

Code and malware analysis with Software Reverse Engineering tools (SRE)

Table des matières

IDA {Pro} – First look	1
Exo1	3
Exo2	3
Exo3	3
Exo4	3
Executables	4
Ghidra	4
Overview	4
First Malwares	11
Symbol Tree	11
String Search	11
OSINT	11
Free Investigations (Bonus)	14

NOTE :

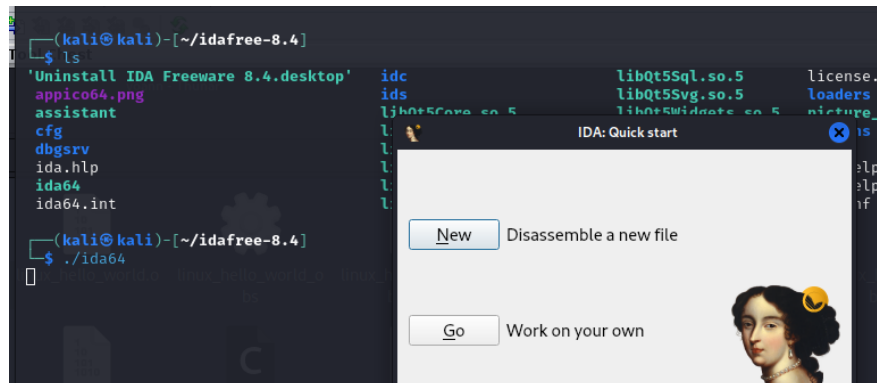
I strongly recommend using a "clean" VM (or at least NOT your preferred VM or main machine) because you can never be too careful. Indeed, we will be handling malicious files, and even though the analysis in this lab will be static, there are malwares that use SRE (Software Reverse Engineering) tools as a trigger/propagation vector (which is quite clever): https://www.cvedetails.com/vulnerability-list/vendor_id-662/product_id-57094/NSA-Ghidra.html

IDA {Pro} – First look

A very famous tool is IDA Pro, which comes in several versions, including a free one that will interest us today. Like Ghidra, it offers decompilation and debugging features and supports a wide range of architectures for reverse engineering. The free version has limited features but is sufficient for hobbying and/or getting familiar with the tool. It's worth noting that there is an intermediate version (Home) that is "affordable" (~€300/year compared to a minimum of €1,800 for the Pro version).

IDA Free is available here (registration required): <https://hex-rays.com/ida-free/#download>

For GNU/Linux, you will need to download a `.run` file. As a reminder, to run a `.run` file, you often need to make it executable (`chmod +x`) and launch it like a bash script (`./filename.run`). This will initiate an installation process that will ask you to accept the terms of use and choose an installation location (by default, in your `/home`). Once the installation is complete, you should be able to launch the program by navigating to the installation directory and running the application.

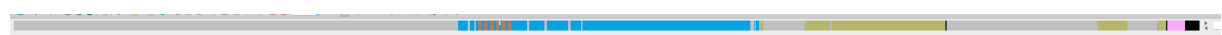


The interface of IDA Pro is composed of several parts, such as the functions window:

Functions		
Function name	Segment	Start
__init_proc	.init	0000000000400480
sub_4004D0	.plt	00000000004004D0
puts	.plt	00000000004004E0
__stack_chk_fail	.plt	00000000004004F0
printf	.plt	0000000000400500
__libc_start_main	.plt	0000000000400510
strcmp	.plt	0000000000400520
gmon_start__	.plt	0000000000400530
start	.text	0000000000400540
deregister_tm_clones	.text	0000000000400570
register_tm_clones	.text	00000000004005A0
_do_global_ctors_aux	.text	00000000004005E0
frame_dummy	.text	0000000000400600
get_pwd	.text	000000000040062D
compare_pwd	.text	000000000040067A
main	.text	0000000000400716
__libc_csu_init	.text	0000000000400760
__libc_csu_fini	.text	00000000004007D0
_term_proc	.fini	00000000004007D4
puts	extern	0000000000601068
__stack_chk_fail	extern	0000000000601070
printf	extern	0000000000601078
__libc_start_main	extern	0000000000601080
strcmp	extern	0000000000601088
gmon_start__	extern	0000000000601090

The same goes for the different representations of the code, including its HEX version (Hex View tab), a graphical representation of the code (IDA View), and various options accessible through the top menu bar.

A key difference compared to Ghidra (another famous tool that we will see later) is the presence of a graphical representation of the analyzed code, which is particularly useful for better orientation.



Take a few minutes to investigate the different colors (particularly the functions displayed in green).

