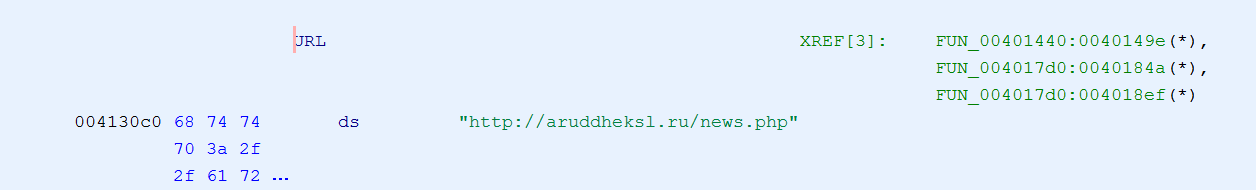


Figure 21: Variable Renamed to url

Source: Ghidra, 2020. 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. To return to your previous location (the main function), press the **Back** button in the top menu.

The strncpy function is used to copy the value of the first command line argument after the program name to the variable you called url. This allows the program to be run and the default URL to be overwritten by the command line argument.

1. Rename the FUN\_00401BE0 function to \_strncpy

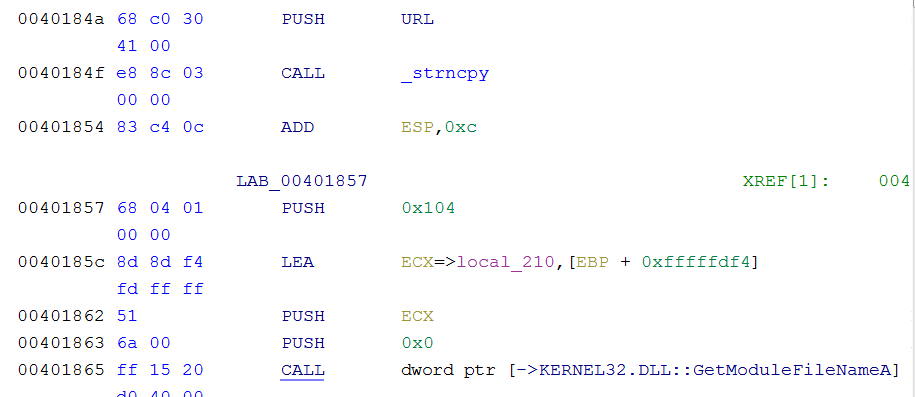


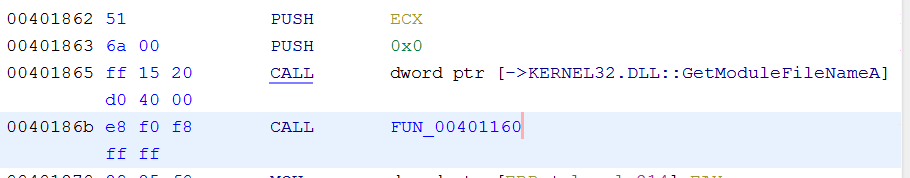
Figure 22: Code Section Including GetModuleFileNameA

Source: Ghidra, 2020 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.

The call to GetModuleFileNameA is an example of how Ghidra displays the call of an imported function. Ghidra knows most Windows API functions, and when one is called the argument parameters are noted as comments beside where they are pushed on the stack.

Also, notice that Ghidra automatically named the stack variable [ebp+Filename], because this stack variable was used as the parameter that receives the result of GetModuleFileNameA.

After the call to GetModuleFileNameA there’s a call to the unnamed function sub\_401160. The return parameter of sub\_401160 is checked and, depending on the result, the program will be exited.



## Reverse Engineering FUN\_401160

1. Double-click the function **FUN\_401160**.

The function appears as shown in Figure 23.

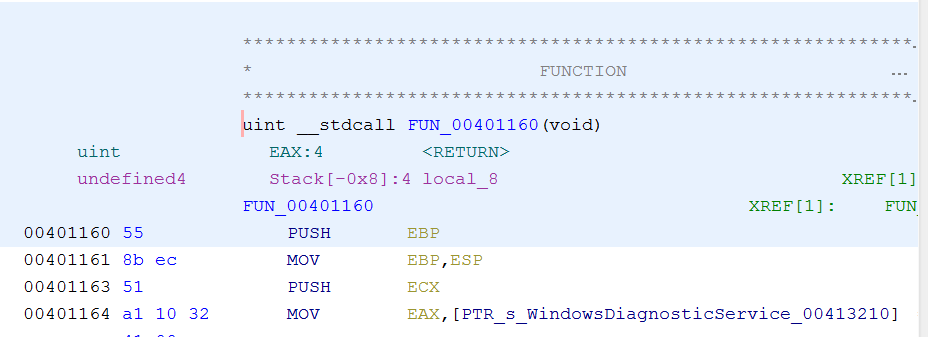


Figure 23: The Function sub\_401160

Source: Ghidra, 2020. 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. Immediately after the call to CreateMutexA, a call is made to GetLastError to get any error conditions as the result of the CreateMutexA call. If the result of GetLastError is 0, the jump is taken and 1 is returned in eax. If the result of GetLastError is non‑zero, implying that there was an error, 0 is returned.
2. What is this purpose of this function? (Ensure you identify what the return value means.)

