SEMINAR 4
INTRODUCTION TO REVERSING

A primer on analyzing programs

José Manuel Redondo López *"S-81 Isaac Peral"* Project v3.0

Departamento de Informática
Universidad de Oviedo

Universidad de Oviedo

Logo: @creative_vanesa (*Instagram*)

INDEX

< Go to Index

Source: zona-militar.com

Introduction

What is Ghidra?

Introduction

José Manuel
Redondo López

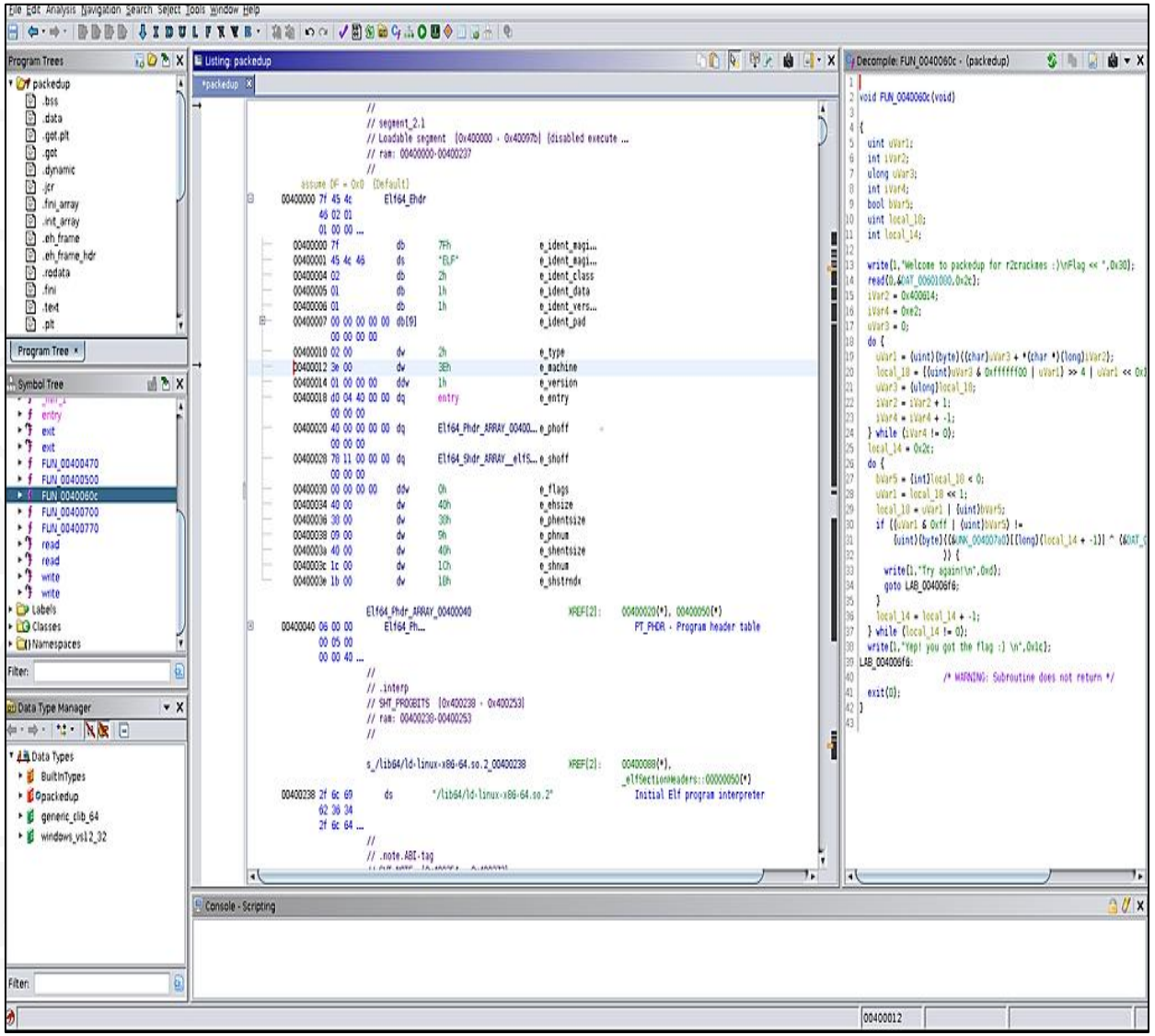
- **This seminar describes an approach for using Ghidra to perform malicious code analysis**
  - It is just a brief introduction of common analysis operations
  - This field is much much more complex! ☺
- **Ghidra is a free software reverse engineering (SRE) framework**
  - Developed by the National Security Agency (NSA) of the United States
  - It is powerful reverse engineering tool available to all for free
  - Allows to analyze malware to better understand it
    - This helps to locate vulnerabilities in systems and networks
- **Since its release, Ghidra has attracted a growing community of contributors**

GHIDRA

https://ghidra-sre.org/

José Manuel
Redondo López

- **Free alternative to the popular IDA Pro**
- **Main features**
  - Analysis of compiled code in Windows, Mac OS, and Linux
  - Disassembly, assembly, decompile and display code
  - Do/Undo actions
  - Large set of CPU instructions and executable formats supported
  - Supports third-party plugins or scripts in Java o Python (Jython) using its API



Source: https://en.wikipedia.org/wiki/Ghidra

< Go to Index

Source: zona-militar.com

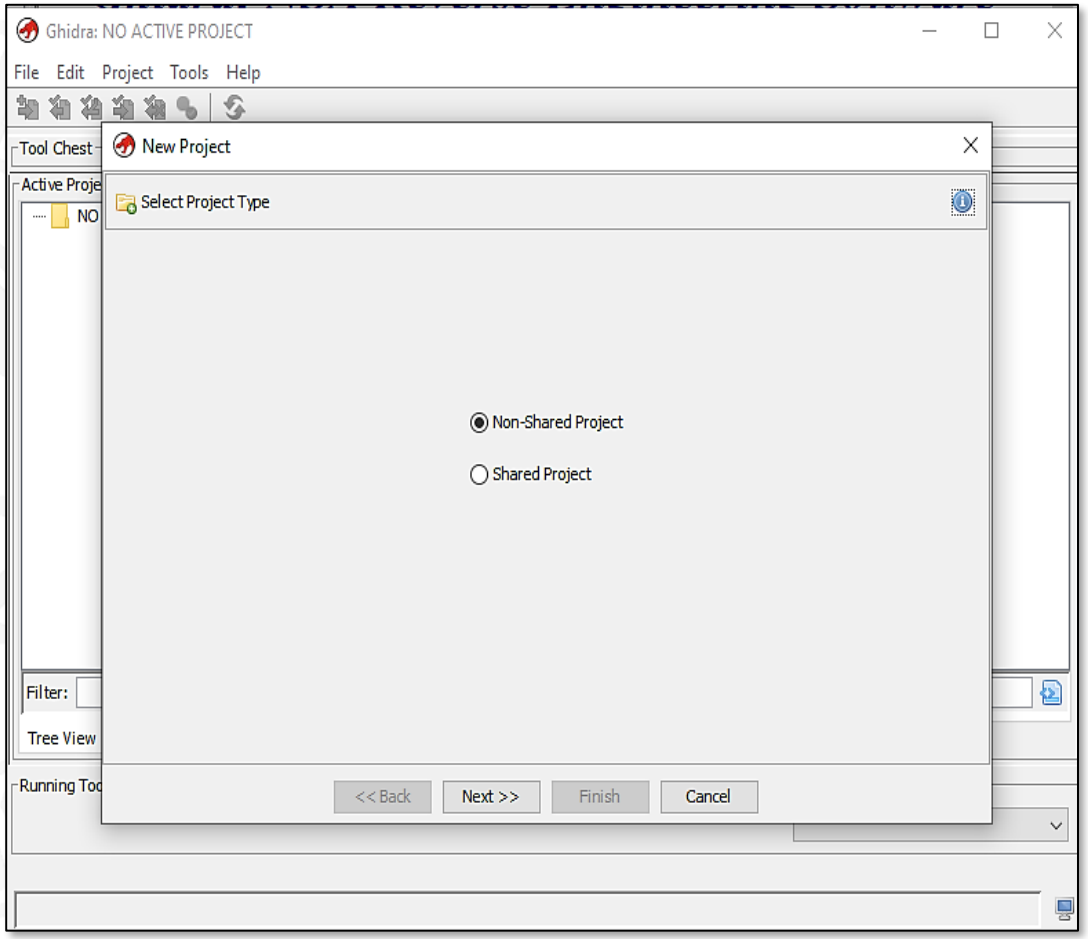
Launching Ghidra & loading an executable