Your objective will be to analyze files (in this case, `.exe` files) and locate FLAGS. A FLAG is a string of characters that demonstrates the successful completion of a cybersecurity challenge.

Your goal is to retrieve the flags from the binaries. You'll notice that the format of the flags is quite... explicit.

NOTE: For the initial levels, it is possible (and even recommended) not to use only IDA. You can also use GNU/Linux commands, programs covered in other courses (especially for dynamic analysis), or even tools/scripts of your choice.

For each exercise, take detailed notes and document your investigation methodology thoroughly.

Exo1

Let's start simple... Try running the program in a terminal to see what it does.

Exo2

Once the program is running, try interacting with it. Then ask yourself the following question: How does the program work algorithmically? How would you implement it? Once you've reflected on this, try finding a command, tool, or method to extract the information.

How would you make this program more robust? (Speculative answers are fine; no need to implement your ideas.)

Exo3

RSE tools are our friends! Attempt to retrieve the password using your reverse engineering tools.

Note: It's certainly possible to obtain this information using other tools or commands (including those you may have used in other courses). Additionally, if you encounter issues installing IDA and prefer not to use Ghidra, you can opt for **Radare2** (the r2 command is already available in Kali Linux).

How would you make this program more robust?

- Consider potential vulnerabilities or weaknesses in the program's structure or logic.
- Propose theoretical changes to improve security (e.g., stronger encryption, better obfuscation, or additional checks to prevent reverse engineering).
- No need to implement these ideas—focus on identifying improvements.

Exo4

This one is a variation of Exercise 3, but this time the string you need to find (also a password) is harder to access. Run the program to learn more about it. Again, use IDA Pro to attempt retrieving the information. **Note:** Here too, other tools may allow faster results, but since our goal is to better understand RSE, take your time with it.

Executables

Analyze the executables “Executable1” and “Executable2” to try and retrieve the flags.

Ghidra

Ghidra (<https://ghidra-sre.org/>) is developed by the NSA (National Security Agency). It is an open-source software tool compatible with many processor architectures and operating systems, designed for analyzing {malicious} programs.

To install it on GNU/Linux (Debian-based distributions), the process is straightforward (though it might take some time depending on the server speed):

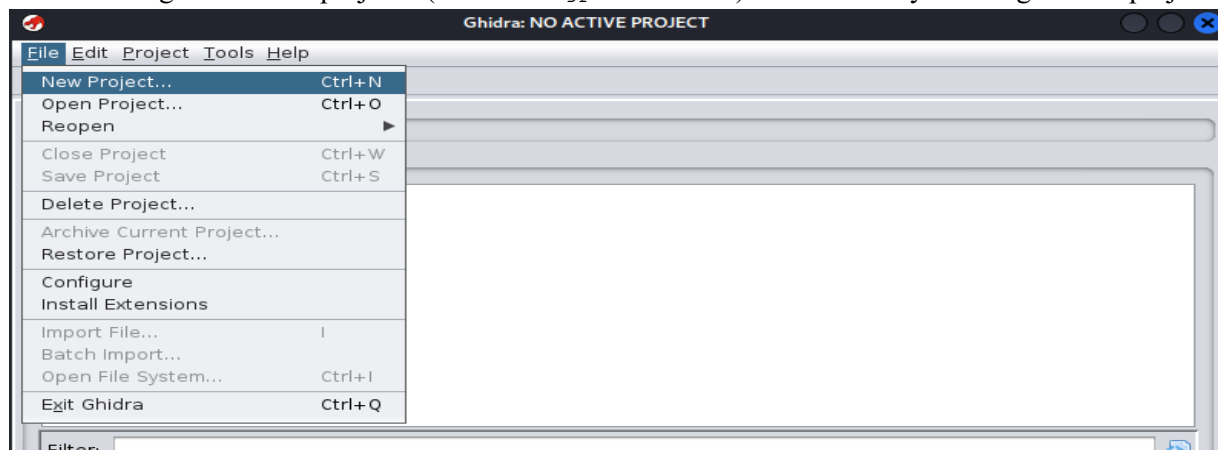
```
(kali@kali)-[~]
$ sudo apt install ghidra
[sudo] password for kali:
Reading package lists... Done
Building dependency tree... Done
```

For other operating systems, you can refer to the well-documented installation guide here: https://htmlpreview.github.io/?https://github.com/NationalSecurityAgency/ghidra/blob/Ghidra_11.0.1_build/GhidraDocs/InstallationGuide.html.