**Answer:**

1. Based on your answer above, rename the function by placing your cursor on it and pressing the **N** key or by right-clicking the function name and selecting **Rename**.
2. Press the **Back** button to return to the previous function. Notice how the return value to the function is used.
3. Examine **sub\_401000**.

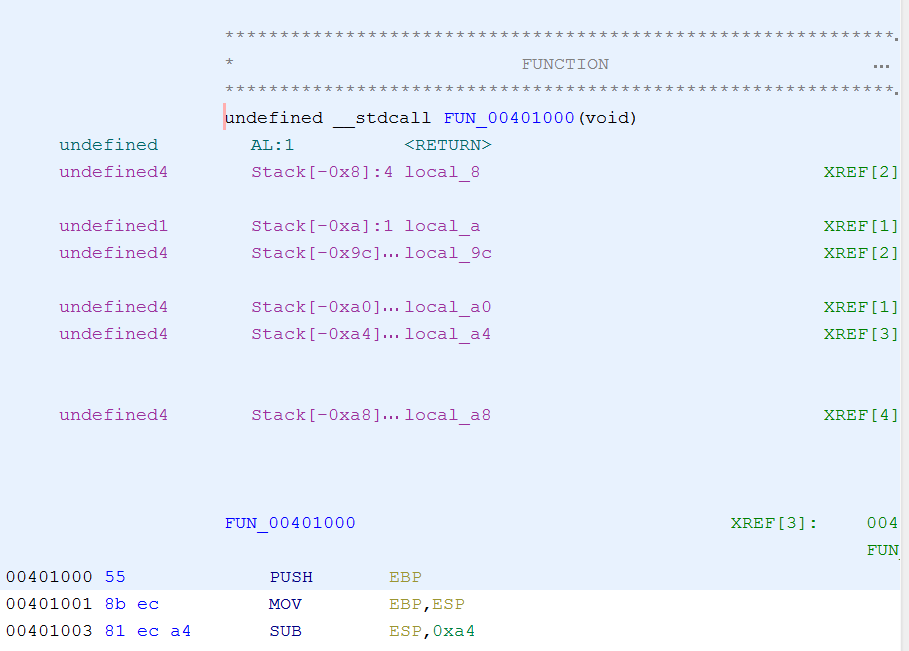


Figure 24: The Call to sub\_401000

Source: Ghidra, 2020. 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.

## Reverse Engineering FUN\_401000

1. Double-click the function **FUN\_401000**.

This function shows how Ghidra displays known Windows API data structures.



Figure 25: The Function sub\_401000

Source: Ghidra, 2020. 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.

The function GetVersionExA takes one argument: a pointer to an LPOSVERSIONINFOA structure.