Setting up the analysis environment

LOADING AN EXECUTABLE

José Manuel
Redondo López

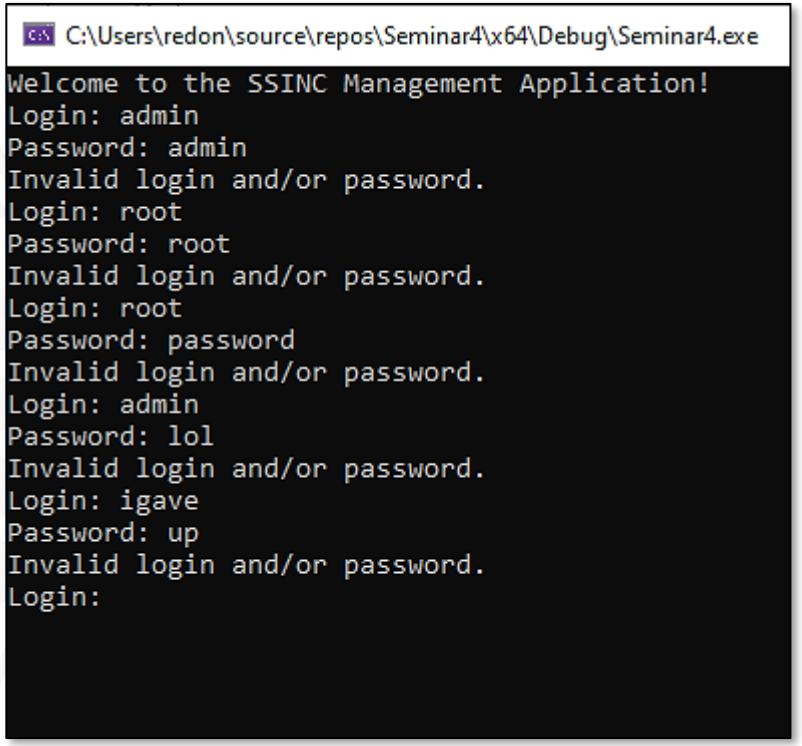
- **Installing Ghidra requires**
  - Downloading and extracting a zip file and launch the batch file
  - Installing **JDK/OpenJDK 11+**
    - https://adoptopenjdk.net/index.html?variant=openjdk11&jvmVariant=hotspot
- **Then, create a project (File - New Project)**
  - Reverse engineering often target multi-component malware, so it uses projects instead of individual files
  - Project options include
    - "Non-Shared" for single-user analysis
    - "Shared" for collaborative work (multiple users can access a project repository on a server)
  - **We will focus on single user projects**, but the other type is good for sharing reversing efforts
  - Projects need a name



Our executable

José Manuel
Redondo López

- **We are going to reverse-engineer a custom executable we provide**
  - It is a fake command-line management application, created with Visual Studio C++
    - No GUI to keep code size small
  - Compiled in debug mode
  - It contains a login-password initial step
  - If successfully logged in, it shows a menu
  - All options are “under construction” ☺
  - But, upon successful login, the program steals user clipboard data periodically!
    - Don’t worry, it only displays it ☺
    - But it will be very easy to store it, send it to another system…
  - **Copy the `.exe` and `.pdb` files to the same folder**

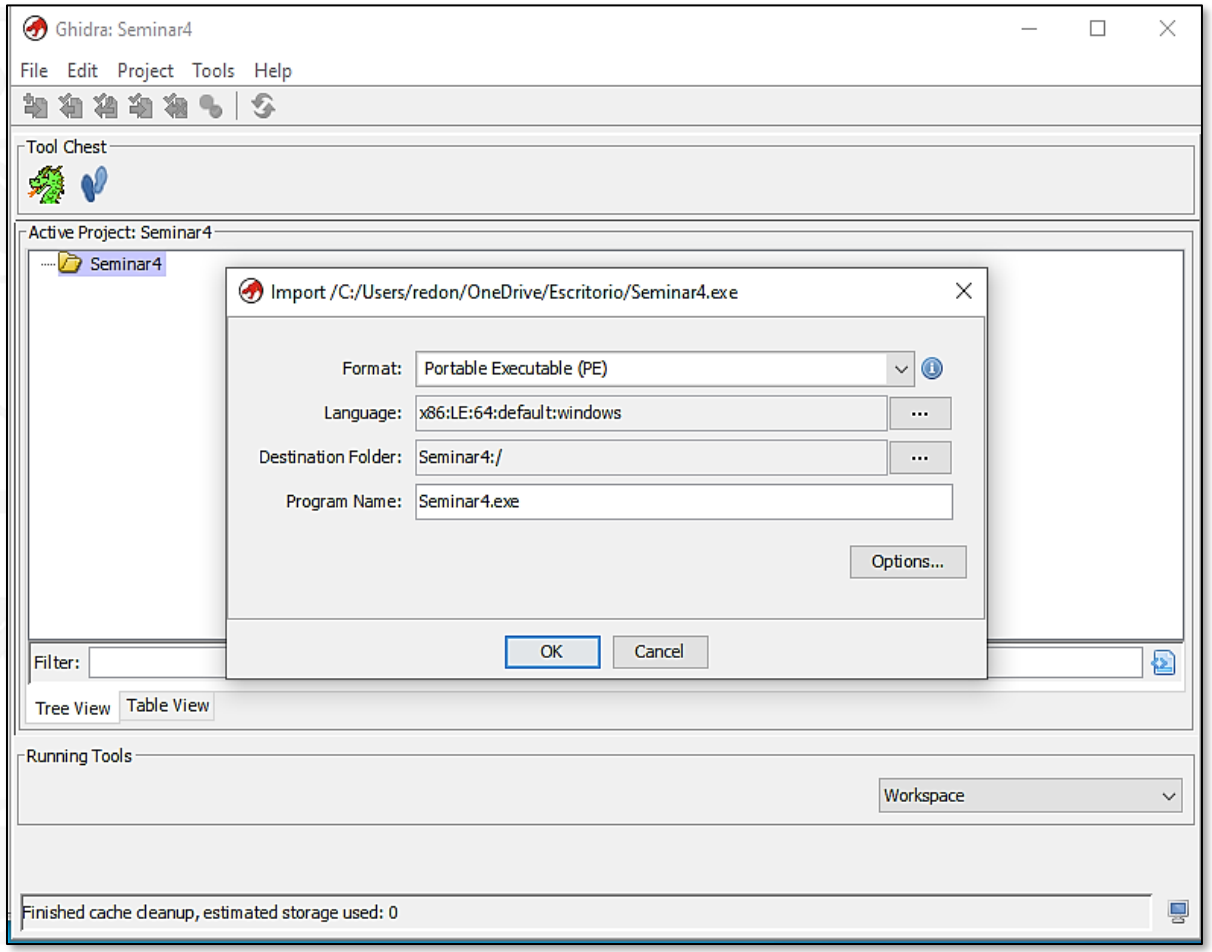


The Visual Studio Project source code is available in the virtual campus!

LOADING AN EXECUTABLE

José Manuel
Redondo López

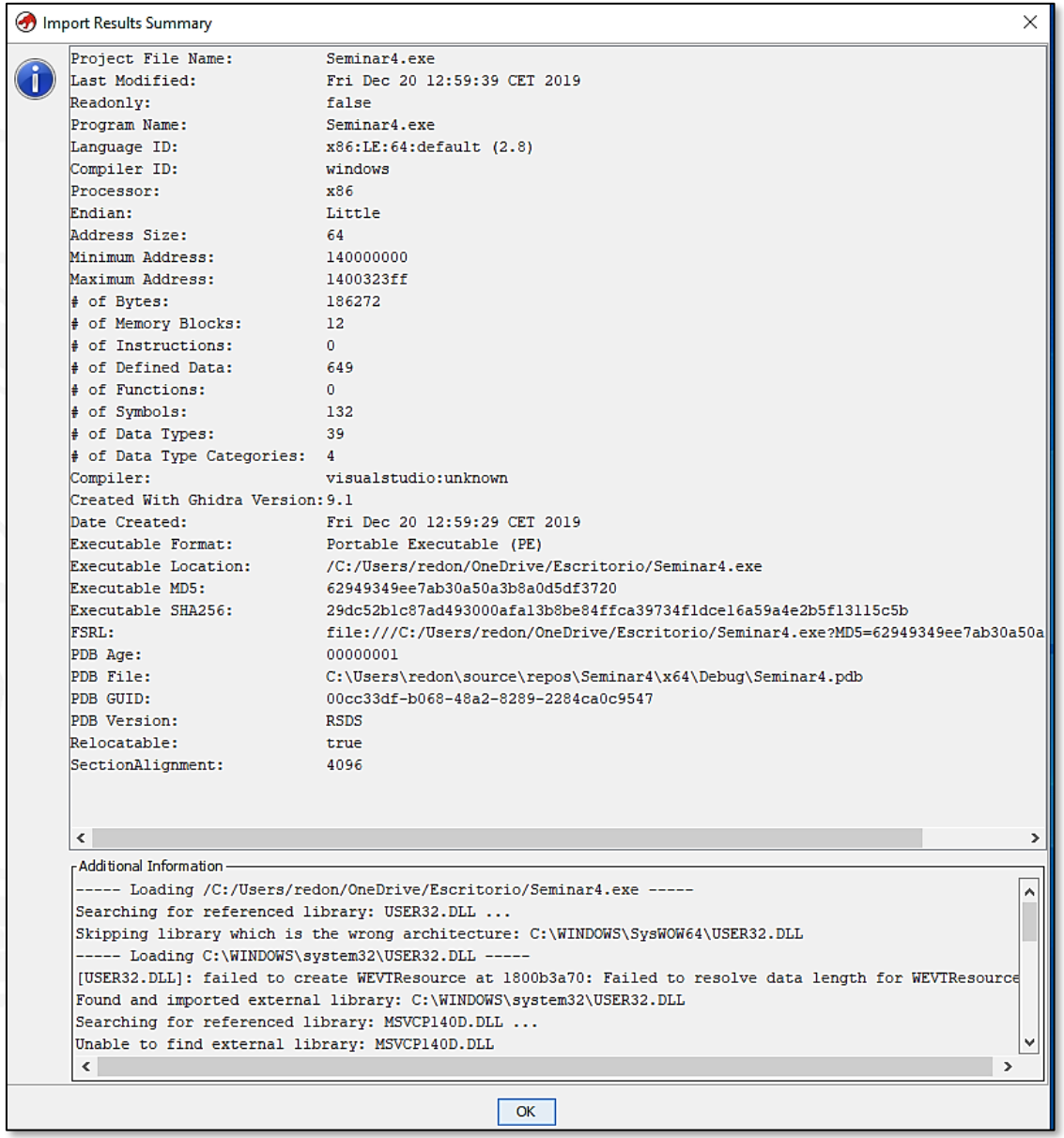
- To analyze an executable file, you need to drag and drop the `.exe` file into the project window
- This will launch a dialog box where you can accept the defaults and click OK
- If in the same folder a `.pdb` file exists, Ghidra will attempt to use it
  - These files contain the debugging symbols
  - They could not work on some configurations (we will see this later)