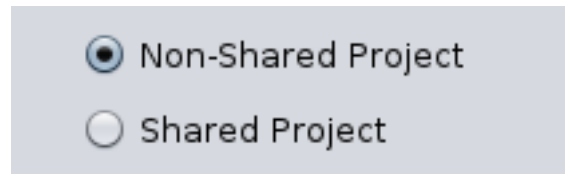
Once installed and launched, the application might throw an `IllegalArgumentException`. This happened in my case (Kali 6.5.0 vanilla), but it had no impact on usability.

Overview

Ghidra is organized into projects (with the `.gpr` extension). Let's start by creating a new project:



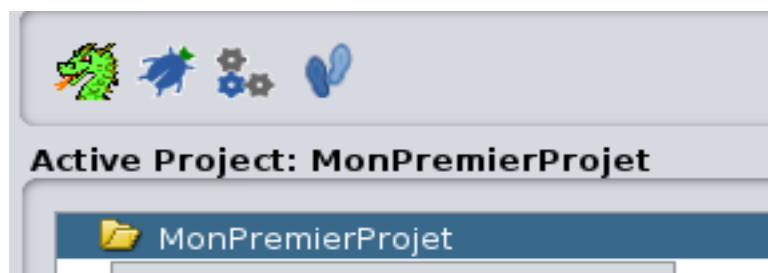
No need to create a shared project in our case; a non-shared project will suffice.



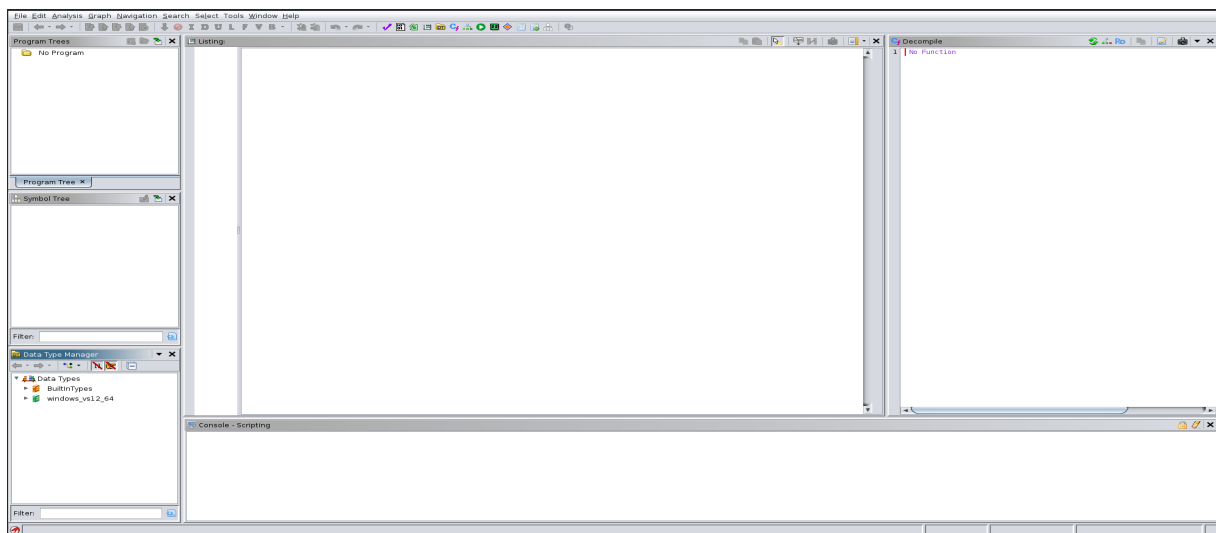
Once the other usual details are filled in, you can create the project.



Once the active project is created, you can interact with it in several ways using the four icons:



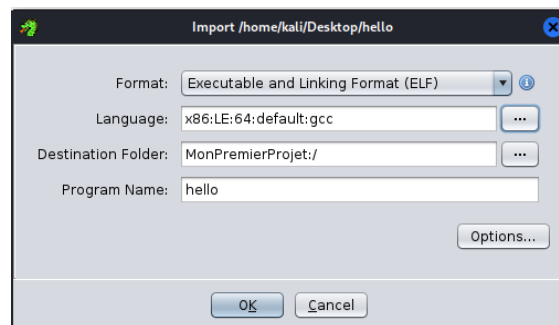
We will click on the green dragon icon to launch the code explorer. After a brief animation, you will arrive at the following interface:



