# Ghidra Project

This is a project for Systems Architecture in which our team has to write about ghidra

This is the first phase of the project, in which each member of the team has had to write 1 of the following tasks

* a) [State-of-the-art (adica care este statusul tehnologiei la acest moment)](#chapter-1-task-a)
* b) [Care sunt perspectivele de dezvoltare](#chapter-2-task-b)
* c) [Care sunt functionalitatile acoperite](#chapter-3-task-c)
* d) [Exemplu si eventual un studiu de caz](#chapter-4-task-d)
* e) Concluzii

Each entry ends with a conclusion, and as such, the tasks were distributed in the following order:

1. Pandelea Dariand on a) [State-of-the-art](http://docx/State-of-the-art.docx)
2. Roșescu Tudor on b) [Development perspectives](http://docx/Development%20perspectives.docx)
3. Neamţu Alexandru on c) [Functionalities curently present](http://docx/Functionalities%20curently%20present.docx)
4. Dumitrescu Codrin on d) [Example on how to use ghidra](http://docx/Exemplu%20si%20eventual%20un%20studiu%20de%20caz.docx)

All entries are converted into the .docx Ghidra Project, containing the entries present inside of this README.md

## Chapter-0 Introduction

## Chapter-1 Task-A

The reverse engineering field of disassemblers is, by its nature, an ever changing and evolving one. There is an ever-present need for state of the art programs that can guarantee efficient translation of machine code into assembly.

Having access to a large variety of processors and file formats is essential. At the moment, IDA Pro offers the most vast range, including support for certain exotic architectures like RISC-V and SPARC. Taking that into consideration, Ghidra comes close behind with support for most file formats, including modern ones, and for most processors including their variants, while allowing for the addition of new architectures or extending pre existing ones.

Processors such as:

* Z80
* X86 16/32/64
* 6502
* ARM/AARCH64
* 8051
* PowerPC 32/64/VLE
* MSP430
* MIPS 16/32/64/micro
* AVR8
* 68xxx
* Java / DEX bytecode
* AVR32
* PA-RISC
* PIC 12/16/17/18/24
* Sparc 32/64
* CR16C

In terms of decompilers, IDA Pro’s "Hex-Rays" is best thanks to its accuracy being comparable to that of a human. The technological advancements implemented through IDA 9.0 also introduced new decompilers for 32- and 64-bit RISC-V code, designed to modernize processes and increase capacities. Ghidra’s decompiler is powerful, but not as refined as Hex-Rays.

Furthermore, IDA Pro provides superior debugging and emulation for running and tracing target systems, particularly for Android and Windows.

Currently IDA Pro delivers greater support for patching binaries, interactively changing characteristics of displayed data, renaming variables, functions, structures or changing data presentation and creating custom types.

Regarding graph generation, Ghidra offers a more informative, highly customizable and detailed visualization of code, even marking “if … else” statements on the graph and making it more organized.

For working teams, facilitating collaborative reverse engineering is important. Ghidra’s design supports working on the same project, while IDA Pro requires certain plugins and the database files are not designed to be shared.

The support for cross-platform, while versatile, is implemented by few tools, like Ghidra, Rafare2 and JaDx, so that they work on macOS, Windows and Linux.

Concerning binaries, Ghidra is faster and loads multiple binaries into projects simultaneously, resulting in an easier approach to following code between applications and libraries, by using multi-threading and multi-core properties of the CPU. For bigger files, 150 MB+, IDA Pro has a better optimization of the memory and CPU usage, it can lessen the processing time, being altogether faster than the other.

The most significant cutting-age technology implemented recently is helping with pattern recognition accuracy and processing very large files due to integration of AI and machine learning in disassemblers. Models are being trained to increase code recognition patterns and data structures. A disassembler based on deep learning, DeepDi, proved to be simultaneously fast, highly accurate and efficient against obfuscated binaries, managing to discern valid instructions from the raw binary. For capturing and propagating instruction relations, it uses a neural network graphing model. It has a very low false positive rate and false negative rate when identifying instructions, 0.02%/0.02%, and compared to IDA Pro, the CPU version is twice as fast and the GPU version is 350 times faster. Taking in consideration code section size-disassembly time correlations, the CPU version has a throughput of 146 KB/s and the GPU one is 170 times faster at around 24.5 MB/s.

### Conclusion

At the moment, state-of-the-art disassemblers are extraordinary, powerful tools used in domains such as reverse engineering, software development and debugging, but still suffer from challenges like the ambiguity of instructions, the absence of symbols and debugging information and mostly from the presence of obfuscation techniques, but these difficulties will begin to become a problem of the past with the evolution of machine learning.

## Chapter-2 Task-B

DEVELOPMENT PROSPECTS FOR GHIDRA

1. Short Term: In the immediate future, Ghidra is likely to see enhanced community contributions, resulting in new plugins and features. User feedback will drive updates to improve usability and functionality. Additionally, ongoing support and bug fixes will ensure stability as more users adopt the tool.
2. Medium Term: Over the next few years, Ghidra may focus on integrating with other cybersecurity tools and platforms, streamlining workflows for analysts. Incorporating machine learning capabilities could become a priority, enhancing automated analysis of binaries and improving vulnerability detection.
3. Long Term: In the longer term, Ghidra has the potential to become a central hub in the cybersecurity ecosystem, with advanced AI features and an extensive library of educational resources. This would not only broaden its user base but also solidify its position as a leading tool for reverse engineering, adapting to the evolving cybersecurity landscape.

### Conclusion

Ghidra is set for growth, with short-term improvements driven by community feedback and plugin development. Medium-term focus will be on tool integration and machine learning enhancements. Long-term, Ghidra aims to become a central cybersecurity resource, supported by AI features and robust educational materials, solidifying its role in reverse engineering.

## Chapter-3 Task-C

Ghidra is a debugging program and reverse engineering tool, used for malware analysis and program dissection without danger to the user’s device

Firstly, it’s most commonly used for reverse engineering malware, general software and firmware.

