**Lab – Introduction to Using Ghidra**

**Overview**

In this lab, you will be introduced to some of the higher-level features of Ghidra. Ghidra is a software reverse engineering (SRE) framework developed by NSA’s Research Directorate for NSA’s cybersecurity mission. Ghidra helps analyze malicious code and malware and can give cybersecurity professionals a better understanding of potential vulnerabilities in their networks and systems.

Ghidra was first released to the public by the NSA in 2019 after being referenced in the WikiLeaks’ March 2017 “Vault 7” disclosure, which discussed several hacking tools used by the CIA disclosing Ghidra as a reverse-engineering tool created by the NSA.

**Lab Requirements**

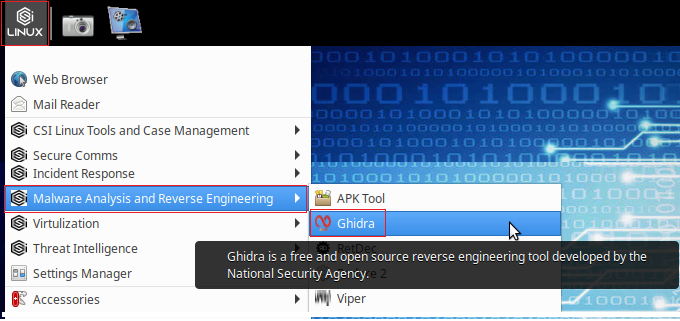
* On installation of VirtualBox with the extension pack.
* One virtual install of the latest version of Ubuntu, Kali Linux, or CSI Linux.
* One virtual install of Windows 7 or 10
* VirtualBox network adapter set to NAT network.

**Launch Ghidra**

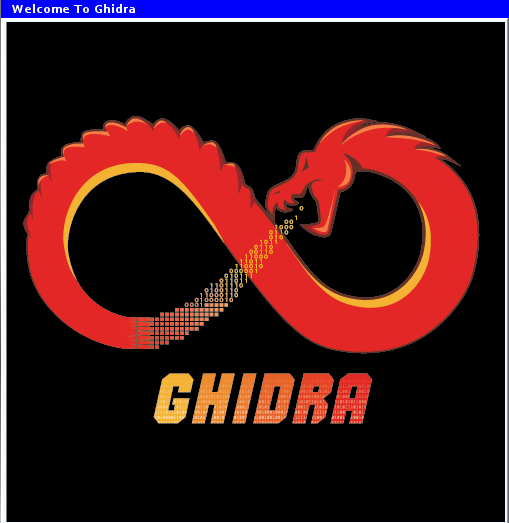
You can launch Ghidra using the Ghidra icon located in the bottom taskbar



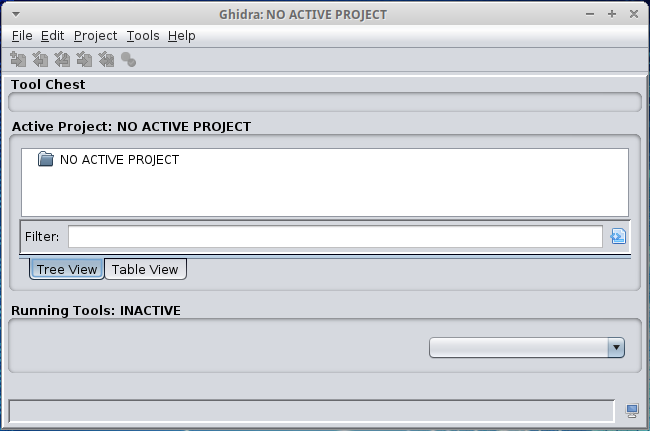
Your other option is to use the Application launcher. Scroll down to **Malware Analysis and Reverse Engineering** and launch **Ghidra** from the content menu.