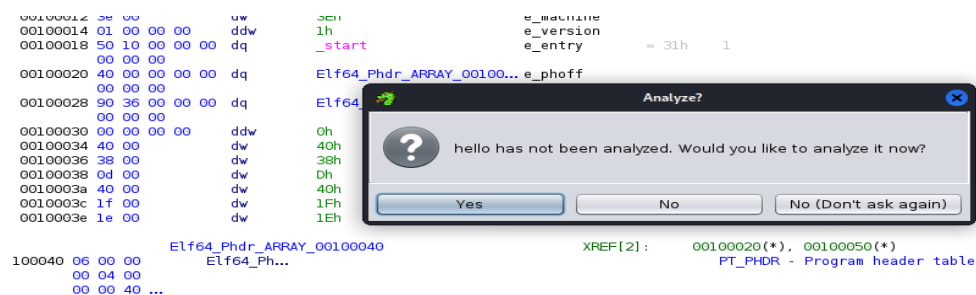
Our project is currently empty, so we'll need to feed it some code. To do this, press the **"i"** key (or go to **File > Import File**).

For the code, we'll use a traditional `HelloWorld.c` program, which we'll compile first. If you don't have the file, take a few minutes to recreate it. Once imported, Ghidra will present you with some options. It's generally best to leave these as default, but you can manually adjust the

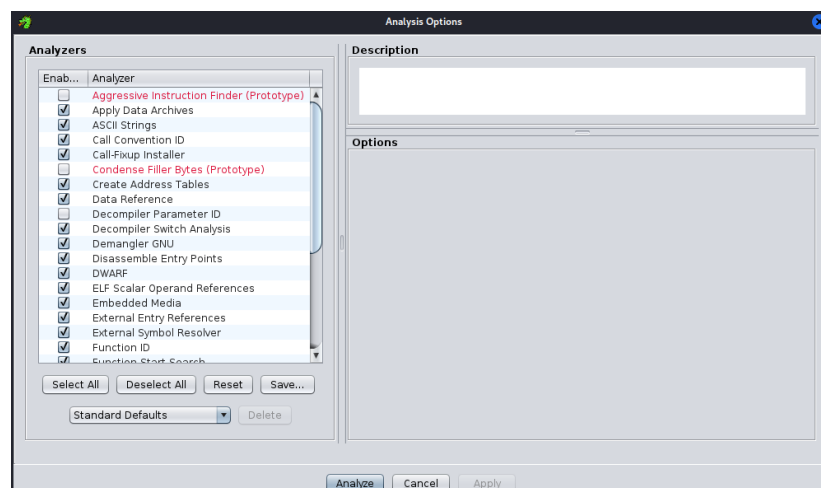
language and format if necessary (for instance, if Ghidra detects the wrong extension or language).



Once the **OK** button is pressed, Ghidra will ask if you want to analyze your executable. Here again, it is always a good idea to accept the suggestion.

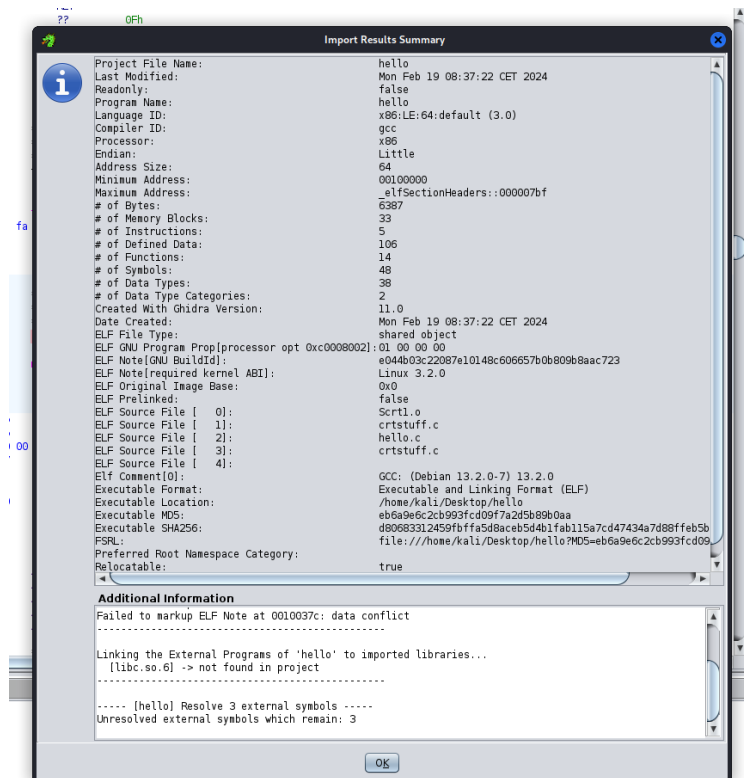


Ghidra provides analysis options, with the options marked in red being beta features (prototypes) that may be unstable or produce inconsistent results. Here again, we will stick with the default options.