Its wide array of tools and interfaces allow intermediate programmers to understand what they’re dealing with very easily, thanks to how it maps out functions

The graph function allows you to lay out the program into a graph, where you can see what the program is doing as interconnected boxes, like a family tree showing which functions are called by their parent function.

The emulator allows you to run the program in customizable environments, you can use the emulator to create a virtual machine on which to run your application, so as to not endanger your own computer.

The debugger lets you manually progress through the program. While it isn’t specific to Ghidra, it allows you to look at what the program does to better understand its inner workings, Ghidra has a large set of powerful debuggers to accommodate different workflows

Ghidra supports the creation of different tools written in python

### Conclusion

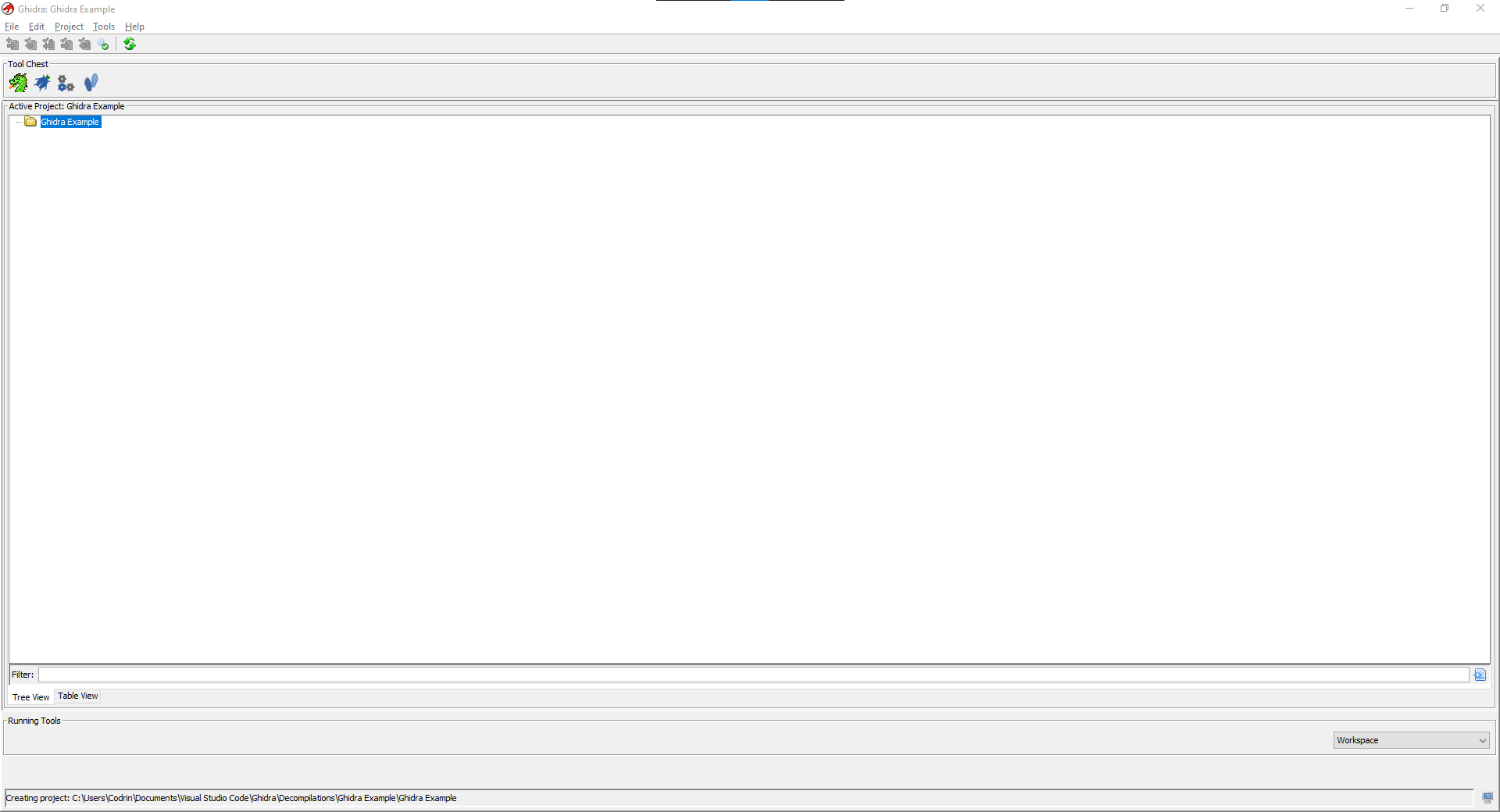
The current suite of tools provided by Ghidra can perform a large variety of tasks with ease