Ghidra launches

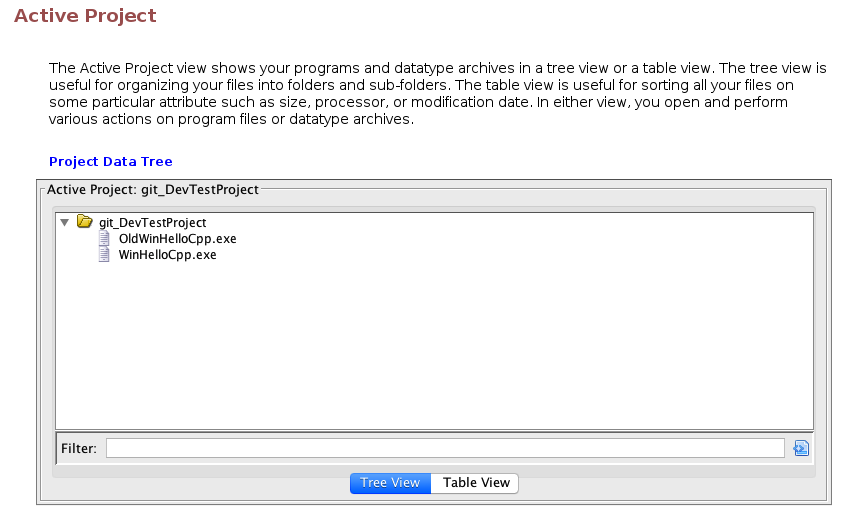


After a short pause, the Active Project screen opens.



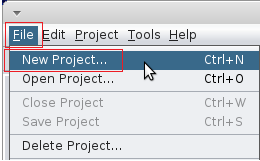
Anytime you need to access the Ghidra help documentation, press the F1 key. Regardless of the Ghidra feature or tool you are currently using, the help documentation will open to the section in the help documentation that references that feature.

Currently, we are working within the Active Project screen. When I press F1, the help documentation opens to the section with information about using the Active Project window.

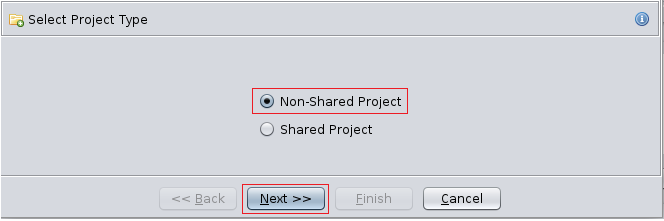


**Creating a new project**

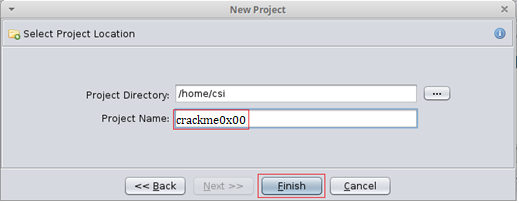
We next need to create a new project. From the Ghidra taskbar, go to File>New Project.



The **Select** **Project Type** screen opens. Ghidra can be used as a collaboration tool or as a stand-alone, non-shared project. We will be working individually on a project, so click “Non-Shared Project.” Then Click Next.



On the next screen, we give our new project a user-friendly name and select a location to save our work. In this example, I have named my first project, **crackme0x00,** and I will accept the default location for saving my work. You are free to name your project as you see fit and save it where you want. Click Finish.

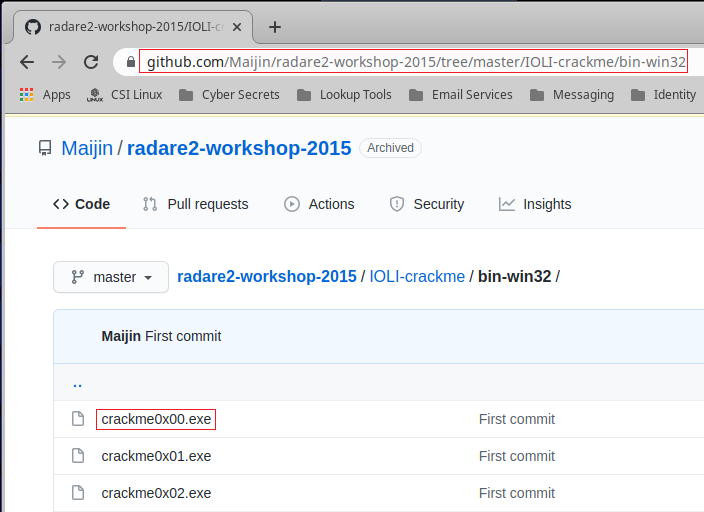


We next need a file to analyze. We can get some rudimentary files to crack using the following GitHub repository.