LOADED EXECUTABLE INFORMATION

José Manuel
Redondo López

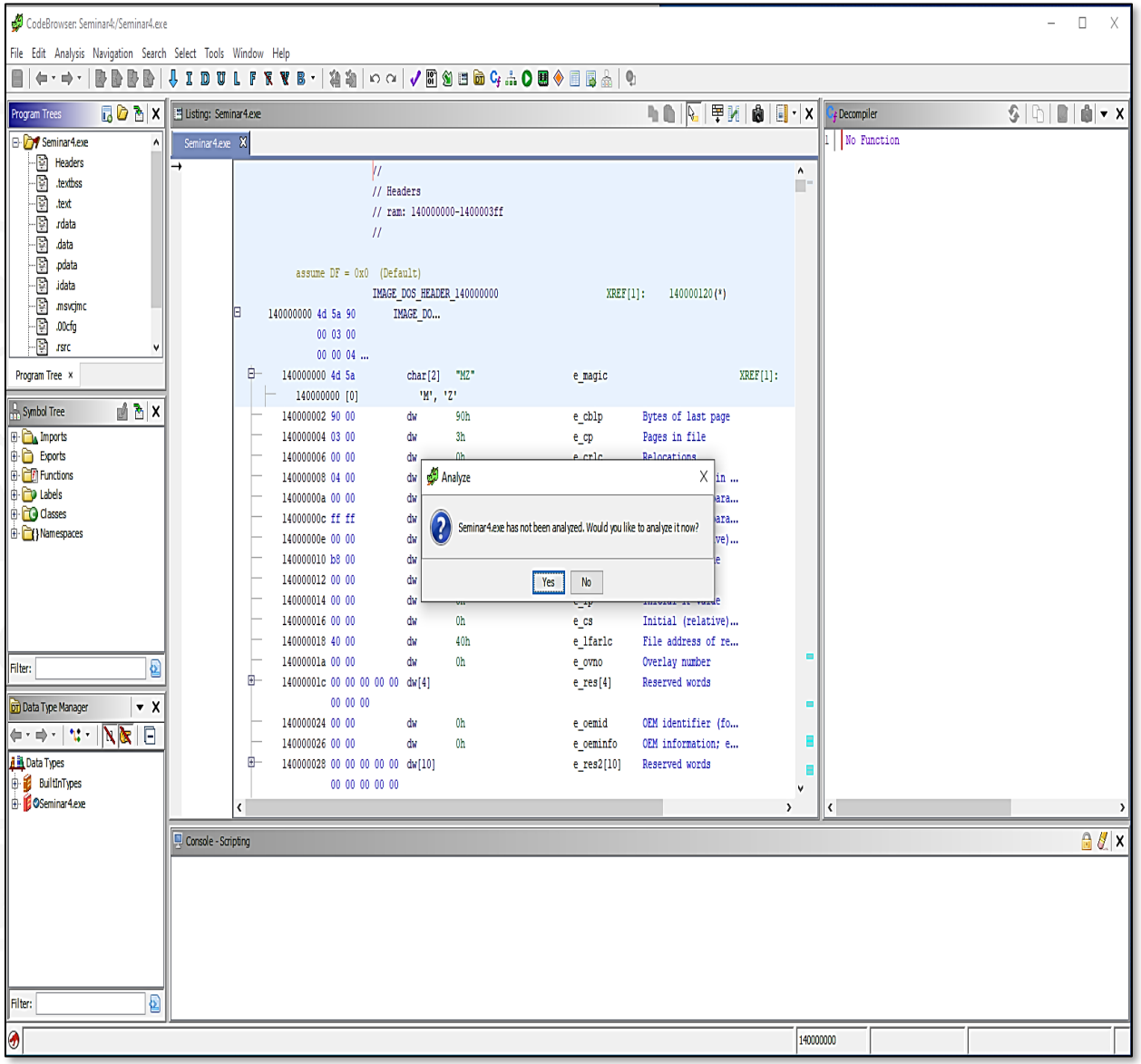
- **When a file is added to a Ghidra project for the first time, a special information window appear**
- **This window contains a lot of useful information about the file, such as**
  - Various sizes
  - Compiler used to create it
  - Timestamps
  - Exe hashes
  - Architecture



ANALYZING AN EXECUTABLE

José Manuel
Redondo López

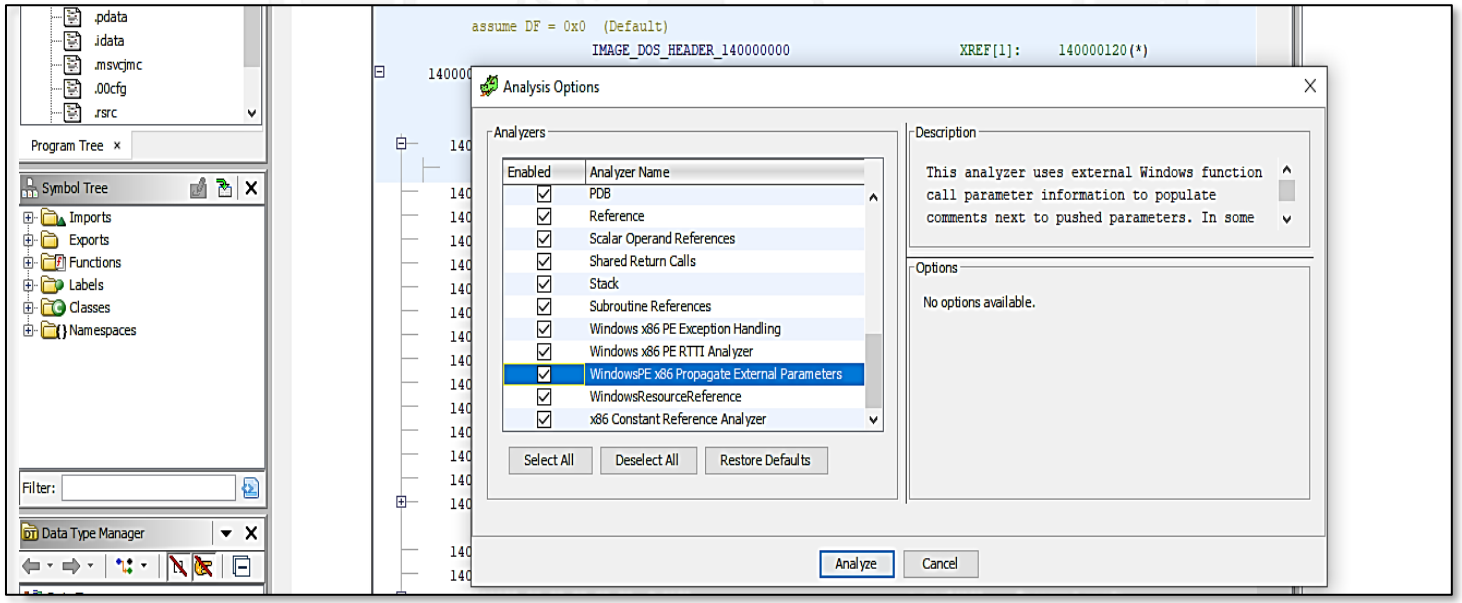
- **Double-clicking the imported file in the project window runs the Code Browser**
  - If not done previously, the tool will ask to analyze the file now
  - **This is required** to continue the reversing process
  - Analysis can be done / redone manually in the "Analysis" menu
- **Enable the option "WindowsPE x86 Propagate External Parameters"**
  - It populates function arguments in the comments
  - This is convenient for better understand the analyzed code