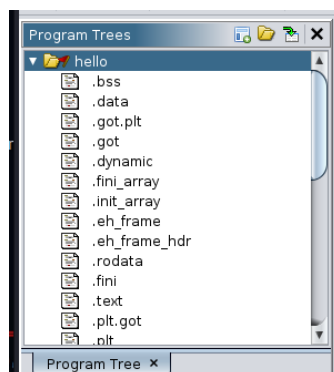


Once the analysis is complete (usually very quick with our small helloworld program), Ghidra provides a summary with interesting information, such as:

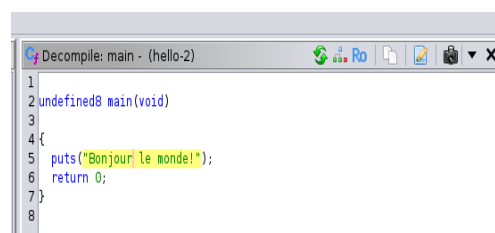
- **Metrics** (size, number of instructions, number of symbols, etc.)
- **Hashes** (MD5 and SHA256)
- **Metadata** (date, created files, CPU type, compiler, etc.)



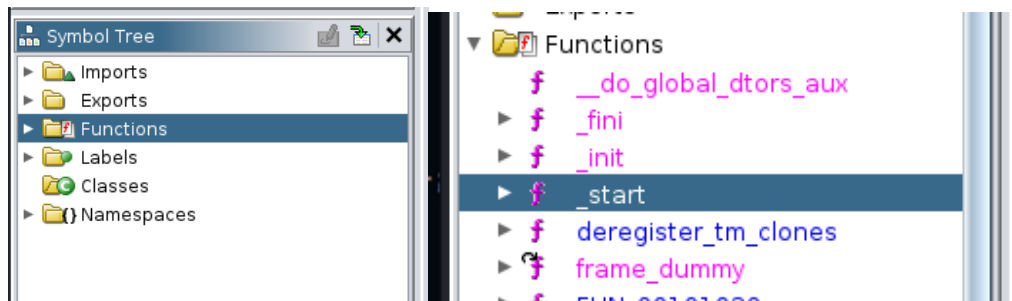
Once the summary popup is closed, you will see that information has appeared in our project. For example, in the top left corner (Program Tree), you can see the structure of assembly (e.g., .data, .text, .bss, etc.). By double-clicking on an element, you can view its details in the main window on the right:



The right-hand section (**Decompile**) provides a view of the decompiled code. This view is particularly useful as it allows you to see the program from a higher-level perspective:

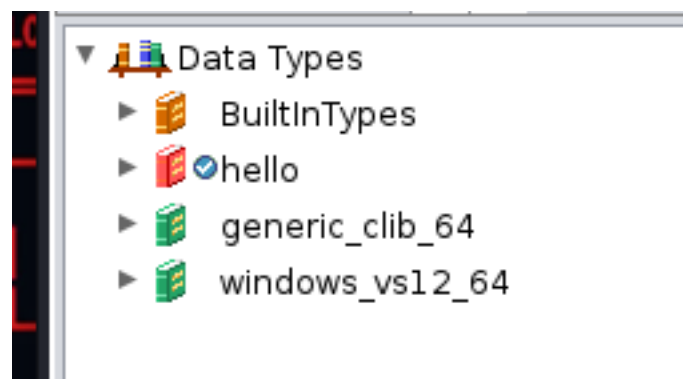


Another interesting view is the **Symbol Tree**, which organizes the program by semantic elements such as functions or classes (in the sense of Object-Oriented Programming). Here, you'll also find previously discussed concepts, such as the famous `_start`, which, when clicked, allows you to navigate directly to the corresponding part of the code:

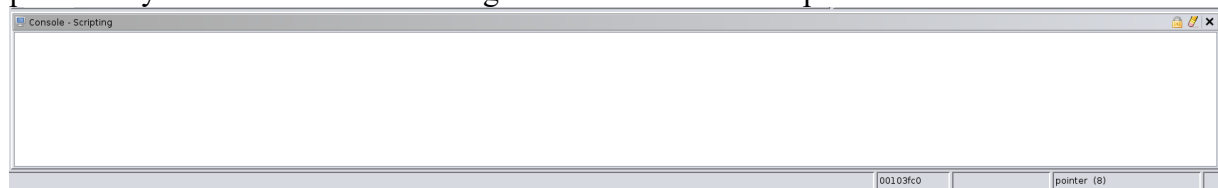


Below this section, you'll find the **Data view**, which, as the name suggests, allows you to define, manage, and apply various data structures—ranging from simple to complex—within your project. The structure is generally organized as follows:

- **Orange** (builtin): Contains all standard data types (e.g., `char`, `int`, `byte`, etc.).
- **Red**: Represents data types identified during the analysis of your project.
- **Green**: Represents additional data types loaded by Ghidra (e.g., archives/libraries from Visual Studio 12, as shown in the accompanying screenshot).