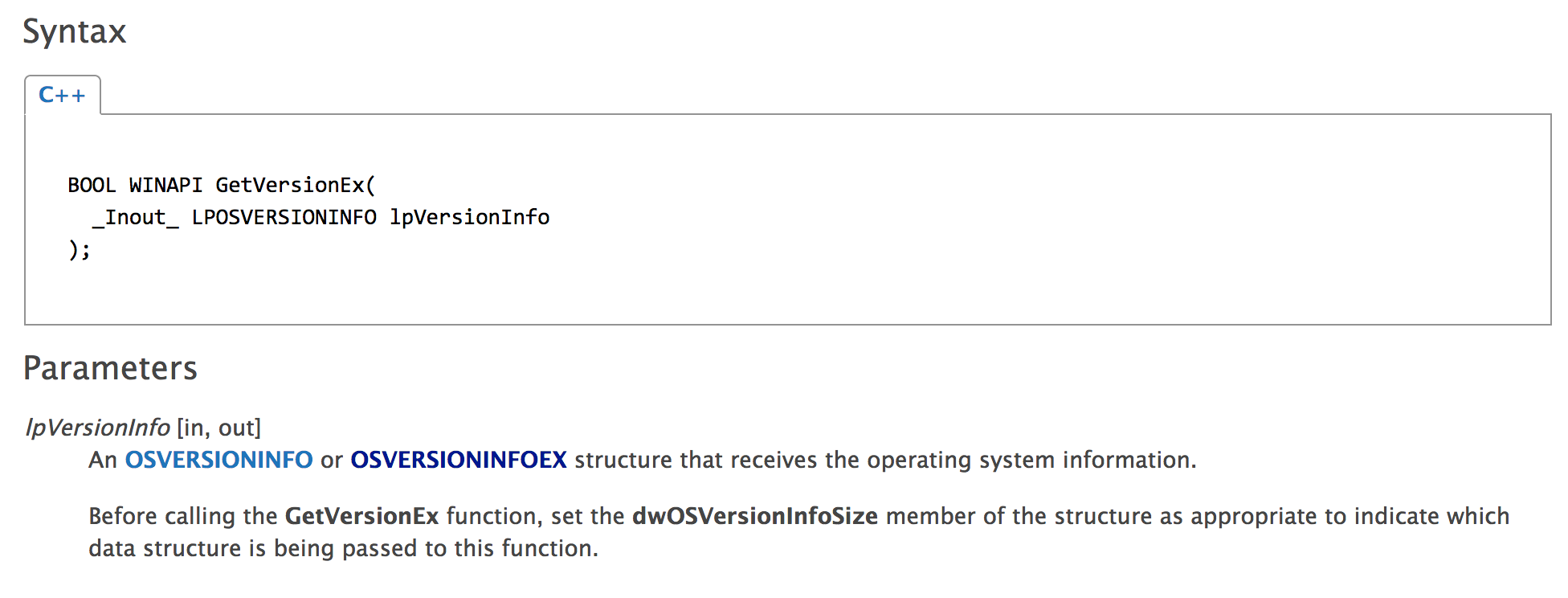


Figure 26: Function Signature for GetVersionExA

Screenshot, Microsoft.com, 2017.

1. The Microsoft Developer Network (MSDN) has a definition of OSVERSIONINFOA. Ghidra can present all of the available structure contents and some editing options that may be useful for your analysis. To view the structure contents in Ghidra, click on the structure, in the Decompiler view, and right click Edit Data Type to open the structure contents window. This is a helpful step if the decompiler or disassembly efforts do not understand and cannot process a data structure inside a malware sample.