ANALYZING AN EXECUTABLE

José Manuel
Redondo López

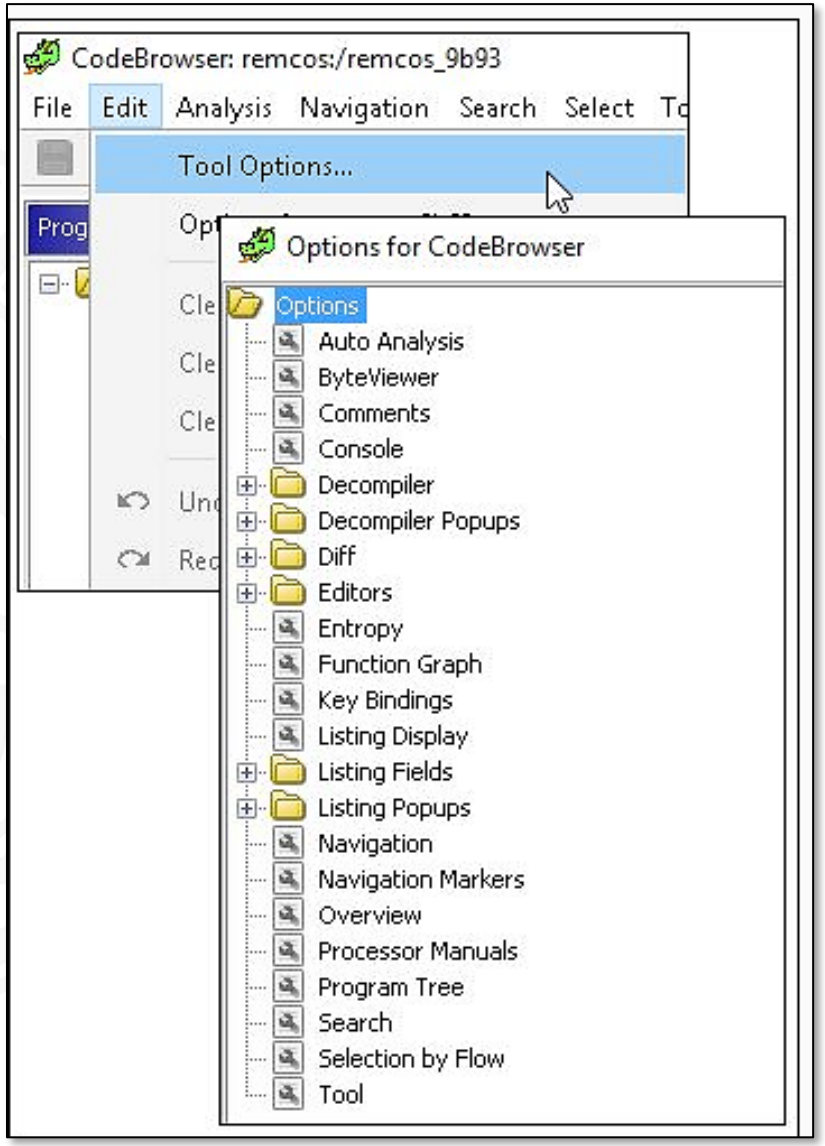
- **To start the analysis, let Ghidra collect information about our `.exe`**
  - As we are in Windows and a `.pdb` is provided, Ghidra should load it and complete the exe information
  - However, it could complain about not finding **the DIA SDK** that is used to fully load `.pdb` files
    - This SDK is installed with Visual Studio (even free versions, like Visual Studio Community)
    - In theory, you could search for the file `msdia140.dll` in the hard disk and register it with `regsvr32` from a terminal with `Administrator` privileges
    - **But this does not work in some cases**: we cannot see extended information (like function names)
    - We will proceed anyway, as having a `.pdb` file is not a common case in real deployments



MODIFY DISPLAYED ELEMENTS

José Manuel
Redondo López

- **It is useful to make some changes to the Listing window via Edit - Tool Options for easier reading**
  - **Listing Display**: Increase the font size and enable bold formatting
  - **Listing Fields: Bytes Field**: Change "Maximum Lines to Display" to 1
  - **Listing Fields: Cursor Text Highlight**: Change "Mouse Button to Activate" to LEFT
  - **Listing Fields: EOL Comments Field**: Check "Show Semicolon at Start of Each Line"
  - **Listing Fields: Operands Field**: Check "Add Space After Separator"



< Go to Index

Source: zona-militar.com

INSPECTING EXECUTABLE DATA

Analyzing strings withing the executable

Investigating string references

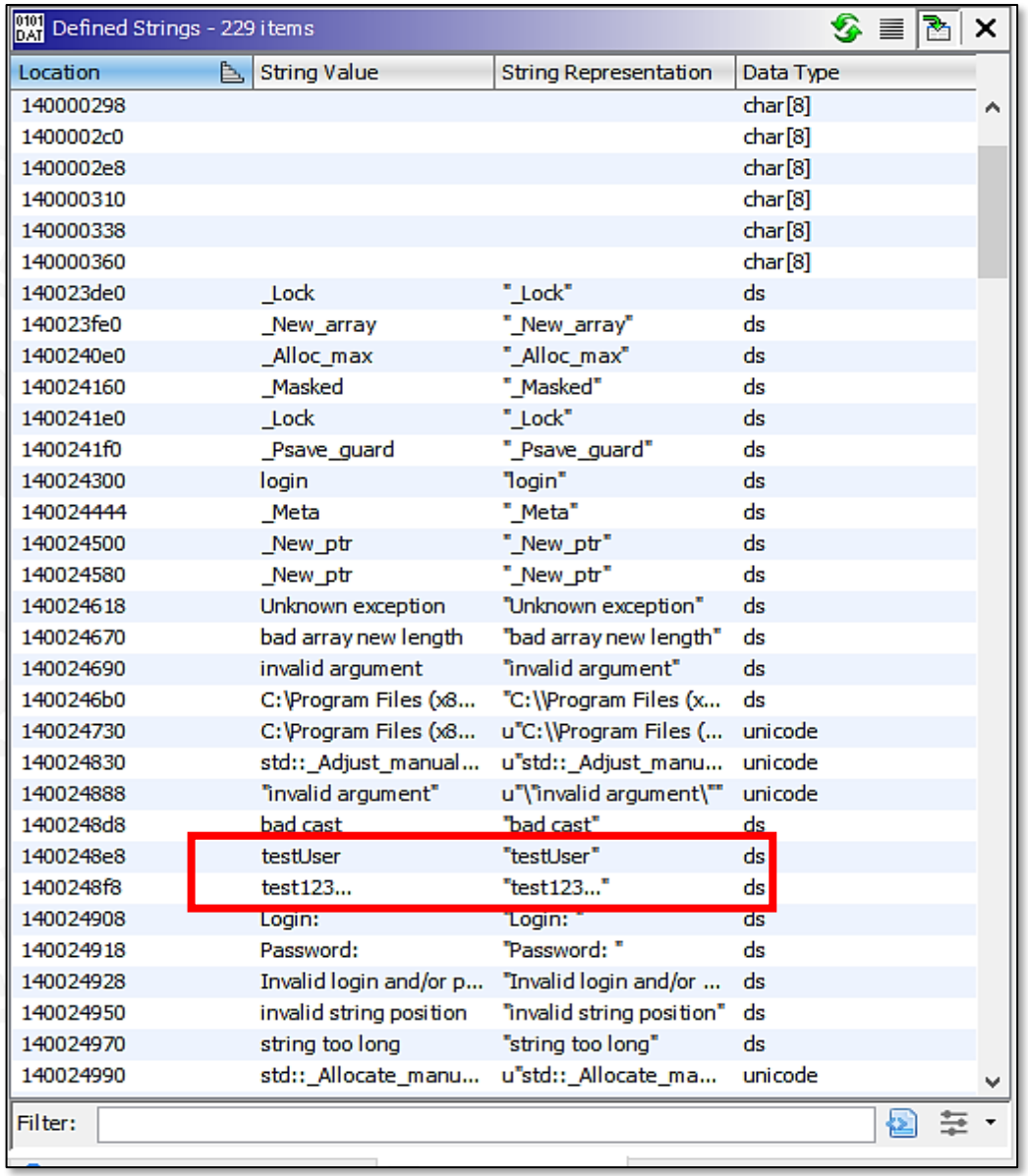
José Manuel
Redondo López

- **One of the things we could not be aware of is that hardcoded strings in our executable are viewable with tools like Ghidra**
- **This means that creating constants with passwords, important IDs, usernames, passwords, etc. (aka "hardcoding") is always a bad idea**
  - Strings with important data must be encrypted
  - Or, much better, loaded from external encrypted sources
- **Data declared within the source code can be located and used against us**
  - E.g.: we have no clue about how to bypass an application authentication mechanism
  - What if the application code have some kind of "hidden" authentication information? (backdoor?)
- **For just viewing hardcoded strings inside an executable, the command-line tool `strings` is recommended**