## Chapter-4 Task-D

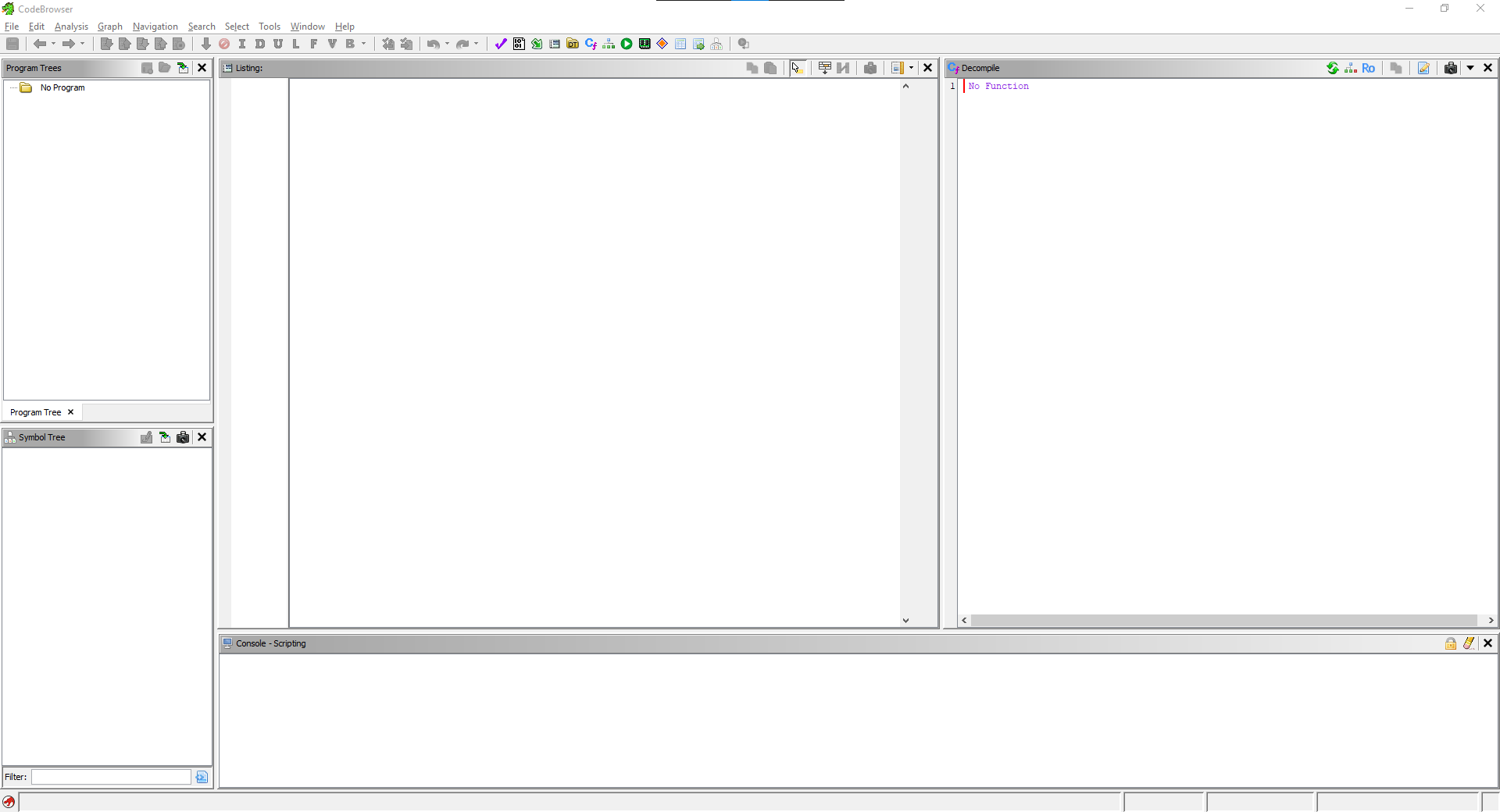
Ghidra, as an application, comes with its own documentation and help files. These help files include examples for using Ghidra and a detailed explanation on all steps to take.

This is an example from within the documentation, where the disassembly of an executable file "WinhelloCPP" is shown.

To begin, we will open up ghidra, and create a new project titled Ghidra Example.

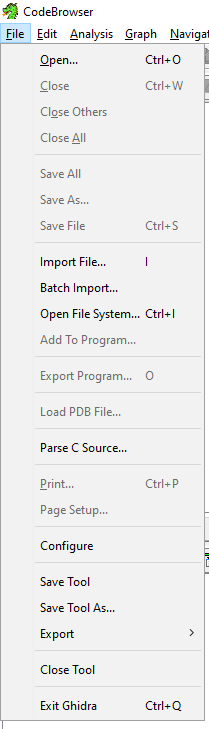


Inside of our new workspace, we can open the CodeBrowser tool



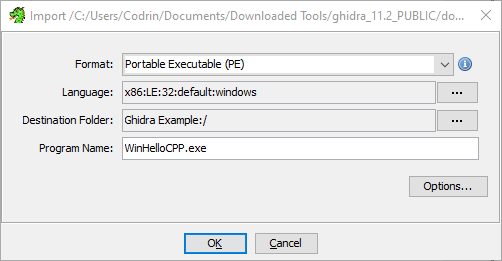
Then, we will have to import our file.

This is done by moving to File,



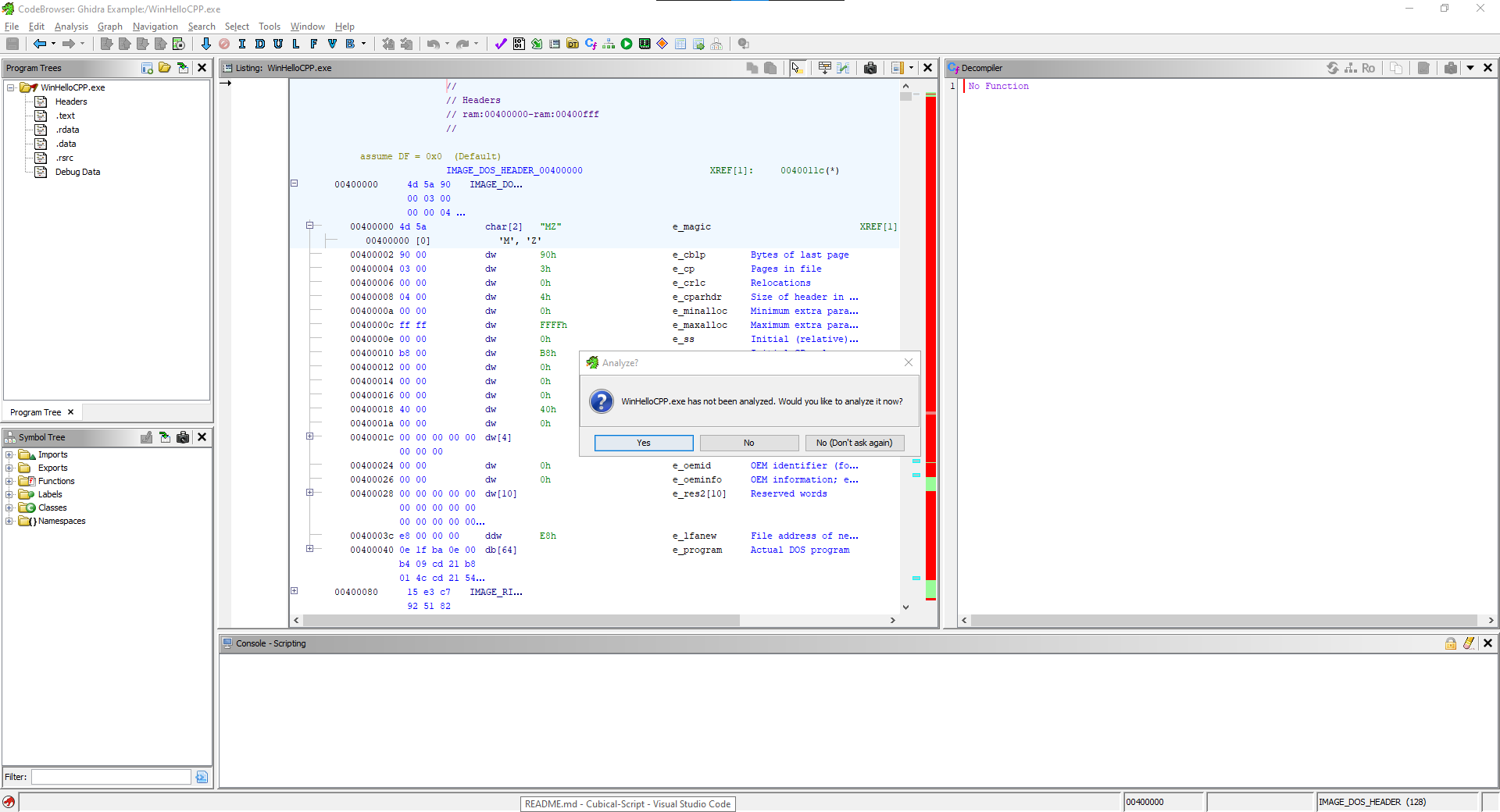
And selecting the file to import.