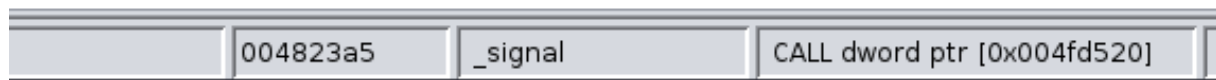


At the bottom of the application, you'll find the **console**, which displays relevant information, particularly during script execution.

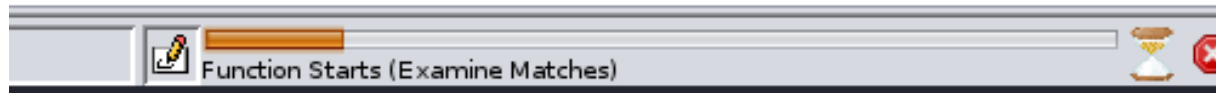


This part of the screen also helps you navigate by displaying the memory address (e.g., `0048...`), the function (e.g., `_signal`), and the instruction (e.g., `CALL dword...`) where your cursor is currently

pointed.



Additionally, the console allows you to monitor ongoing processes, which is useful for ensuring that your analysis hasn't crashed or stalled.



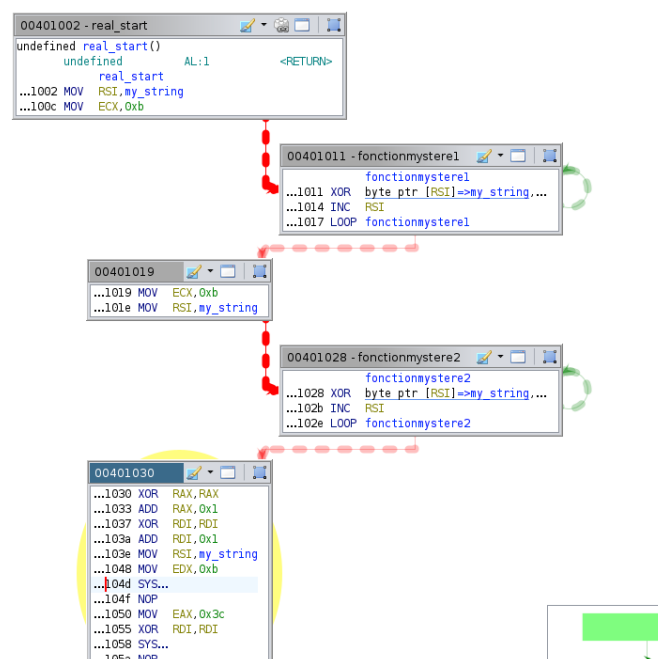
Finally, the toolbar allows you to perform various operations such as searching, navigating, manipulating, analyzing, and displaying data.



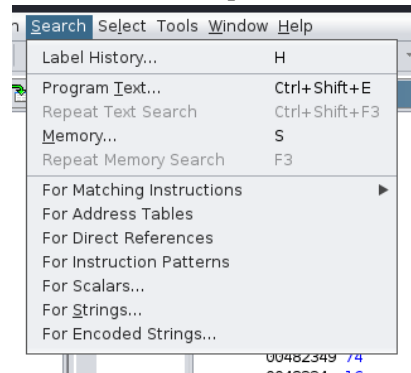
For example, it can be useful to visually represent a program's execution during analysis. To do this, you can use the **function graph** icon (generally, you need to highlight the code by selecting it for the graph to appear).