INVESTIGATING STRING REFERENCES

José Manuel
Redondo López

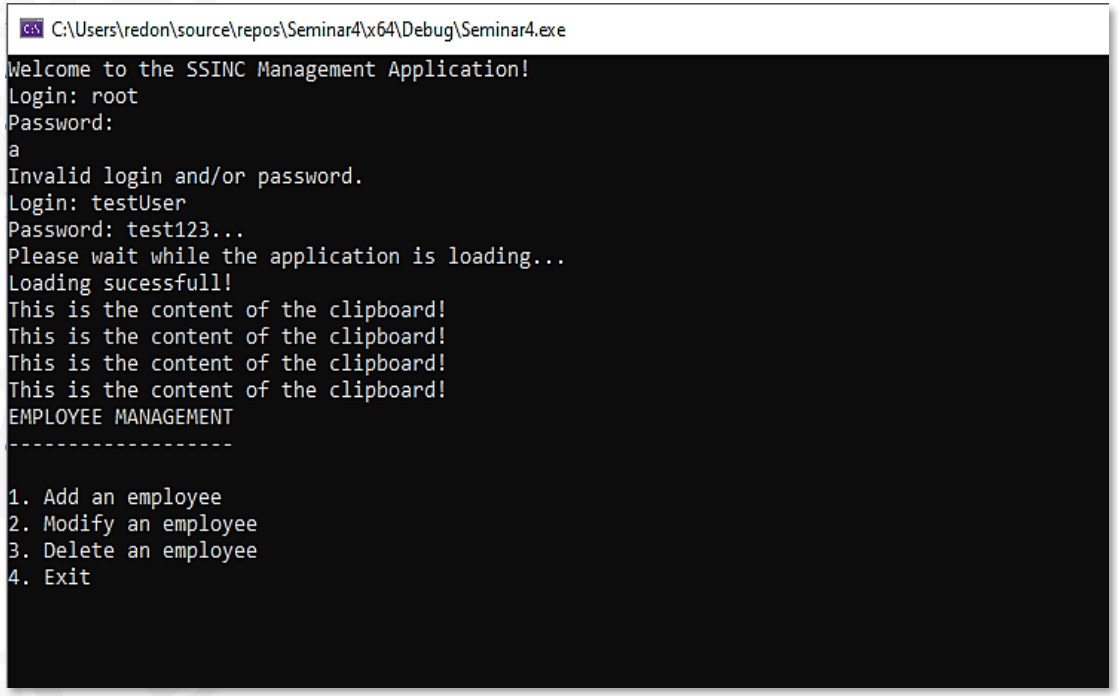
- **To view strings embedded in an executable go to Window - Defined Strings**
  - This window shows a list of all the defined strings
  - Including its address within the code, value and string representation
  - We see that there are a lot of strings belonging to application standard messages
  - But we also located two suspicious ones "testUser" and "test123…" that seems to be a username / password
- **What happens if we try this combination?**



Hidden? Data within an executable

José Manuel
Redondo López

- **We correctly guessed that these were a combination of valid user / password**
- **Is this situation common?**
  - Certainly, it is not rare to find backdoors / privileged information hardcoded in an executable file
  - Valid usernames
  - Passwords
  - API Keys
  - IDs
  - "Secret" (not linked) URLs / IPs / routes…
  - Undocumented functionality names
  - …
- **Hardcoding important stuff is not safe!**



NOTE: Our sample application crashes if the clipboard contents is not a string

< Go to Index

Source: zona-militar.com

Decompiling >

Data >

Code >

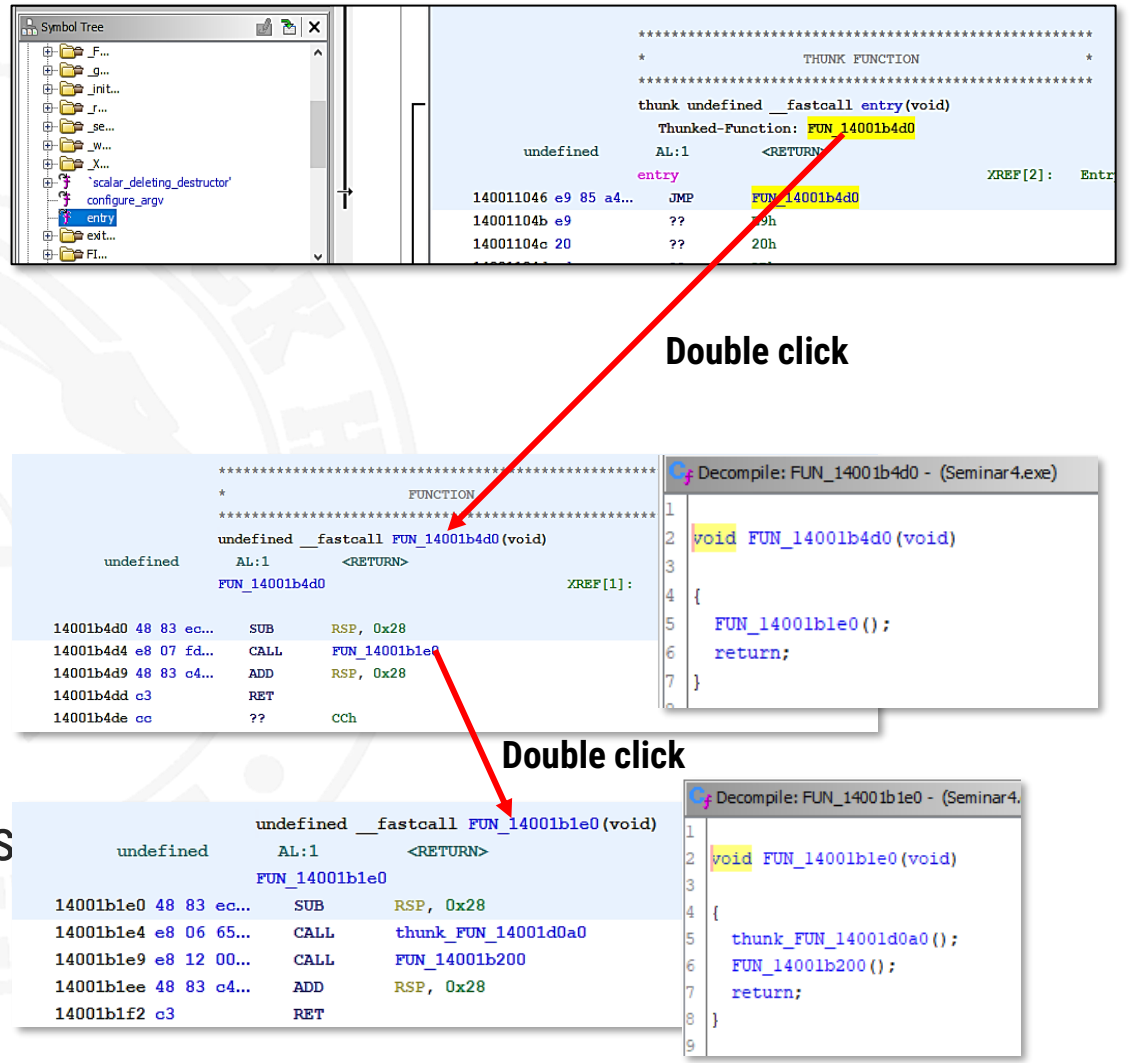
Inspecting executable code and behavior

See what our program does

"Following" a program

José Manuel
Redondo López

- **Program behavior can be inspected analyzing its calls from its entry point**
  - Program entry points have a special name in the symbol tree (`entry`)
  - From there we can follow the execution
- **We may find thunked functions**
  - Functions used to **inject an additional calculations** into another one
  - Used to delay a calculation until its result is needed
  - Or to insert operations at function beginning or end
  - Commonly used by compiler's code generation phas
  - Most of the times we can just **follow them ignoring their contents**