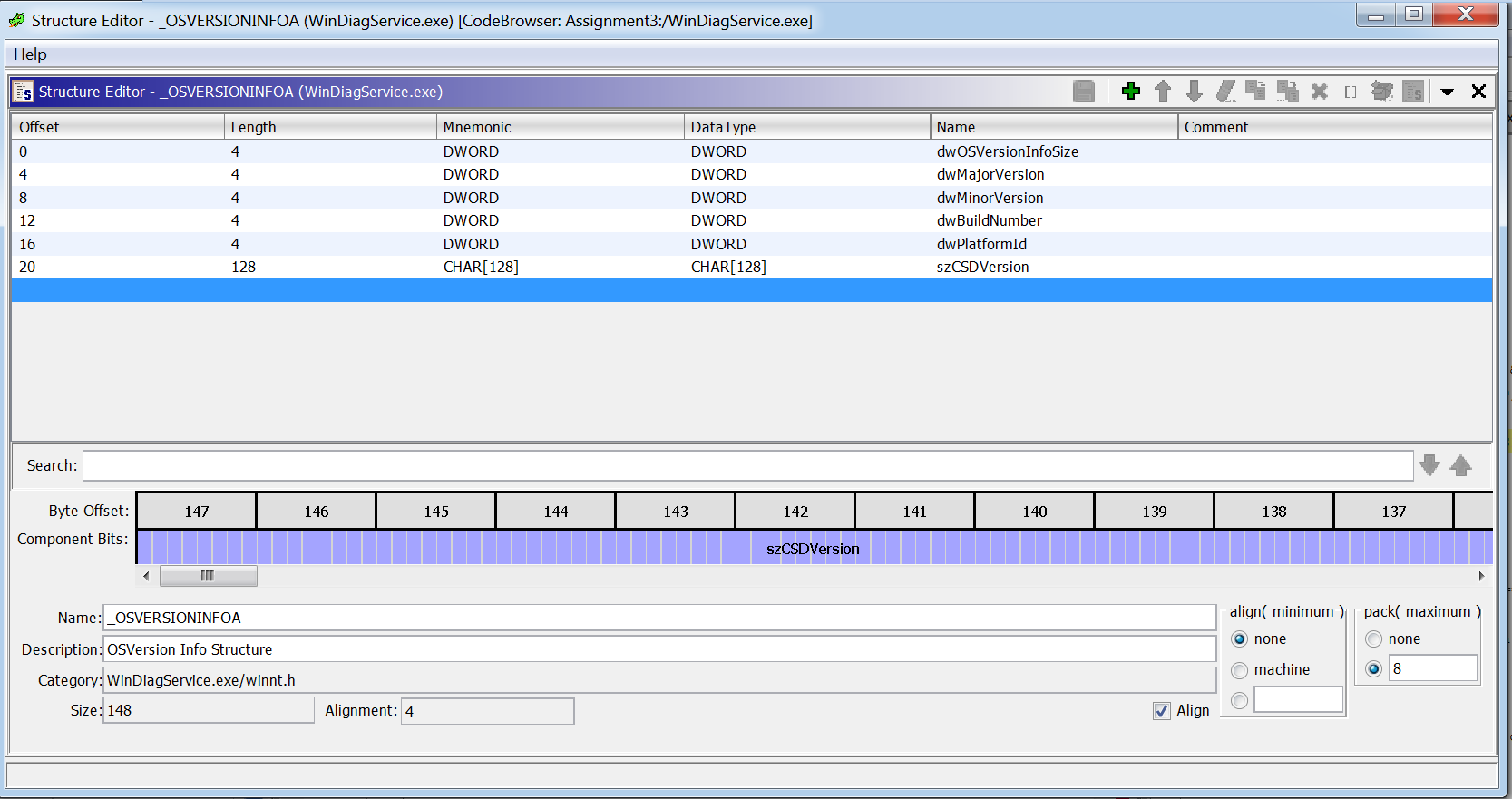


Figure 26: Structure of OSVERSIONINFOA

Source: Ghidra, 2020. 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. There is a problem here with what Ghidra has done. The function GetVersionEx can be passed either an OSVERSIONINFOA or an OSVERSIONINFOEXA structure; however, the size of the structure, specified in the dwOSVersionInfoSize attribute of the structure, determines which structure has been passed in. You can see from the function listing that the size of the structure is set to 0x9c. In the definition of the OSVESIONINFOA structure in the MSDN, its size is only 0x94 bytes. In this case, Ghidra should have chosen OSVERSINFOEXA rather than OSVERSIONINFOA.
2. Right click the \_OSVERSIONINFOA variable in your decompiler view, and select Retype Variable, or press CTRL+L to open the prompt
3. Ensure the text isn’t highlighted by clicking inside the text box. Begin pressing backspace in the text section and it will start to show other data types available. Once you see \_OSVERSIONINFOEXA, double click the structure and it will load into your window. Press OK

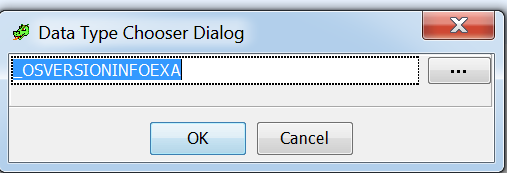


Figure 27: Changing Data Structure

Source: Ghidra, 2020. 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. Rename the local\_a4 (your name may be different) variable to be VersionInformation.

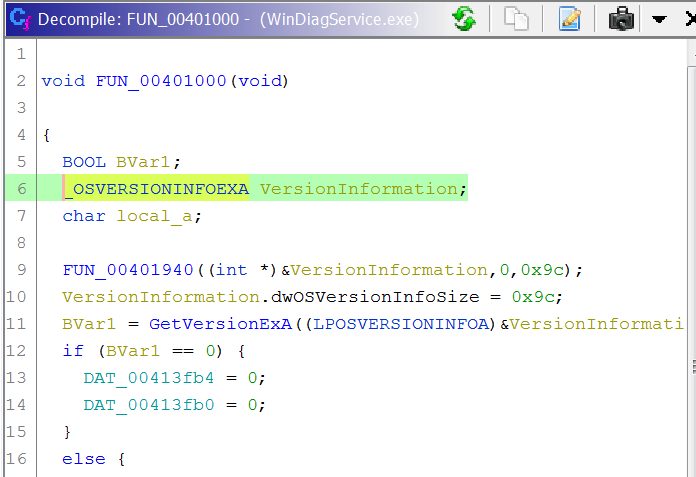


Figure 28: Renamed Variable to VersionInformation

Source: Ghidra, 2020. 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.

The function sub\_401000 can now be reverse engineered. Choose a good name for the function. In addition, the function sets the values dword\_413FB0 and dword\_413FB4, so both of those should be renamed and their possible values and the meaning of the values documented.

**Questions**

1. What is a good name for dword\_413FB0? What are its possible values?
2. What is a good name for dword\_413FB4? What are its possible values?
3. What is a good name for sub\_401000? What is the purpose of the function?

After you answer these questions, go back to the main function.

## Reverse Engineering FUN\_401190

The next function call is to FUN\_401190. The push instruction before it indicates that it takes one argument.

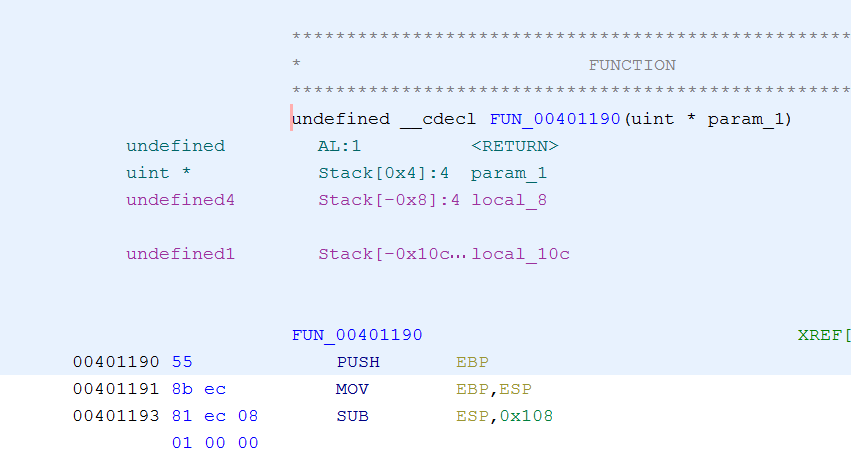


Figure 33: The Code Following FUN\_401000

Source: Ghidra, 2020. 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. Double-click the function **sub\_401190**.