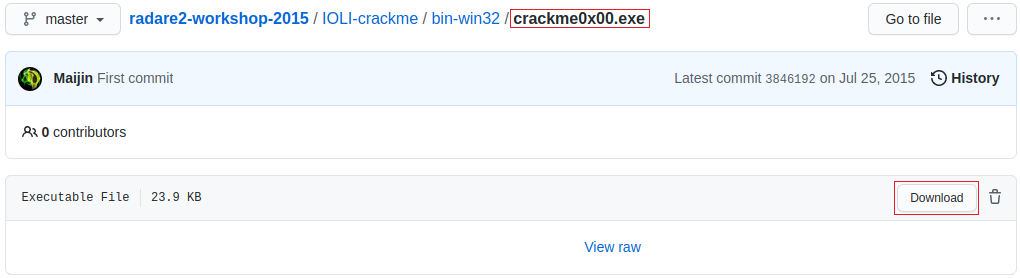
<https://github.com/Maijin/radare2-workshop-2015/tree/master/IOLI-crackme/bin-win32>

From your CSI Linux, open your default browser and copy and paste the following URL into the address bar.

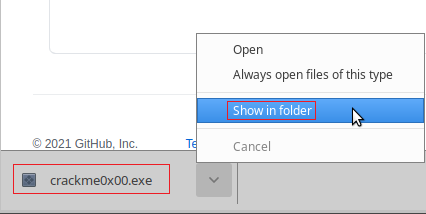
We will be analyzing the **crackme0x00.exe** file. From the list of crackme files, click on the crackme0x00.exe file.



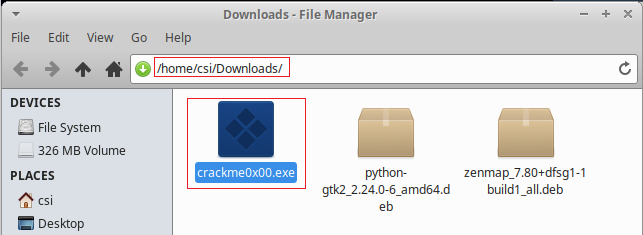
On the next page, click on the Download button.



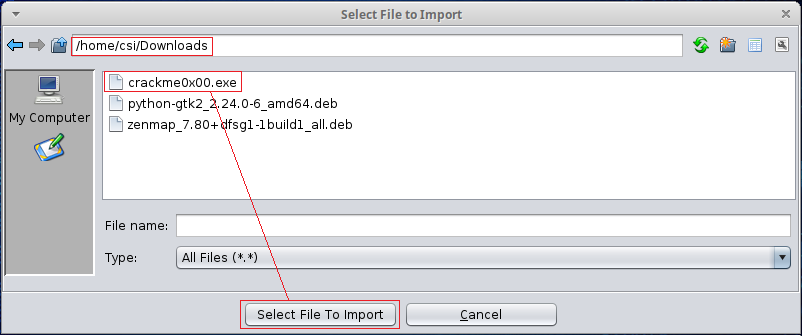
Once the file downloads, click the down arrow, and from the context menu, select **Show in folder**.



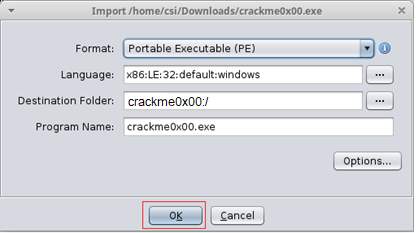
Take note that the file is in your Downloads Directory.