To initiate a disassembly, you must first specify the language of the file,

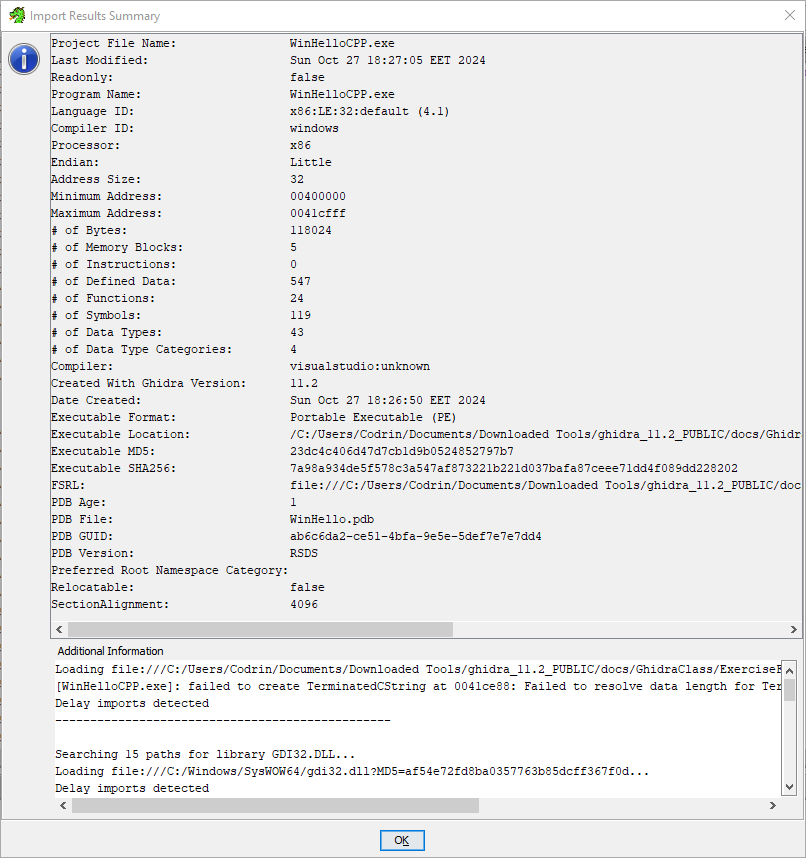


Ghidra usually detects it for you, but in the rare case it gets it wrong, the dropdown menu has all usable languages.