This function takes an argument. When arguments are pushed onto the stack, Ghidra names the first argument with local place holders. Function arguments can be renamed as usual, and it’s a good idea to do so.

Notice that in the call to SHGetFolderPathA, Ghidra may have given it a local variable name, however it should be a constant called csidl. Many Windows API functions take constants as arguments. When the function is shown in an IDE, the symbol name of the constant is most often used (e.g., FILE\_SHARE\_READ or FILE\_SHARE\_WRITE when using CreateFile). When you look at the function in Ghidra, the symbolic name is lost but the constant value may remain.

Ghidra knows many of the Windows constants’ symbolic names. The constant generally needs to be looked up on MSDN. In this case, you need to look up the symbolic names for 0x1and 0x1c for SHGetFolderPathA. See [CSIDL](https://msdn.microsoft.com/en-us/library/windows/desktop/bb762494(v=vs.85).aspx) (https://msdn.microsoft.com/en-us/library/windows/desktop/bb762494(v=vs.85).aspx) for more information.

MSDN lists the constant symbolic names but not the values. You can find the exact values in Windows header files (.h files), but since they all start with CSIDL\_ you can look them up that way.

1. To look up constants, place your cursor on the constant value you want to look up (in this case, **0x1a** on the left).
2. Right click and select Equate, or press the E key
3. You will see a prompt to change the value, start typing CSIDL, and select CSIDL\_APPDATA, and ensure it’s set to persist through the entire program

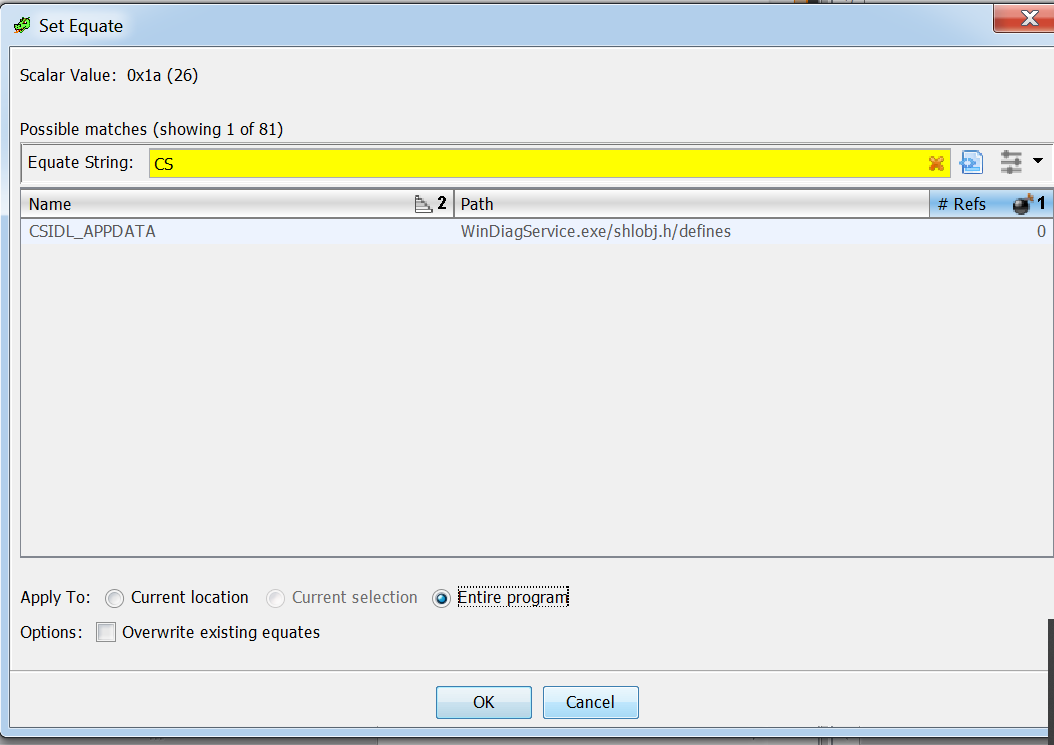


Figure 35: Equate Confirmation Screen

Source: Ghidra, 2020. 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. Add the second value, 0x1c, using the same procedure. You will likely need to research this constant in the MSDN

Using the constant values, finish reverse engineering the function and then answer the following questions.

**Questions**

1. What is the argument passed into fun\_401190? (Ensure you have given it a meaningful name.)
2. What is a good name for fun\_401190? What is the purpose of the function?
3. Continue reverse engineering the main function following the call to fun\_401190 (which you should have renamed) to the second strcat function. What is the code doing in the main function between the call to fun\_401190 and the second strcat?