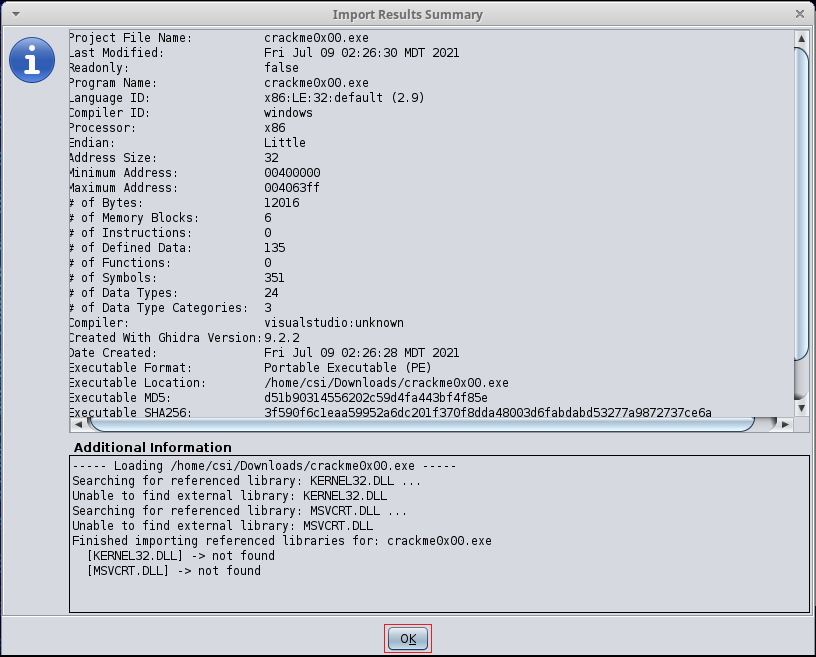
Close out your browser and the File Manager. Return to Ghidra. There are two ways to import our downloaded file into Ghidra. You can click on File> Import File or open your Downloads directory and drag and drop the crackme0x00.exe into the project window.



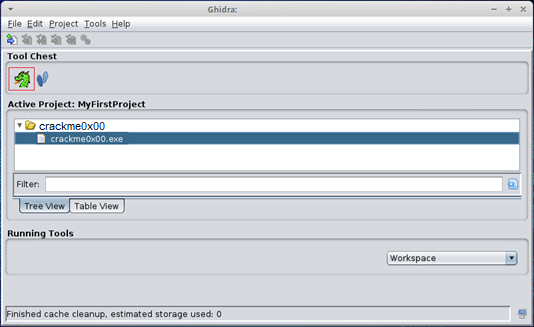
When you first import your file, Ghidra will respond with the information about the file. Click OK.



Ghidra then displays a screen like the one shown below with an import summary of the file. Click OK.