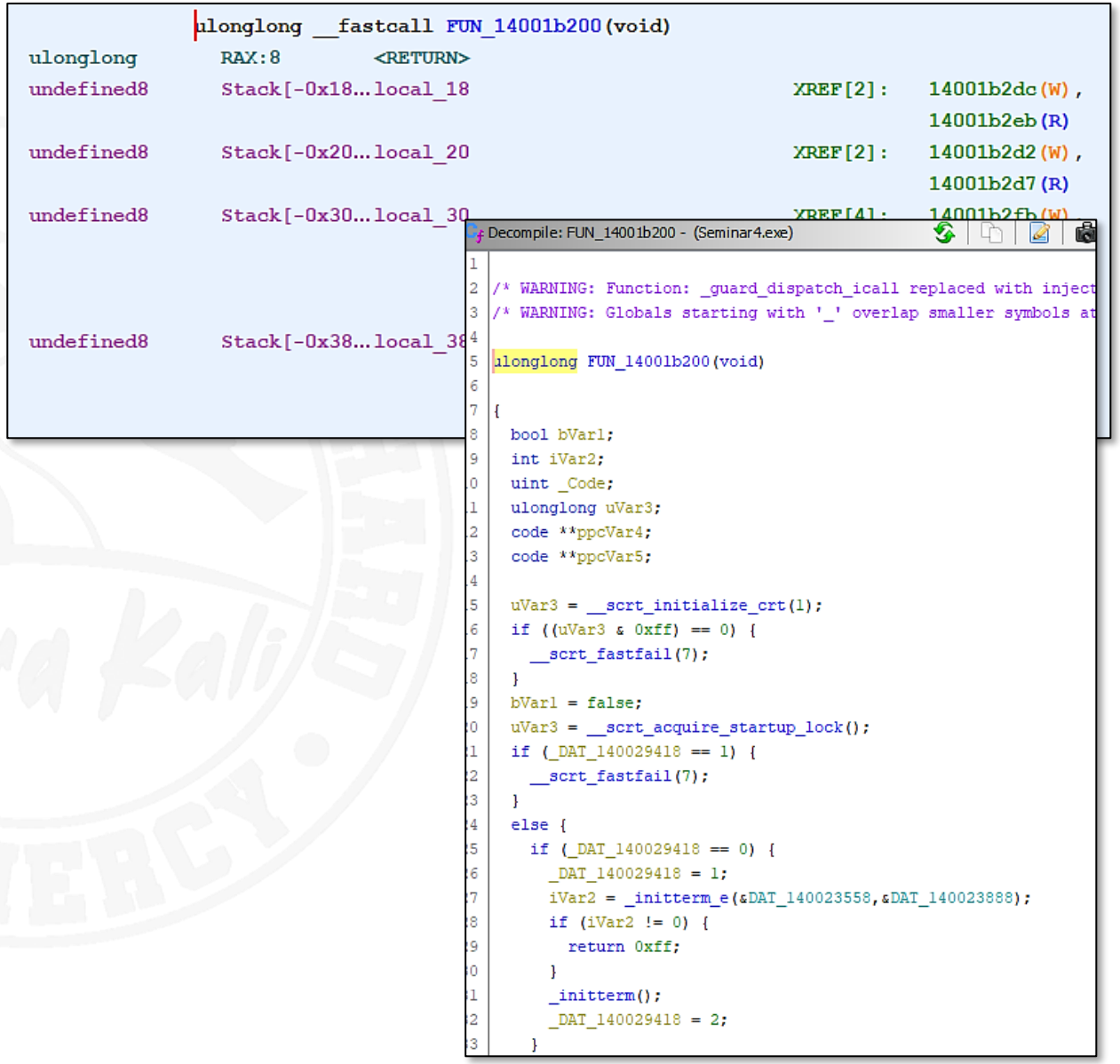
Decompiling

Obtaining readable source code

"Following" a program: Decompiling

José Manuel
Redondo López

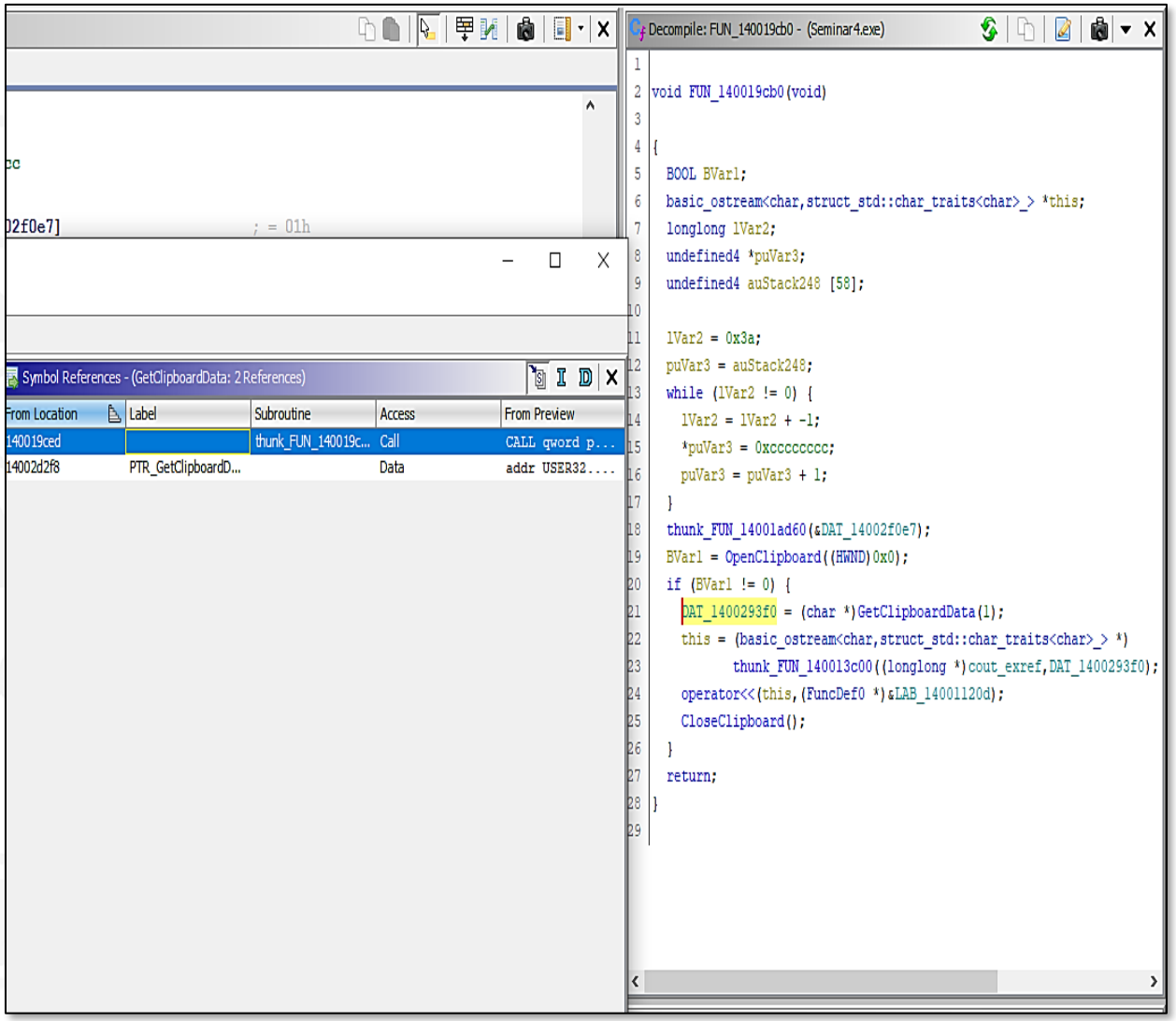
- **Following the call sequence, we will arrive to functions with the actual program contents**
- **We can see the layout of the assembly code and the output of a decompiler**
- **This decompiler will try to express the ASM code in equivalent C code**
  - This code is more readable, but unfortunately **it is also difficult to follow**
  - Compiling to x86 assembly loses information
    - About variable and function names, among other things
    - Also, some different code constructs can output the same assembly source



VIEW DECOMPILER OUTPUT

José Manuel
Redondo López

- **This means that, while helpful, we cannot obtain the original source code**
  - Even worse if we cannot use debugger (.pdb) files
- **Decompiler output syncs with changes made within the Listing view**
  - Yes, Ghidra allows us **to perform changes in the program executable code**
  - Why don't we do that? because is way beyond the objectives of this course ☺
  - Any changes made to the function, variable, or argument names in either the Listing or Decompile view will update the other window



Investigating an API reference

José Manuel
Redondo López

- **It is a common practice to identify interesting code based on OS API references**
- **To do that go to Window - Symbol References**
- **Select an API call you want to investigate and identify where that function is called (Symbol References window)**
  - For example, the `GetClipboardData` API is worth investigating to understand what this program do
  - To locate a reference to this API simply click on the API name and choose a reference on the right-hand side
  - Clicking a reference populates the Listing view with the appropriate disassembly (ASM + C code)

INVESTIGATING AN API REFERENCE

José Manuel
Redondo López

```
140019cda 33 c9          XOR       ECX, ECX
140019cdc ff 15 1e...    CALL      qword ptr [->USER32.DLL::OpenClipboard]
140019ce2 85 c0          TEST      EAX, EAX
140019ce4 75 02          JNZ       LAB_140019ce8
140019ce6 eb 3b          JMP       LAB_140019d23

                     LAB_140019ce8                          XREF[1]:   140019ce4(j)
140019ce8 b9 01 00...    MOV       ECX, 0x1
140019ced ff 15 05...    CALL      qword ptr [->USER32.DLL::GetClipboardData]
```

```
undefined4 auStack248 [58];

lVar2 = 0x3a;
puVar3 = auStack248;
while (lVar2 != 0) {
  lVar2 = lVar2 + -1;
  *puVar3 = 0xcccccccc;
  puVar3 = puVar3 + 1;
}
thunk_FUN_1400lad60(&DAT_14002f0e7);
BVar1 = OpenClipboard((HWND)0x0);
```

Symbol References, Symbol Table [CodeBrowser: Seminar4:/Seminar4.exe]

File Edit Tools Help

Symbol Table - (Filter settings matched 780 Symbols)