**Question**

1. Return to the main function and continue reverse engineering the code until the fun\_4012C0 function call. What is the code in the main function doing until the function call to fun\_4012C0?

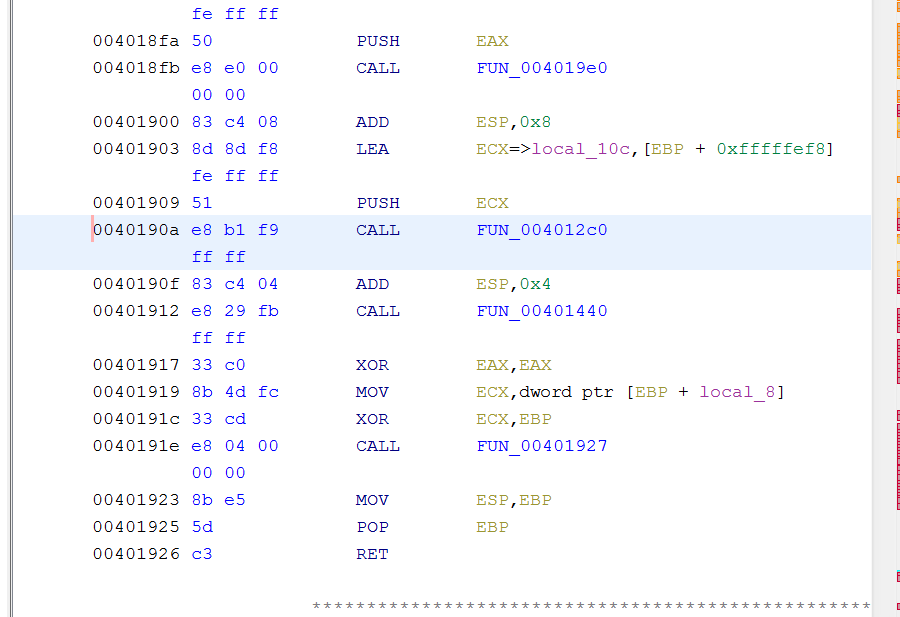


Figure 40: The Final Part of the Main Function

Source: Ghidra, 2020. 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.

## Reverse Engineering FUN\_4012C0

1. Open the function **FUN\_4012C0**.

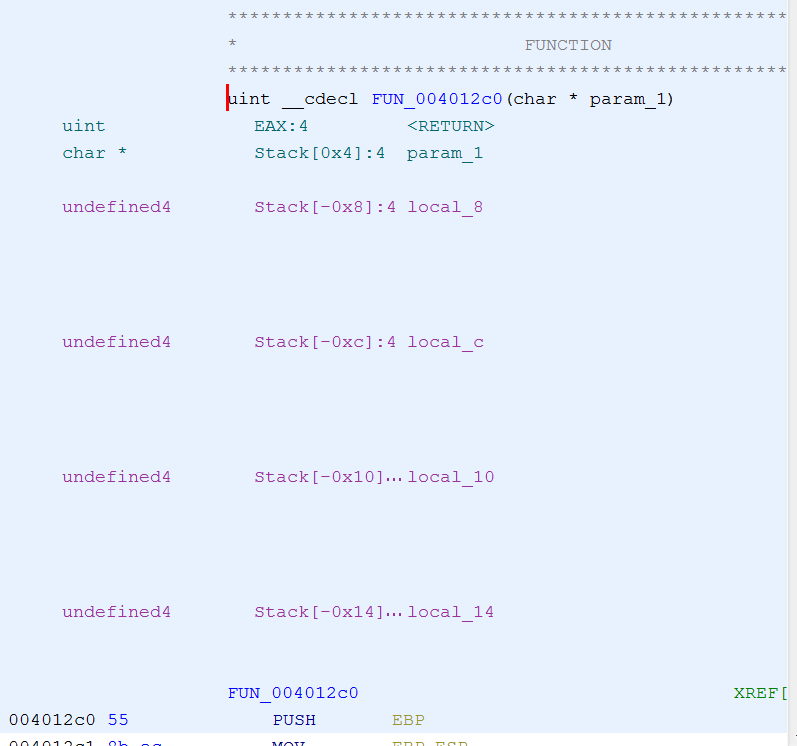


Figure 41: The Function sub\_4012C0

Source: Ghidra, 2020. 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. Look at this function to see something you haven’t seen before. The following instructions appear in the code block beginning with loc\_4012F9:

call dword\_413FA0

1. Double-click **dword\_413FA0** to see that it is not initialized with any value.
2. Notice that a number of arguments are being pushed on the stack prior to the call instruction, one of which is SOFTWARE\Microsoft\Windows\CurrentVersion\Run. The argument implies that the function call likely has something to do with the registry.
3. Can you determine where the value of dword\_413FA0 is set? (include the offset value)

**Answer:**

1. The most efficient way to determine where dword\_413FA0 is located is to place your cursor on it and press the **CTRL+SHIFT+F** or right click and select References