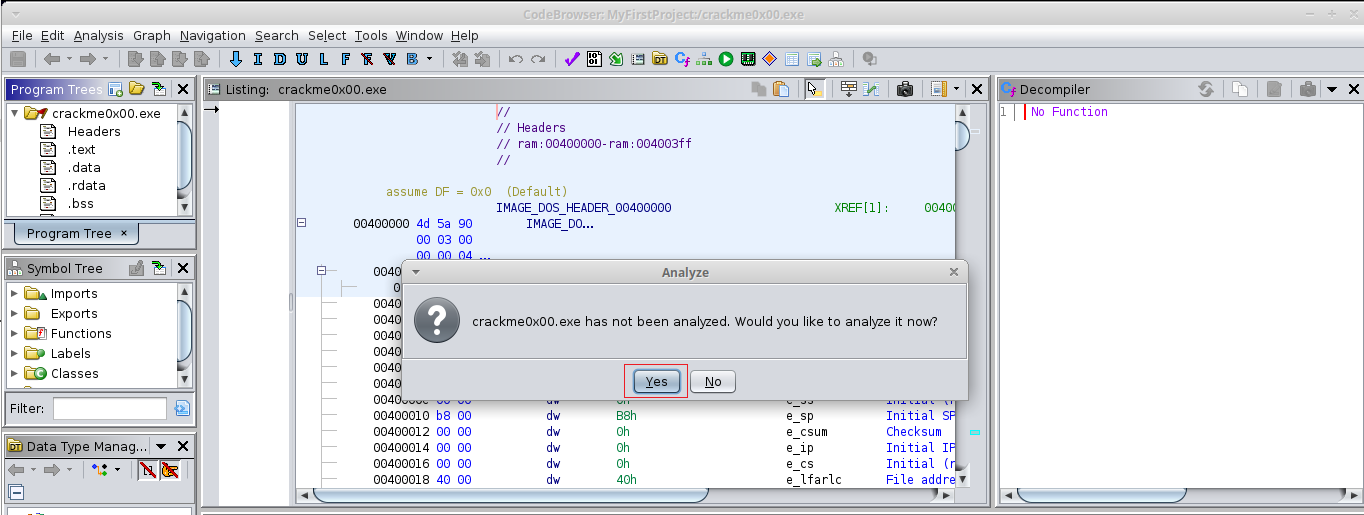


Back at the Active Project window, you can either double-click on the file or use the green Ghidra dragon to begin the analysis.

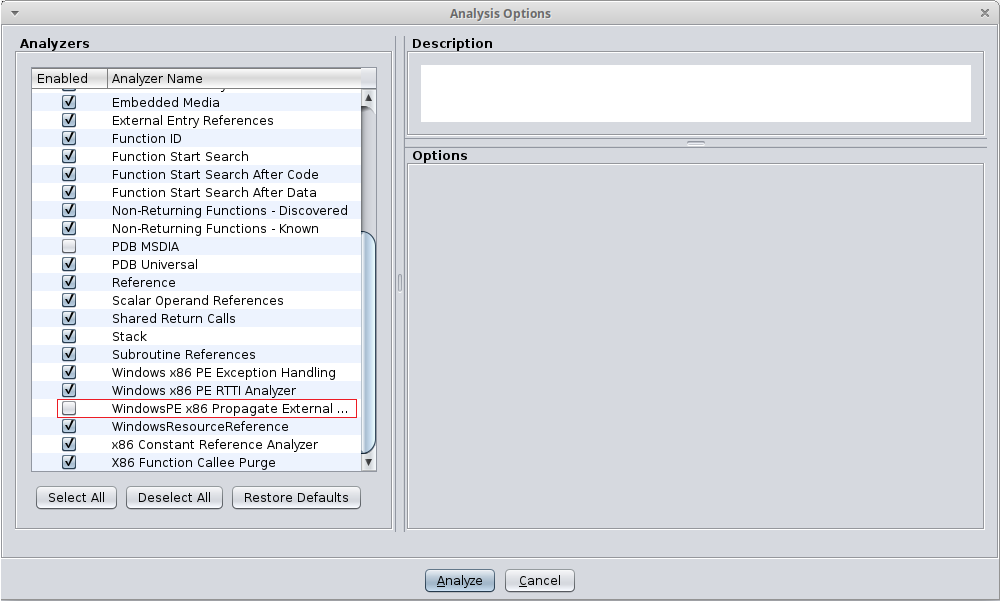


Ghidra first displays the file’s assembler language in the center Listing window and asks if you want to analyze the file. Click “Yes.”

If the Ghidra fails to show any data in the Listing window, close out the Listing window and back the Active Project window, x2 click the file one more time and relaunch. This is a well-known feature of Ghidra.