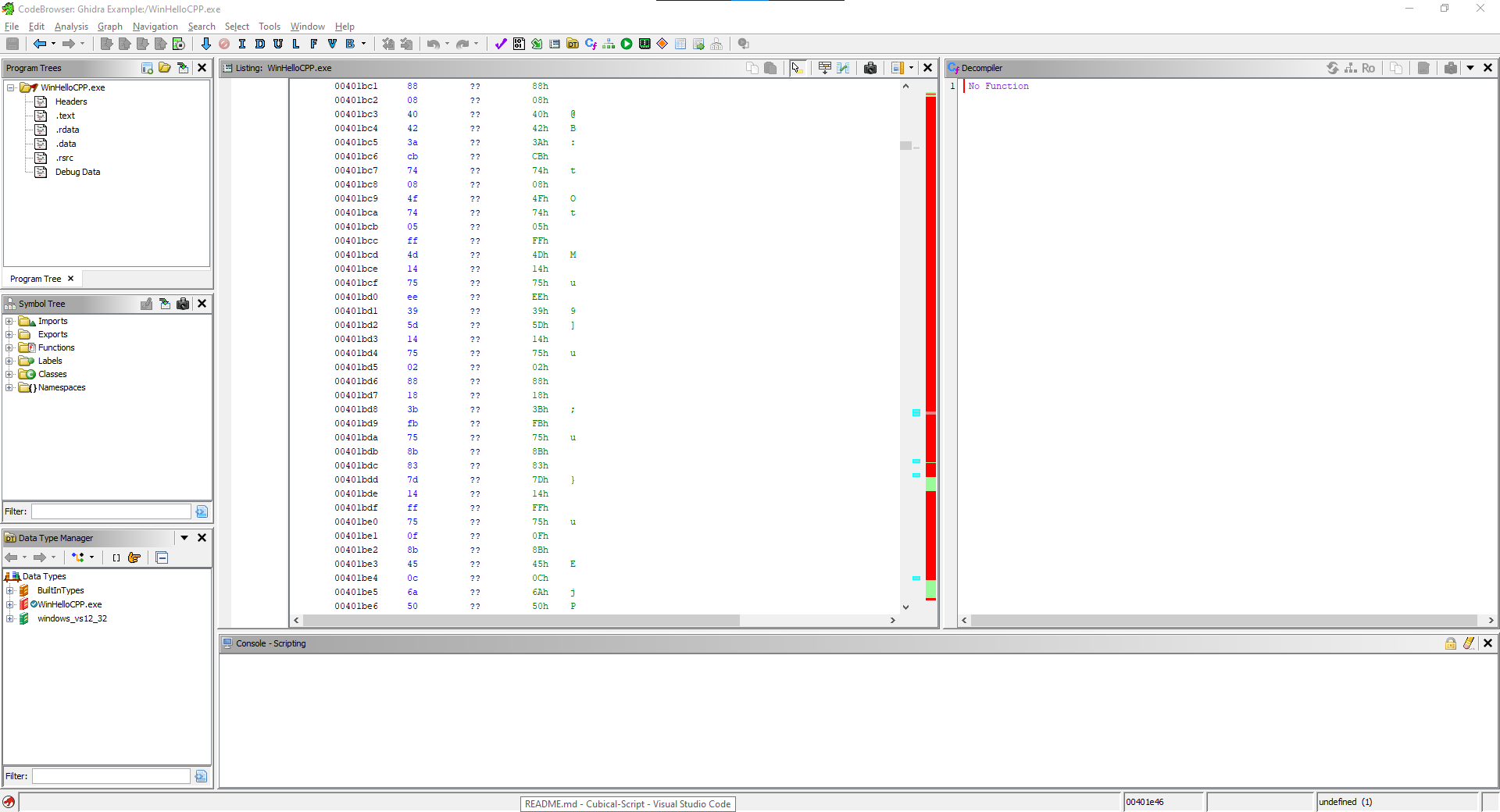
Click `Ok` to start disassembling the file. Once the process is finished, you will be prompted to perform an auto analysis,



For the purposes of this example, we will not perform an auto analysis just yet. After closing the prompt, you will be shown the import results.

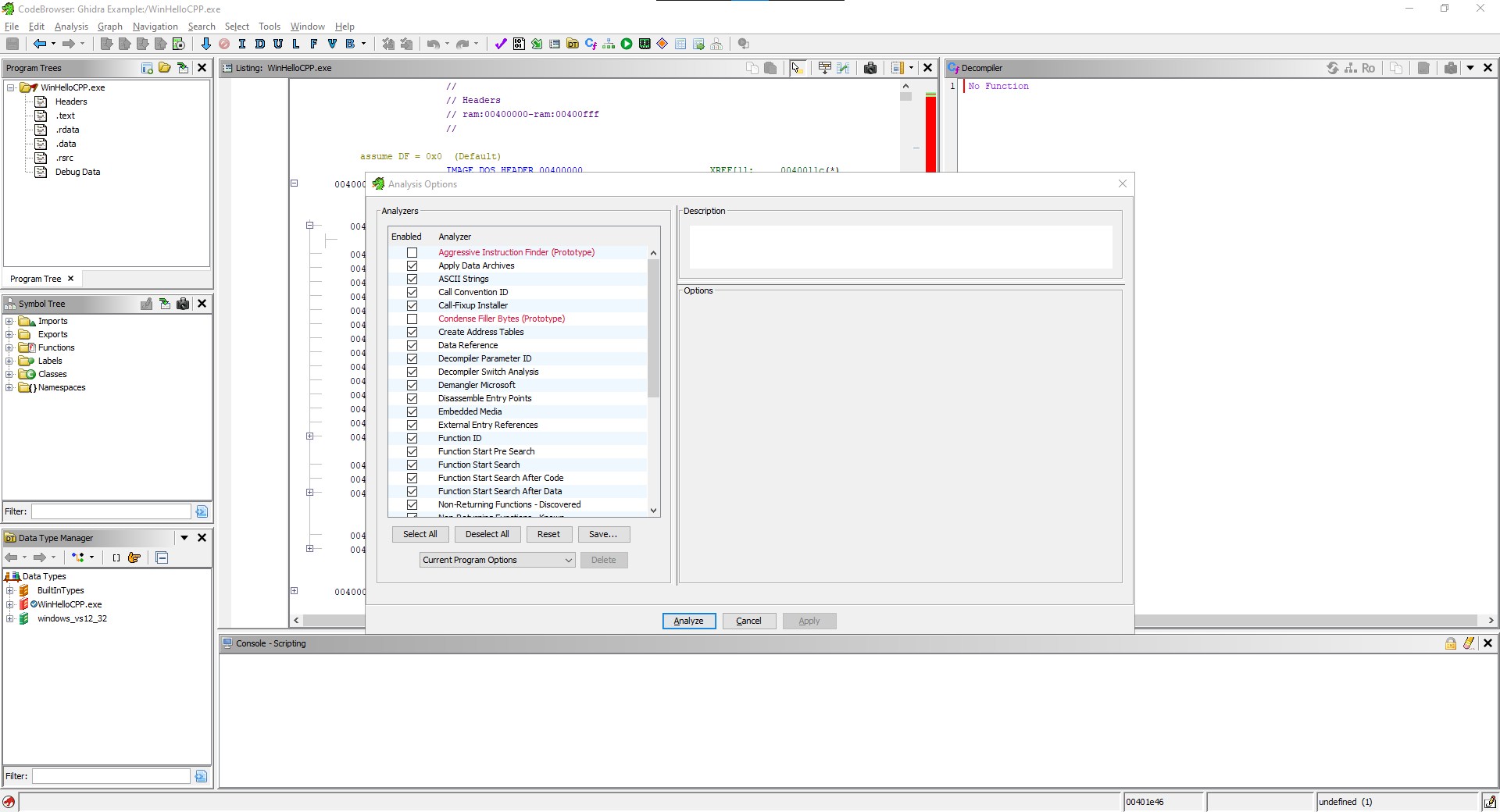


Closing these results, you will be able to see in the middle of the CodeBrowser the assembly code of the imported executable, The bar to its right, is the navigation slider, coloured red are unrecognized fields, being fields that ghidra could not recognise the contents of during disassembly.