Here is an example of a “mystery program” used in another lab, where you can see the “mystery functions” made up of loops:

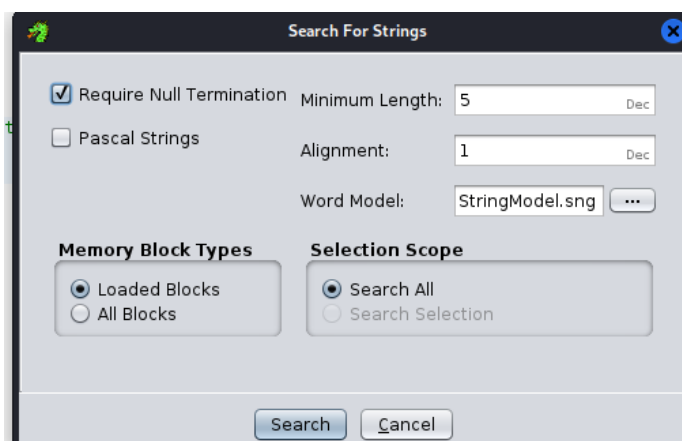


Another interesting feature of Ghidra is its **search functionality**! You can perform several types of searches through the dedicated menu at the top:



1. **Address Table Search:** Allows you to search for address tables to find pointers or addresses in the code.
2. **Direct Reference Search:** Enables you to locate all direct references to a specific object, such as a function or a variable, to see where and how an element is used in the code.
3. **Instruction Patterns Search:** Helps identify specific instruction sequences in the code, useful for recognizing code sections that perform particular operations or known algorithmic constructs.
4. **Scalar Search:** Allows you to search for specific scalar values in the code, such as hardcoded numbers.
5. **Strings Search:** Facilitates searching for literal strings (e.g., URLs, common names, etc.).

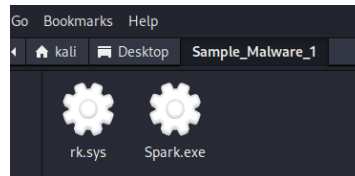
For most of these features, you can refine your searches. For example, in **Strings Search**, you can select word models. Ghidra offers a default model (`StringModel.sng`), which can be customized or enriched according to your needs:



First Malwares

Now that we have familiarized ourselves with Ghidra's main interfaces, we will open a malicious file.

Our small malware is available in an archive that I will provide during the course (feel free to ask if needed). It consists of two files:



The objective will be to use Ghidra to learn more about these files.

To do this, start by creating a new project (you can name it anything you like, such as `first_malware`) and import the `.exe` file into it. Use Ghidra's automatic analysis (you can leave the default options). Since the program is more complex than our "Hello World" example (and possibly more complex than other applications you've tested yourself), the analysis might take a bit longer. However, this is a "necessary evil" to save time later. Remember, the progress bar in the bottom right corner indicates that processing is ongoing.

Symbol Tree

Once the analysis is complete, a good first step is to examine the **Symbol Tree**. As mentioned earlier, this provides information about the functions used, particularly the imported functions. These often provide clues about the program's capabilities or functions (e.g., reading buffers, saving files, creating sockets, using web protocols, starting services, etc.).

Look at the different DLLs used and, based on the method names, infer what the program is capable of doing. Justify your assumptions.

String Search

Use the **Search for Strings** feature to look for various pieces of information, such as:

- Programs that might interact with the malware
- Usernames or machine names
- Interesting information
- Etc.

Again, take notes and justify your assumptions.

OSINT

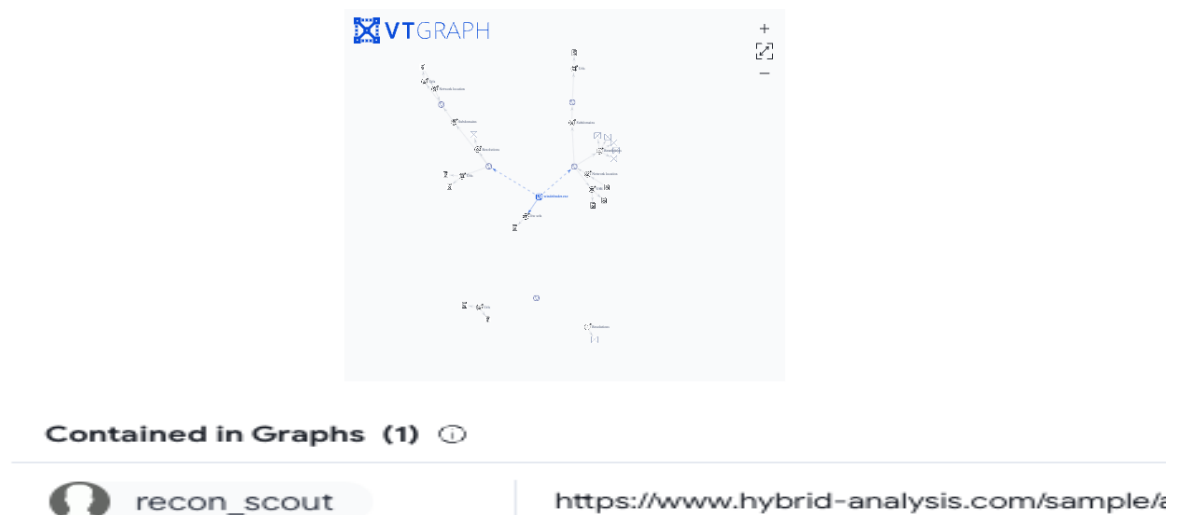
OSINT (Open Source Intelligence) is a technique used to gather information about a target (e.g., an IP address, a person, a machine) from publicly accessible sources (e.g., the Internet, public databases, metadata). OSINT is often used during the reconnaissance phase of a