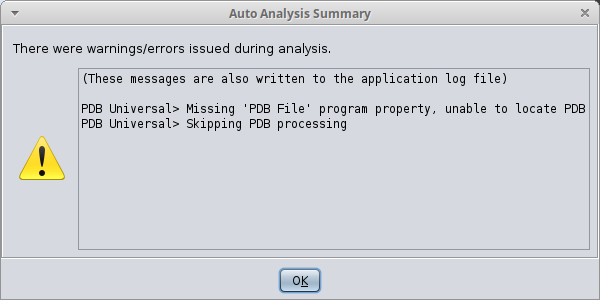


On the next screen, you have your analyzing options:



Select the **“WindowsPE x86 Propogate External Parameters”** option. This analyzer populates push instructions with comments which might help us make sense of the binary file more easily. We then select “Analyze,” and Ghidra starts analyzing our executable.

Upon completion, we are presented with the following error message regarding the “PDB”:

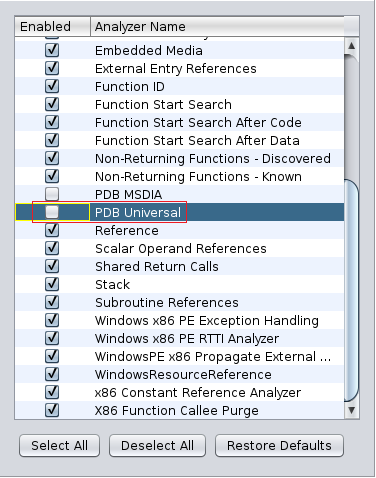


Click OK.

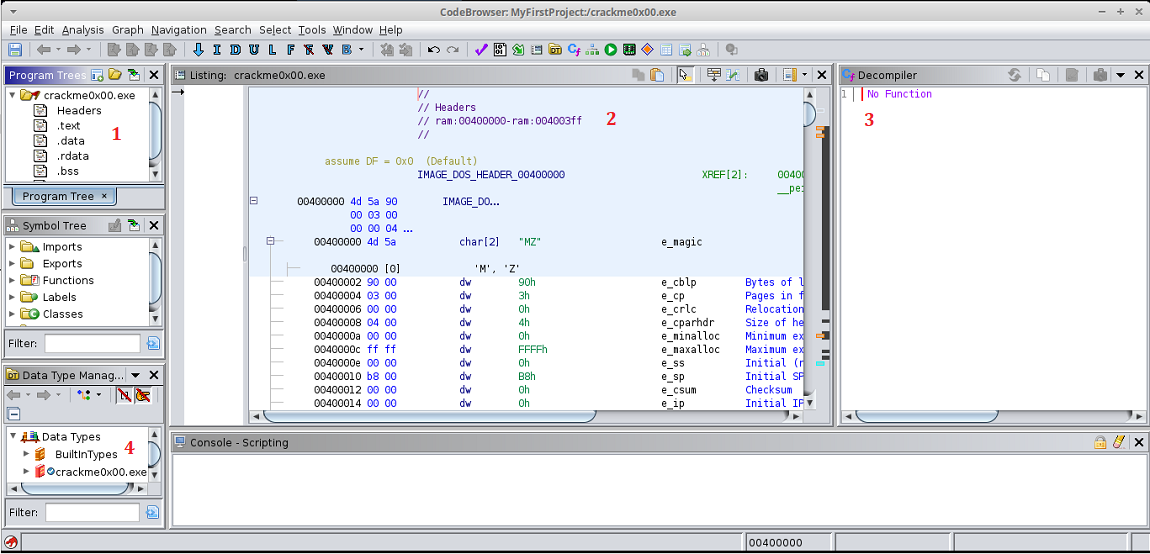
A Program Database (PDB) file contains information intended to help the debugger and user debug the file. PDB files, sometimes referred to as “symbol files,” were developed by Microsoft and are “typically created from source files during compilation” (docs.microsoft.com). They store a list of “symbols in a module with their addresses and possibly the name of the file and the line on which the symbol was declared” (docs.microsoft.com).

Since we specified that our PE file was compiled using Visual Studio in our import file step, Ghidra automatically searches for a PDB file associated with our executable when we run our “Auto Analysis.”

To handle this error, we can uncheck the PDB box before we run our Auto Analysis:



Ghidra will now analyze your file and display the information as seen in the five windows below.