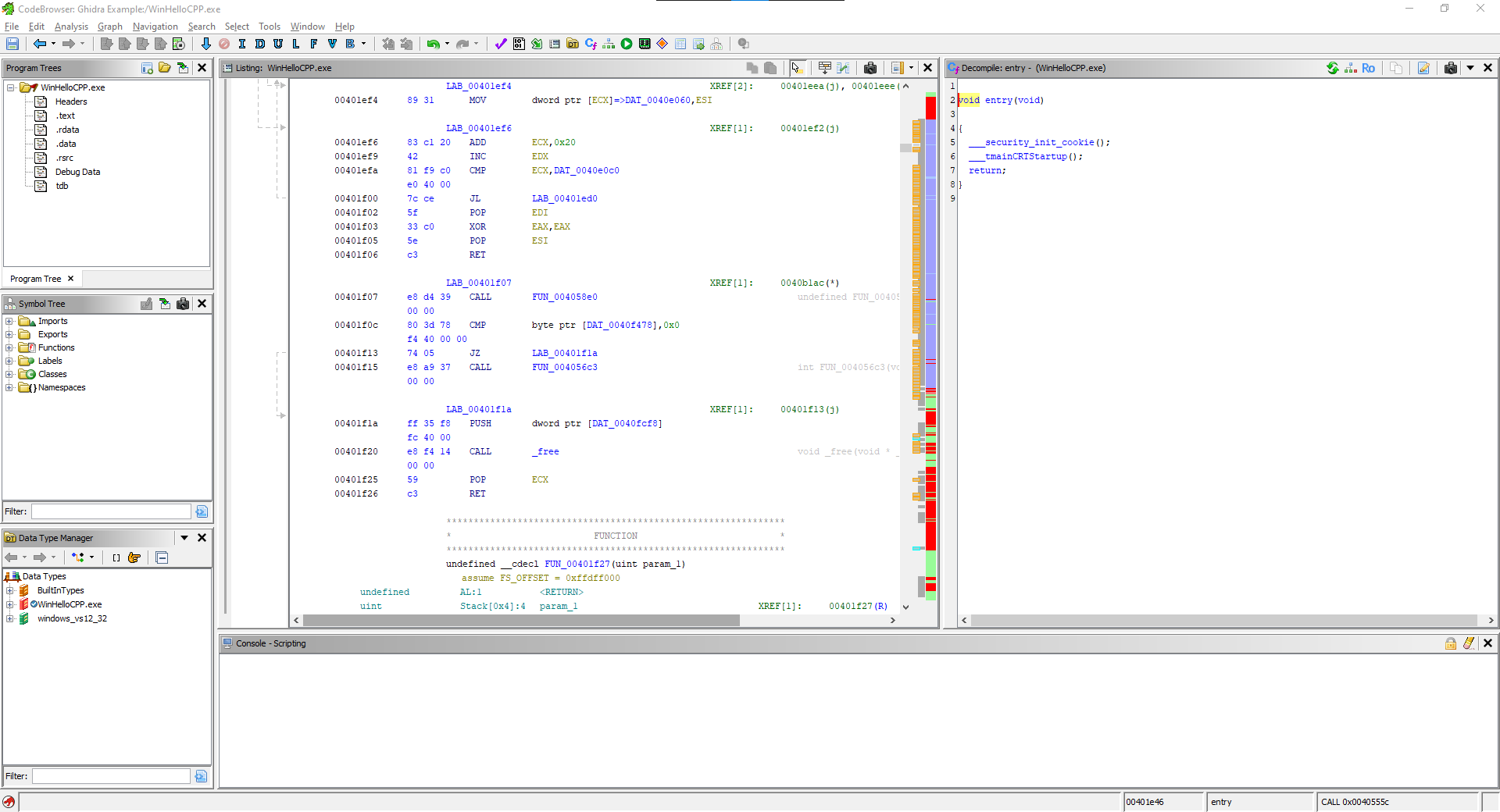
We will now request an auto analysis to display the changes it makes. This is done by clicking auto analysis inside of the Analysis tab, this request will bring up an analysis window, in which you can choose what you want to be done during analysis.

The configuration shown below is the default selection.



After the analysis is complete, you will be moved to the program's entry sequence, the starting point of the executable.

The code decompiler window at the right of our assembly code is a C representation of the actions performed by the entry function.