Name	Location	Symbol T...	Data Type	Namespace	Source	Referenc...	Offcut Re...
FUN_140020600	140020600	Function	undefined	Global	Default	0	0
FUN_140020630	140020630	Function	undefined	Global	Default	0	0
FUN_140020660	140020660	Function	undefined	Global	Default	0	0
FUN_140020690	140020690	Function	undefined	Global	Default	0	0
FUN_1400206c0	1400206c0	Function	undefined	Global	Default	0	0
FUN_1400206f0	1400206f0	Function	undefined	Global	Default	0	0
FUN_140020720	140020720	Function	undefined	Global	Default	0	0
FUN_140020750	140020750	Function	undefined	Global	Default	0	0
FUN_140020780	140020780	Function	undefined	Global	Default	0	0
FUN_1400207b0	1400207b0	Function	undefined	Global	Default	0	0
FUN_1400207e0	1400207e0	Function	undefined	Global	Default	0	0
FUN_140020880	140020880	Function	undefined	Global	Default	0	0
FUN_1400208c0	1400208c0	Function	ulonglong	Global	Default	0	0
FUN_1400219a0	1400219a0	Function	undefined	Global	Default	1	0
GetClipboardData	External [0002ddfe]	External ...	HANDLE	USER32....	Imported	2	0
GetCurrentProcess	External [0002eab4]	External ...	HANDLE	KERNEL3...	Imported	2	0

Symbol References - (GetClipboardData: 2 References)

From Location	Label	Subroutine	Access	From Preview
140019ced		thunk_FUN_140019c...	Call	CALL qword p...
14002d2f8	PTR_GetClipboardD...		Data	addr USER32....

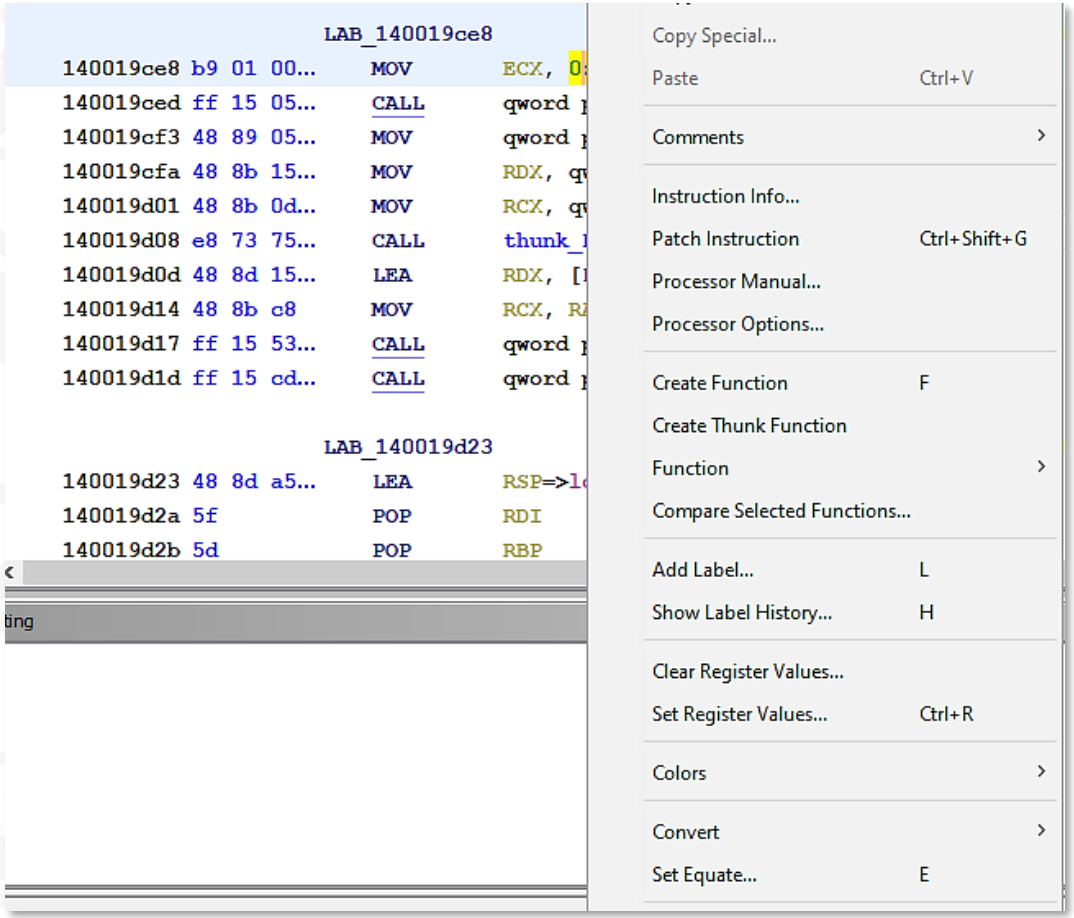
Data annotation and location

Giving meaning to data container inside source code

Data annotation

José Manuel
Redondo López

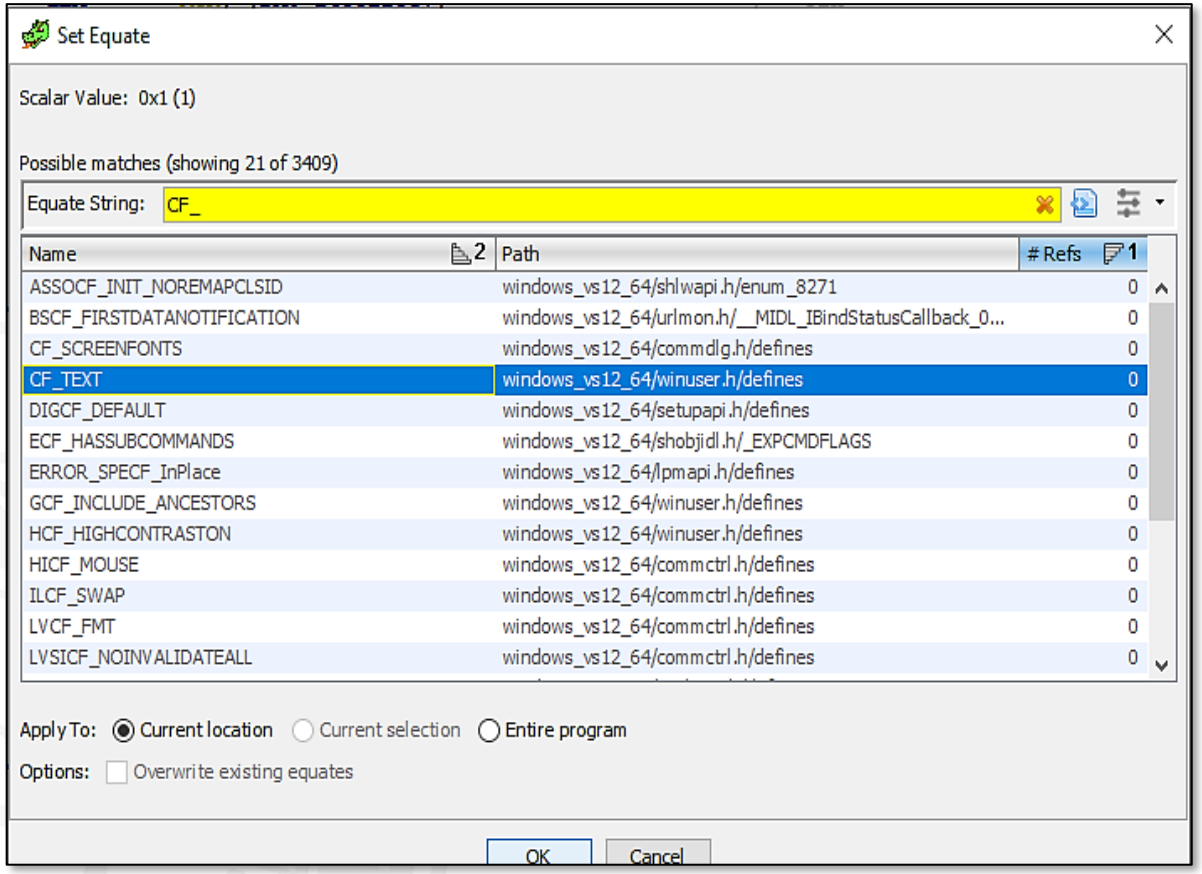
- **In the listing view we see**
  - The initial `CALL` to the API function
  - Potential `PUSH` instructions above the `CALL` refer to the function arguments (if any)
  - On the right, autogenerated comments that identify the function parameters
    - As described on Microsoft's website
- **The only parameter of `GetClipboardData` is the expected clipboard data format**
  - https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-getclipboarddata
  - The value `0x1` represents a symbolic constant
  - We can convert this to a **more human-readable representation**
    - By right-clicking the value and selecting "**Set Equate…**"



DATA ANNOTATION

José Manuel
Redondo López

- **The resulting window offers options for the chosen hexadecimal value**
- **To know the correct one, we check the API call documentation**
  - https://docs.microsoft.com/es-es/windows/win32/dataxchg/clipboard-formats
  - Most of the clipboard formats begin with CF_
  - Typing this into the "Equate String" field yields only one compatible option: CF_TEXT
    - Means that the clipboard is expected to contain text
    - https://docs.microsoft.com/es-es/windows/win32/dataxchg/clipboard-formats
  - Select this value and press OK
- **The assembly now displays the text, not the hexadecimal value**
- **We know how to annotate flags and values and make it persist!**