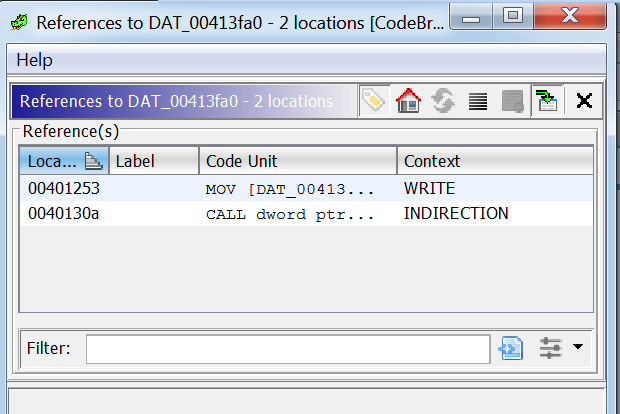


Figure 42: Cross References to dword\_413FA0

Source: Ghidra, 2020. 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.

Ghidra displays two cross references to dword\_413FA0.

1. Open the function **sub\_401220**.

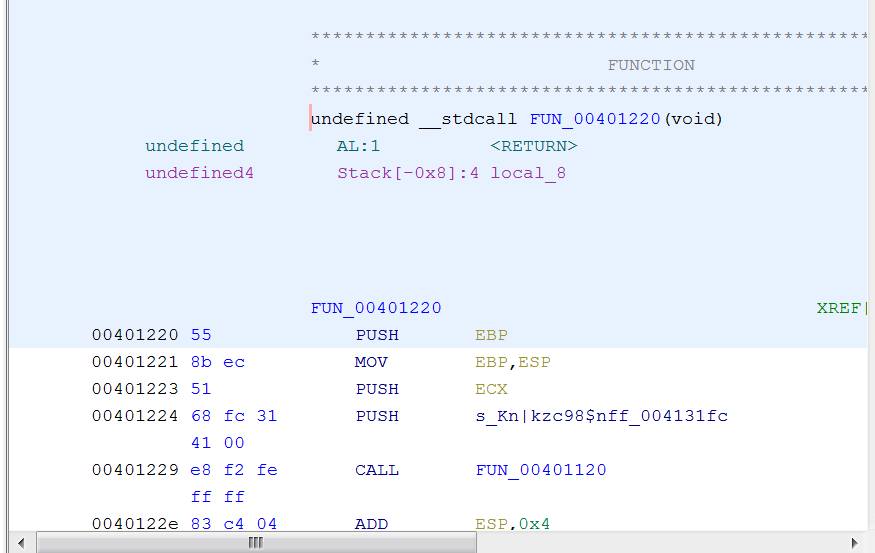


Figure 43: The Function sub\_401220

Source: Ghidra, 2020. 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.

The function sub\_401220 uses the Windows API functions LoadLibraryA and GetProcAddress. LoadLibraryA allows a program to load a DLL by name. GetProcAddress gets the address of an exported DLL function by name from the passed-in DLL handle.

1. Notice some seemingly nonsense strings used as arguments to a number of calls to the function sub\_401120.
2. Open the function **sub\_401120**.

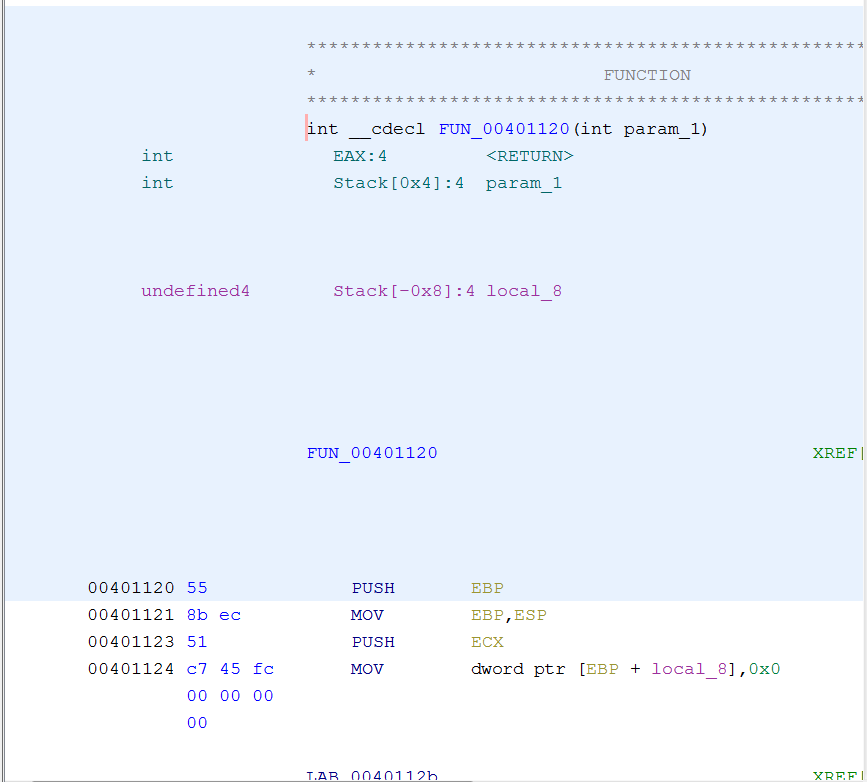


Figure 44: The Function sub\_401120

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. What does the function sub\_401120 do? (Rename the variables and add comments as appropriate.)