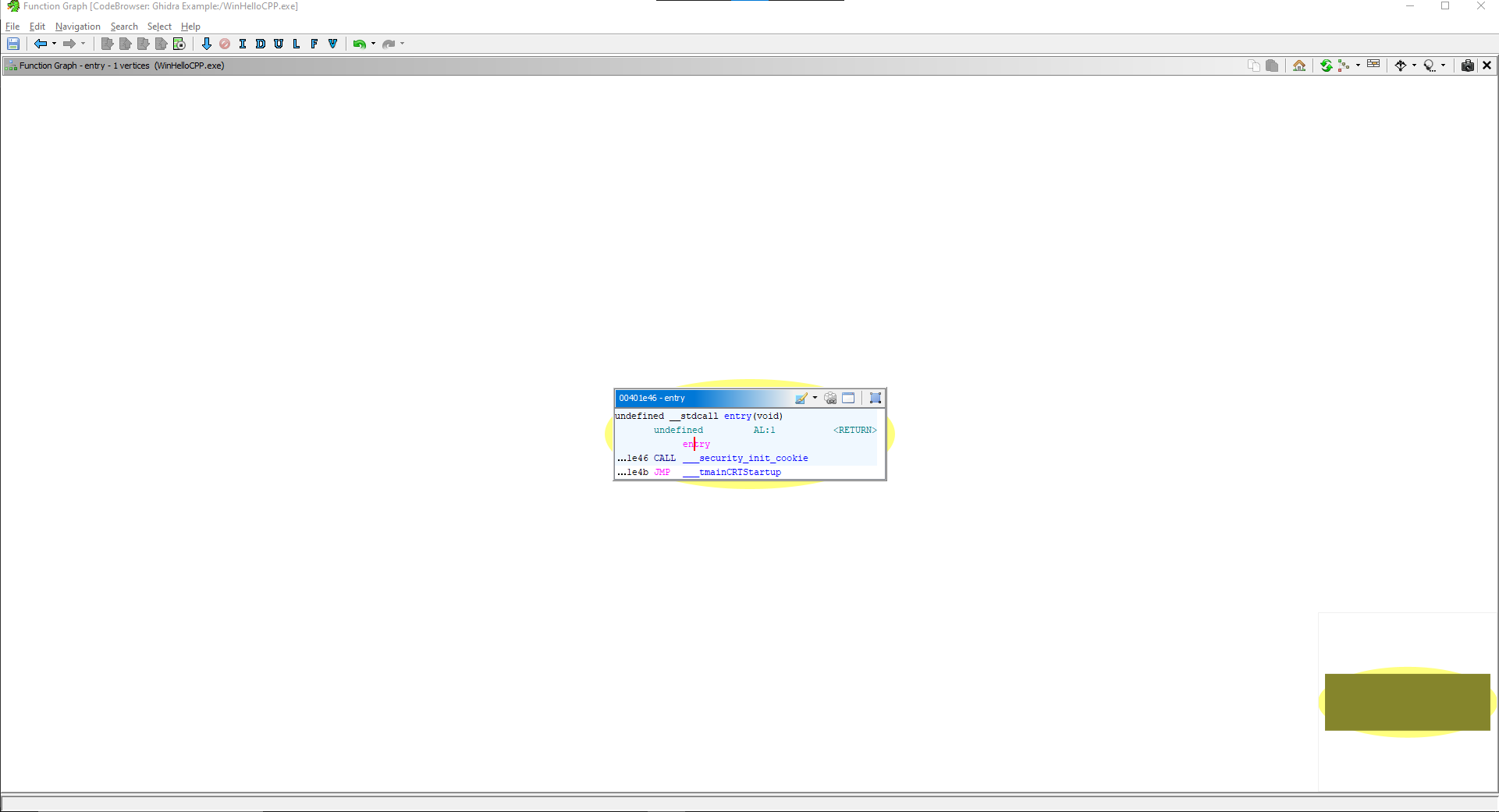


Additionally, you will see that the scrollbar has gotten populated with more colors, though the red sections are still visible, this is because sequences of code can remain unrecognized, even after analysis.

To not make this example too long, I will be covering the graphing tool inside of the code browser.

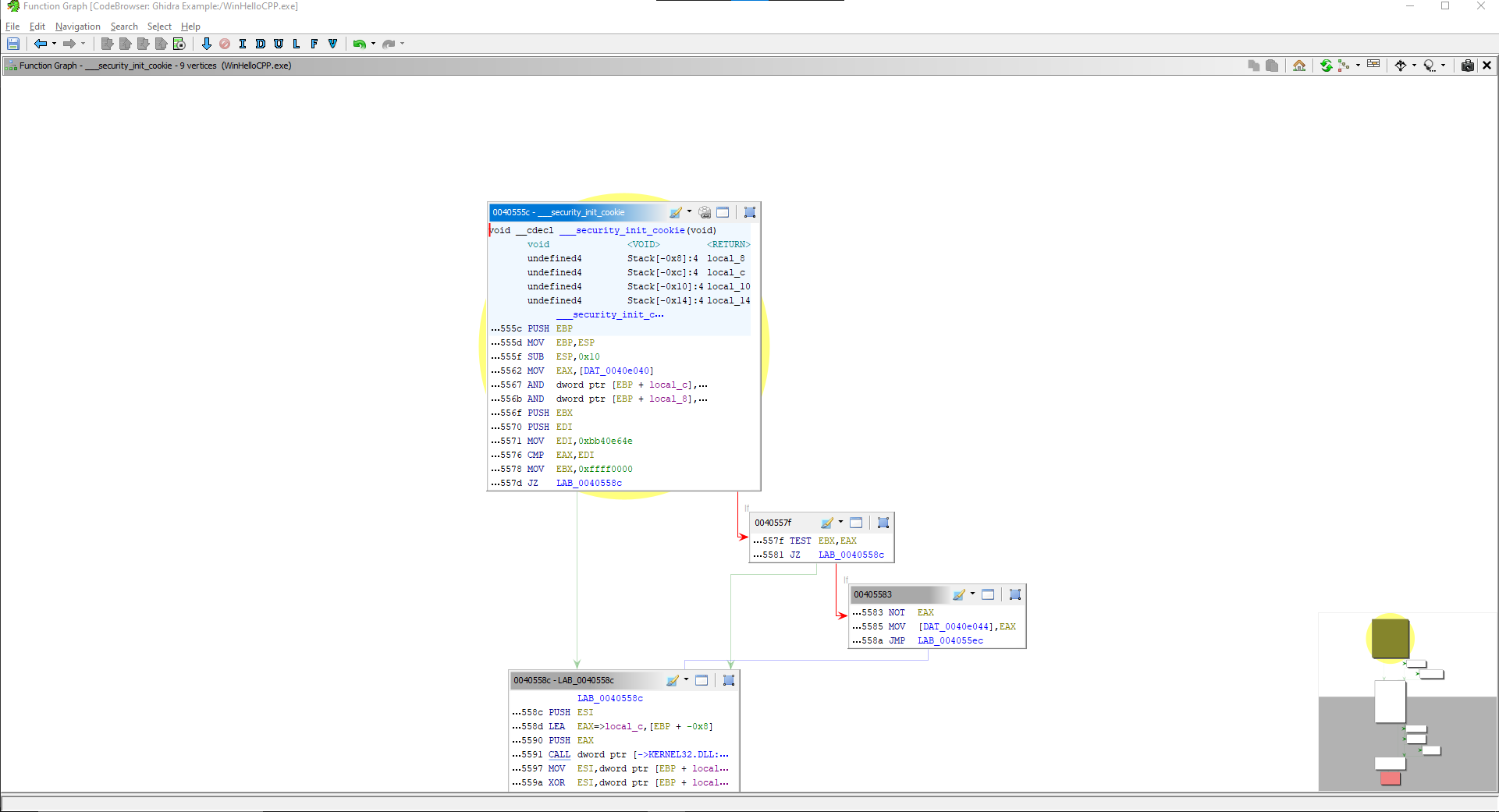
By clicking the graphing tool, you get a screen popup that shows the actions performed inside of the selected field of code.