penetration test or an actual attack. Its advantage is being indirect—it doesn't require direct interaction with the target.

Reverse engineering can use OSINT to corroborate a hypothesis. For example, try searching for common web extensions (.fr, .net, etc.) in the strings. If you find interesting items, visit **VirusTotal** and use the **URL** tab to gather more information!

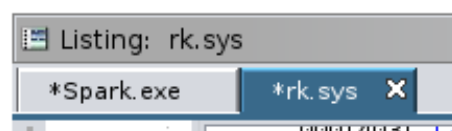


Specifically, you might find in the comments a link to **Hybrid Analysis** and a **VirusTotal graph**, which could help you gather even more information.



Remember to take detailed notes to support your report.

Now, let's examine the second file (`rk.sys`). Just like with the previous file, import it into your project and let Ghidra analyze it. Once the analysis is complete, you should have both files in your project:



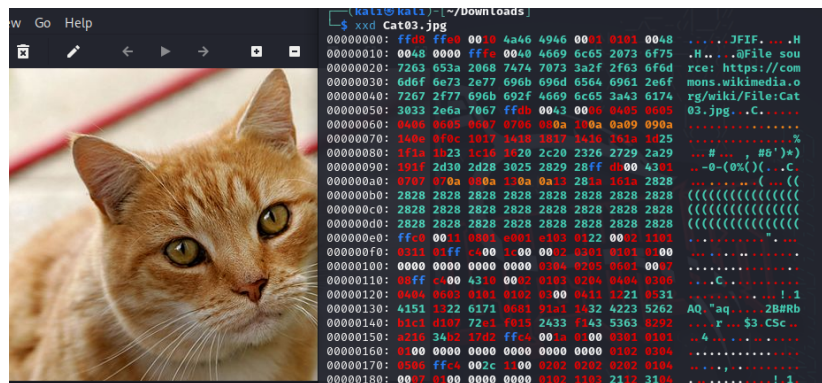
Open the window that allows you to view the file in hexadecimal format:



At the bottom of this window, you should see the starting and ending addresses of this file. What are these addresses?

In a hexadecimal file, you can navigate using the headers and footers, known as **magic numbers**. These are specific signatures associated with common file types (e.g., .jpg, .doc, etc.). For example, by opening an image with `xxd` (a CLI hex viewer/editor), you can see that the file begins with `ffd8`, the magic number for the JPEG format.

If this image is embedded within a larger file, you could locate it using this header, even if the image is "buried" within other data.

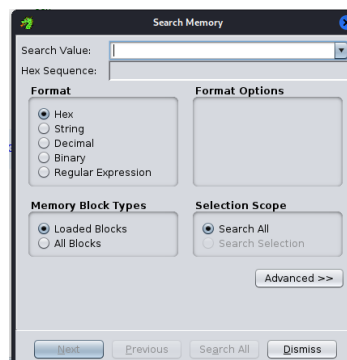


You can easily find the magic numbers for well-known file formats, for instance, on this page: [List of File Signatures - Wikipedia](#).

With this information, investigate the header of the file `rk.sys` to determine its format.

Once identified, check the file `spark.exe` for occurrences of this magic number. To do so, use the **Search Memory** tool and click on **Search All**.

Document any occurrences you find and try to understand what these occurrences represent.



As with the previous exercise, feel free to explore other Ghidra features to learn more about the program.

Free Investigations (Bonus)

If you have extra time, you can visit the following links to download and investigate malware samples of your choice (be careful, don't download it with your main OS) :

- **MalwareBazaar:** <https://bazaar.abuse.ch/browse/>
- **TheZoo:** <https://github.com/ytisf/theZoo>
- **VX-Underground:** <https://vx-underground.org/Samples>

You can also try other SRE tools such as:

- **Radare2:** <https://github.com/radareorg/radare2>
- **Cutter:** <https://cutter.re/>
- **Any Run** (online platform): <https://app.any.run/>
- **Relyze :** <https://www.relyze.com/index.html>