**Answer:**

Now that you know what sub\_401120 does, you can perform its action on its arguments. You could copy the strings from Ghidra to another program and do it manually.

The following is the skeleton of a Python script used to perform the action of sub\_401120:

def decrypt():

addr = GetOperandValue(ScreenEA(), 0)

idx = 0

s = ''

while Byte(addr+idx) != 0:

s += 'a' # Need to fix this

idx += 1

if AskYN(0, 'Rename string at 0x%x with "%s"?' % (addr, s)) == 1:

MakeNameEx(addr, s, 0)

AnalyzeArea(GetFunctionAttr(ScreenEA(), FUNCATTR\_START),

GetFunctionAttr(ScreenEA(), FUNCATTR\_END))

1. Fix the line in the script with the comment **# Need to fix this** to actually perform the action of sub\_401120.
2. After you fix the line, copy the new decryption function as you did before.

The new code replaces the old code.

1. Once the decrypt() function is working, ensure that you decrypt all the strings in sub\_401220.
2. Rename the following variables in sub\_401250: **dword\_413FA0**, **dword\_413FA4**, **dword\_413FA8** and **dword\_413FAC**.
3. With those variables renamed, rename **sub\_401220** to reflect its purpose. What will you call it?

**Answer:**

1. Return to sub\_4012C0 (all function calls should now be named). What is a good name for sub\_4012C0? What is the purpose of the function?

**Answer:**

1. Return to the main function.

## Reverse Engineering FUN\_401440

Notice that FUN\_401440 is by far the most complicated function you have seen in this binary so far.

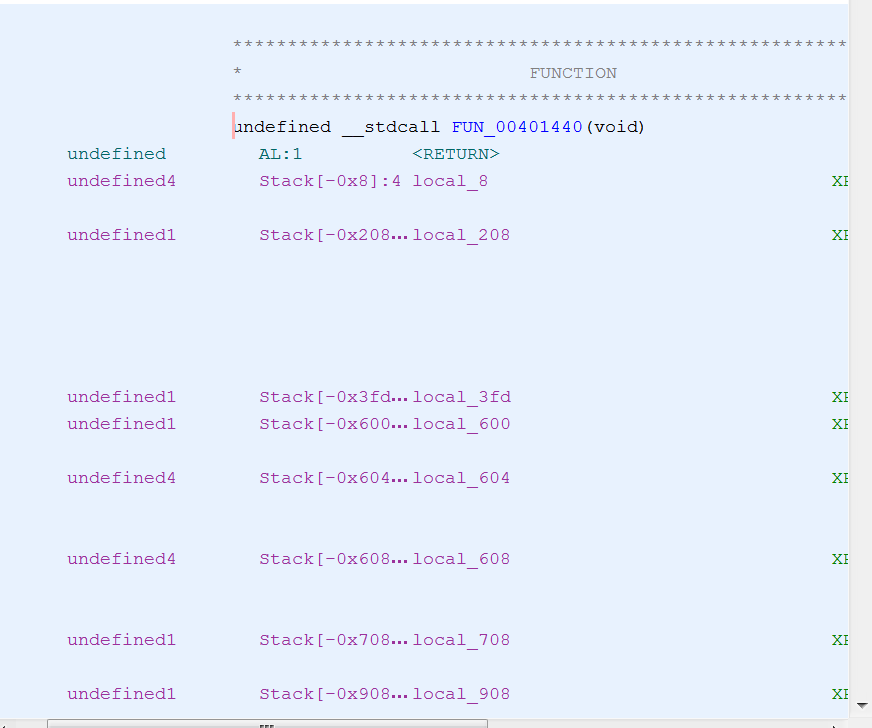


Figure 49: Graph View of sub\_401440

Source: Ghidra 2020 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.

Although the function is complicated, it is relatively simple compared to in-the-wild malware samples. For now, you don’t need to understand each part of it precisely. However, you do need to look at it and provide a general explanation of what it is doing.

**Questions**

1. What is a good name for FUN\_4014400?
2. What is the purpose of the function? What does the function allow the malware to do?

**Hint:** Look at FUN\_4013A0 and FUN\_403F0 also called from this function.

1. What are the names of the commands this function supports?

**Hint:** Ghidra lets you convert data values from hex, to int, to string. You can do this by right-clicking a data value you want to convert. This will help you find the command names.

**Command 1:**

**Command 2:**

**Command 3:**

**Command 4:**

References

National Security Agency (NSA), 2020 [Computer software]. Retrieved from https://ghidra-sre.org