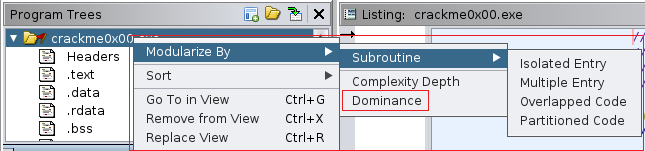


**The left Windowpane**

**The Program Trees Manager**

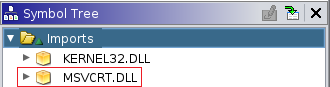
The program Trees Manager is used to organize programs into a tree-like structure. In addition, the Program Tree Manager allows you to create, delete, rename, and close program tree views.

Within “Program Trees,” you can right-click on the “crackme0x00” folder to organize the sections of disassembly code in different ways. You can do this by selecting “Modularize By” and choosing “Subroutine,” “Complexity Depth,” or “Dominance.” You can also make new folders and drag/drop sections according to your organizational preferences. For example, under Subroutine, choose Dominance.



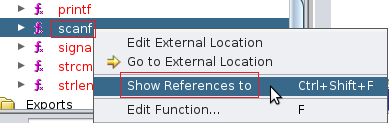
**The Symbol Tree Window**

Directly below the “Program Trees” is the “Symbol Tree.” Within this window, you are shown all the different symbols used with the program itself. The different symbols are shown as folders labeled imports, exports, functions, labels, classes, and namespaces. Try expanding the “Imports” section to see the various DLLs and functions used by the target.

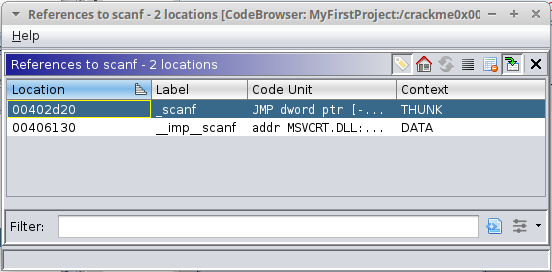


Open MSVCRT.DLL under imports. Scroll down until you come to function, scanf.

Right-click on the function and select “Show references to”



Another window pops up showing what the scanf function references.

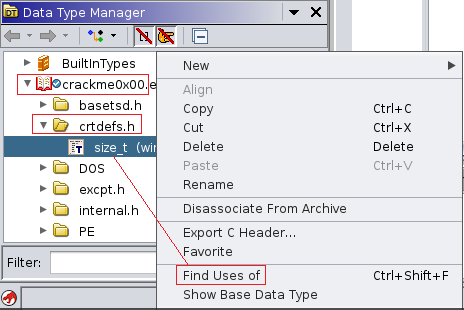


This window displays the breakdown of the code in assembler language.

**The Data Type Manager Window**

Below the Symbol Tree window, we have Data Type Manager. The “Data Type Manager” allows you to see all the defined types, including the built-in types, those specific to the binary, and others that were included with Ghidra (such as the Windows ones we see called “windows\_vs12\_32”).

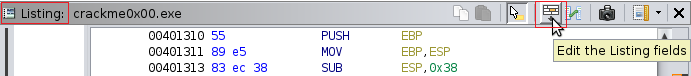
By expanding the book icon labeled crackme0x00.exe, right-clicking on a data type **size\_t** **(windows\_vs12\_32),** and selecting “Find uses of,” we can see where a data type is being used within the binary.



**The Main Windows**

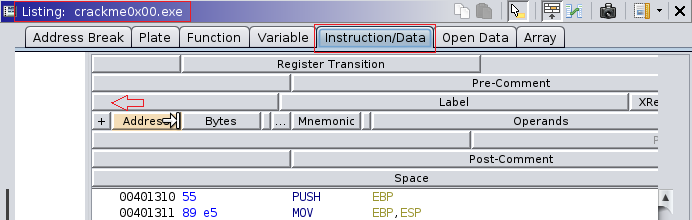
**The Listing Window**

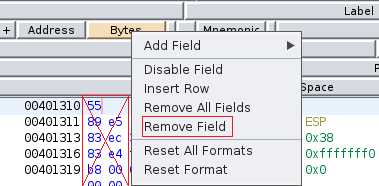
The Listing Window shows us the disassembled code and allows us to begin piecing together what different portions of the binary are doing. To customize our listing window view, we can click on the “Edit the listing fields” icon in the top right.