Since I have selected the entry, this single box represents the actions performed when starting the executable.



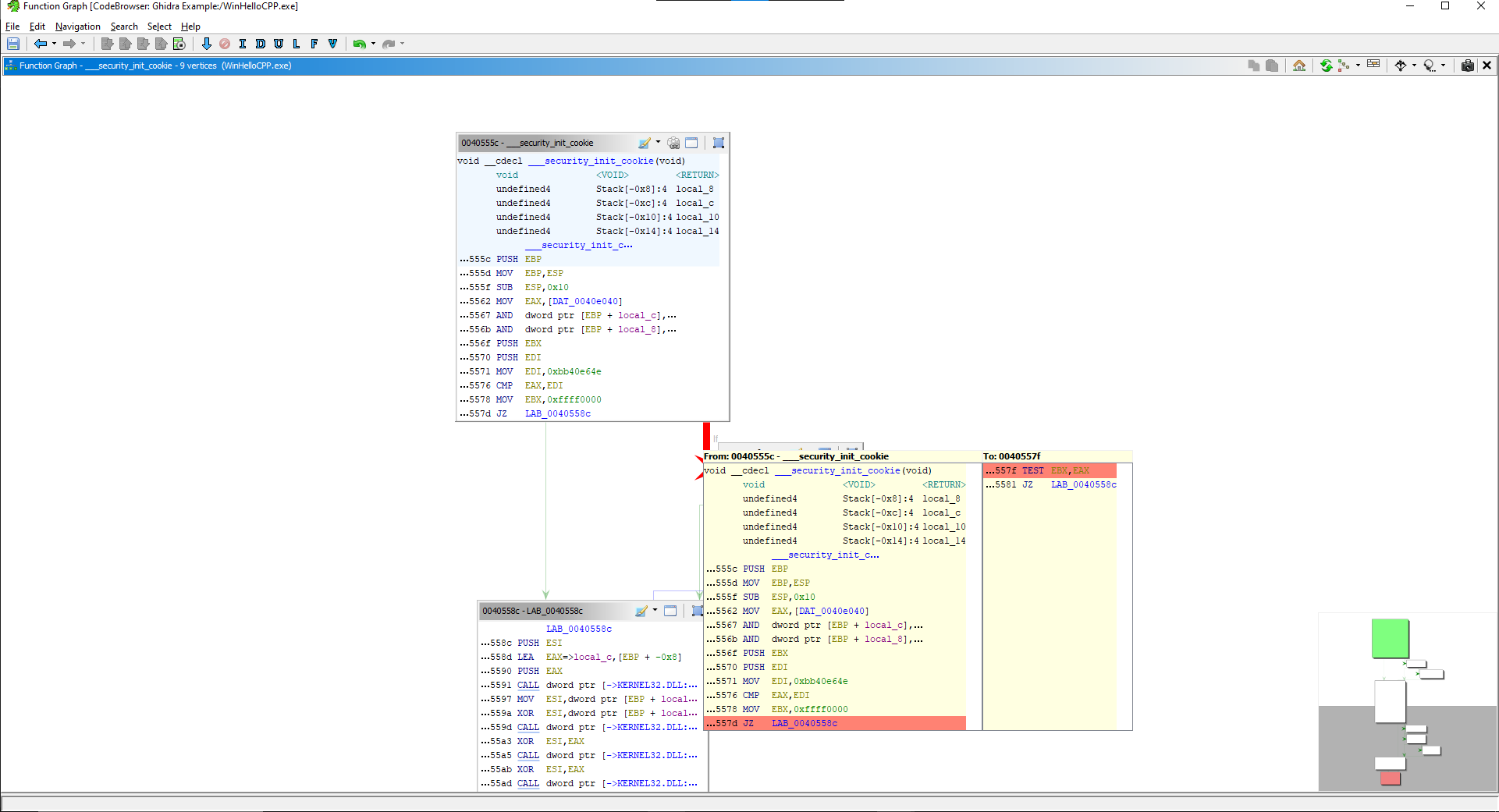
As you can see, this entry calls for a function.

Clicking on this, we're taken to the contents of this secondary function



The lines going out represent subfunctions, such as if, and the pointer returns to the main function once finished.

By hovering over a field, you will see the direction in which these sub functions are performed and by hovering over a line you will see the section being hovered on



### Conclusion

This example is to show an overview of ghidra's code browser, tho this explanation was short, it showcases how powerful this tool is and how versatile it can prove to be

## Chapter-5 Bibliography

* <https://github.com/NationalSecurityAgency/ghidra/wiki/Frequently-asked-questions>
* <https://hex-rays.com/ida-pro>
* <https://letsdefend.io/blog/top-7-reverse-engineering-tools?utm_source=perplexity>
* <https://hackmag.com/security/nsa-ghidra/>
* <https://www.apriorit.com/dev-blog/366-software-reverse-engineering-tools>
* <https://www.retroreversing.com/disassemblers>
* <https://medium.com/@Progsky/ghidra-vs-ida-pro-a-comparison-of-two-popular-reverse-engineering-tools-55223fad9193>
* <https://startup-house.com/glossary/disassembler>
* <https://cybersecuritynews.com/ida-9-0-released/>
* <https://www.linkedin.com/advice/3/what-advantages-disadvantages-using-ghidra>
* <https://www.deepbits.com/deepdi>
* <https://github.com/DeepBitsTechnology/DeepDi>
* <https://www.cs.ucr.edu/~heng/pubs/DeepDi.pdf>
* <https://blog.deepbitstech.com/2020/06/deepdisassembly-blazing-fast-and.html>