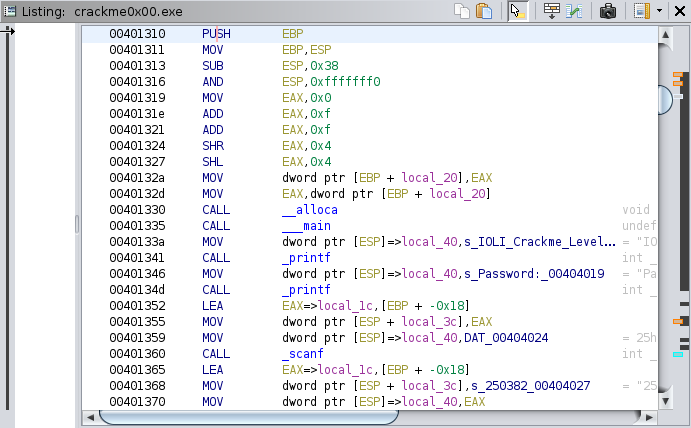
Clicking on the “Instruction/Data” tab gives us access to each element shown in the listing interface. These elements can be re-sized, moved around, disabled, or deleted. You can also add new elements by right-clicking and using the contextual menu.

Try re-sizing the “Address” field to make it smaller.

****

****Delete the “Bytes” field.

Close out the editor and return to your Listing Window. The address field has been moved to the left, and the bytes field is no longer present.



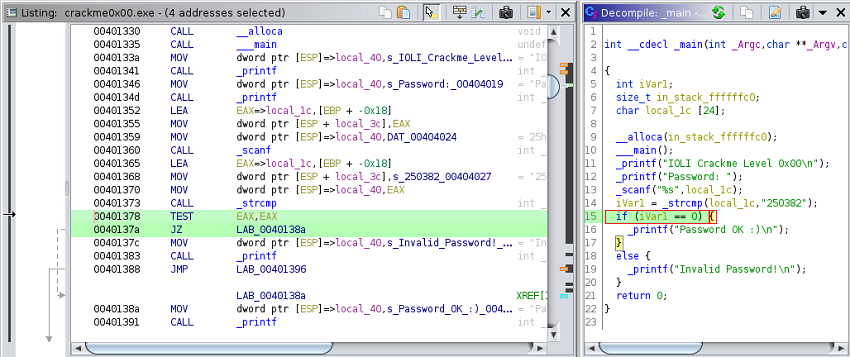
**Contextual Editor**

By right-clicking anywhere within the assembly code shown in the Listing Window, the contextual menu within the disassembly listing can be seen. The contextual menu allows you to perform patching instructions, setting a bookmark, commenting, and editing labels.

**Decompile Window**

To the right of the Listing Window, we have the Decompile window. The Decompile Window shows us Ghidra’s best estimation of the high-level code representing the assembly code in the listing/function graph windows.

Highlight one of the “if” statements shown in the decompile window. In the Listings Window, you will see that it highlights the corresponding assembly. This feature allows you to build a mental mapping of what groups of assembly instructions map to which high-level instructions.



The Decompile Window allows you to see what the high-level language would most likely look like.

**Summary**

Reverse engineering malware is one of the highest-level skillsets within cybersecurity and one of the highest-paid. Ghidra is an excellent reverse engineering tool capable of running on nearly any platform and priced very attractively (free). In this series on Reverse Engineering, we will be using this tool from the US NSA to reverse engineer multiple pieces of malware, beginning with the simple and progressing to the more advanced.