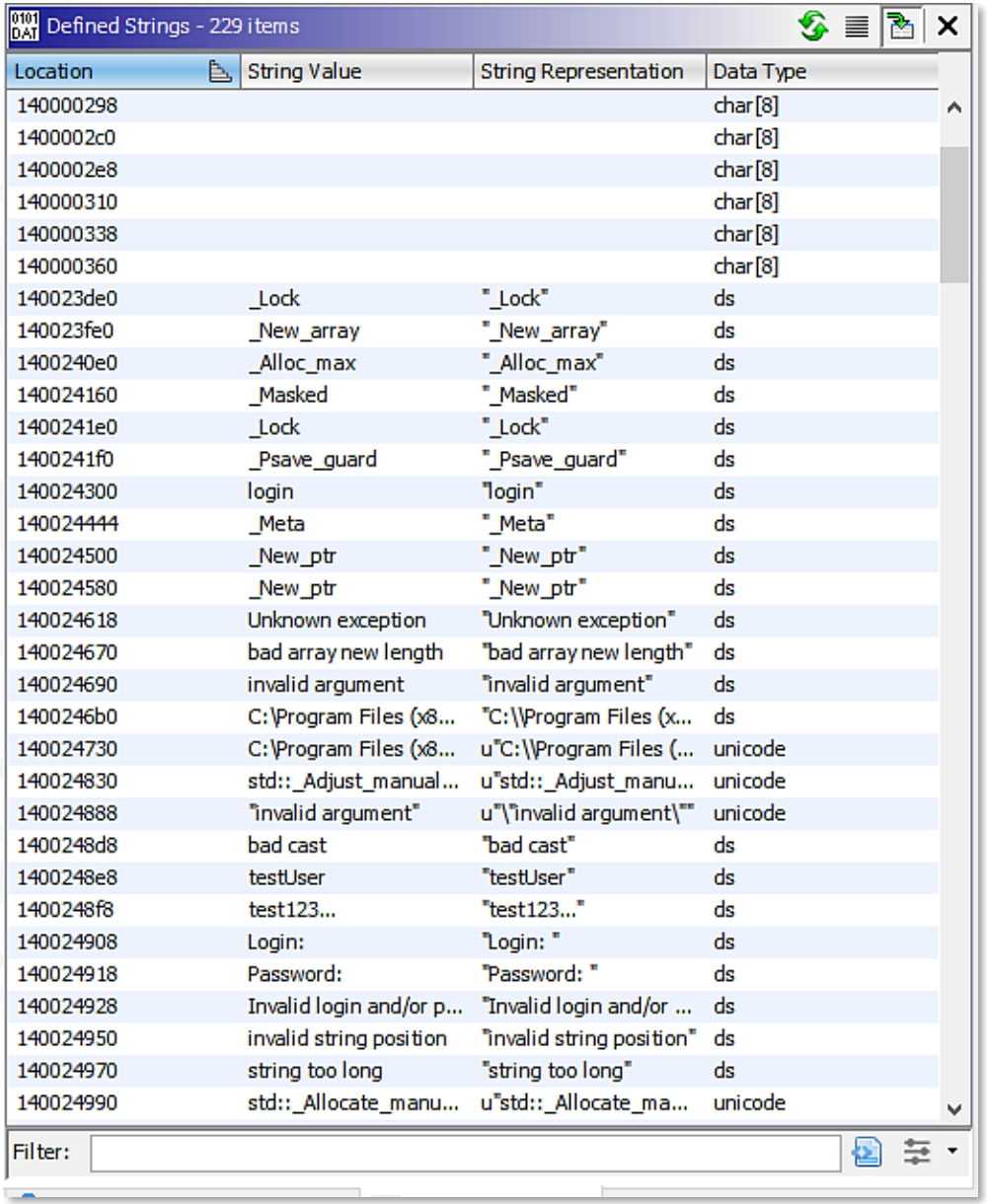

```
B_140019ce8                                    XREF[1]:    140019ce4(j)
  MOV       ECX, CF_TEXT
  CALL      qword ptr [->USER32.DLL::GetClipboardData]
  MOV       qword ptr [DAT_1400293f0], RAX
  MOV       RDX, qword ptr [DAT_1400293f0]
  MOV       RCX, qword ptr [->MSVCP140D.DLL::std::cou...; = 0002e464
  CALL      thunk_FUN_140013c00                          ; undefined thunk_FUN_140
  LEA       RDX, [LAB_14001120d]
  MOV       RCX, RAX
  CALL      qword ptr [->MSVCP140D.DLL::std::basic_os...
  CALL      qword ptr [->USER32.DLL::CloseClipboard]

B_140019d23                                    XREF[1]:    140019ce6(j)
```

LOCATING CODE FROM STRING REFERENCES

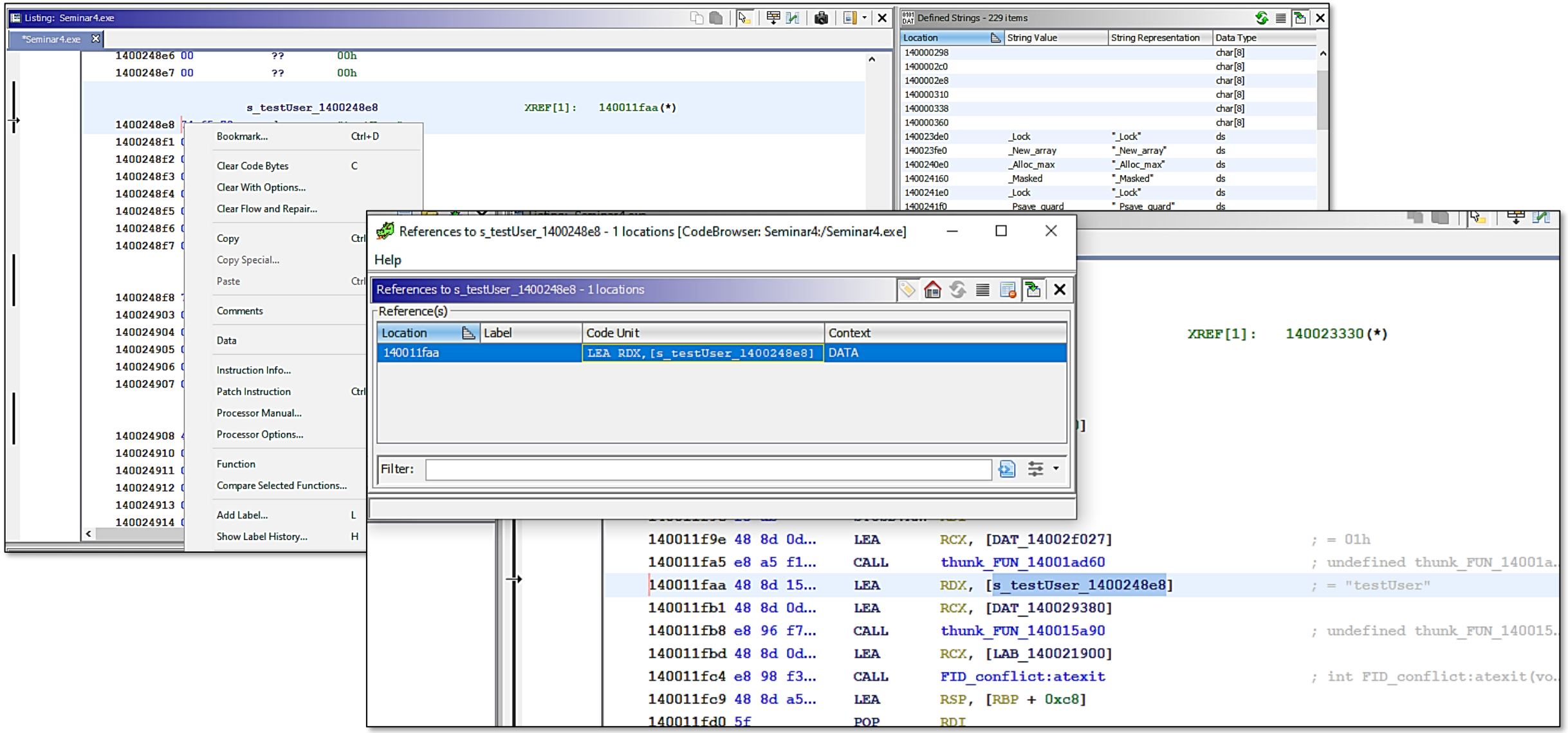
José Manuel
Redondo López

- **String analysis not only allows to guess hidden information within the executable**
  - But also, **to trace where these strings are used in the source code**
- **Clicking on the row associated with a string populates the Listing window with the data at its address**
- **To identify references to this string we can right-click in the blue area - References - Show References to Address**
- **Let's see what happens with the "testUser" string we located**



LOCATING CODE FROM STRING REFERENCES

José Manuel
Redondo López



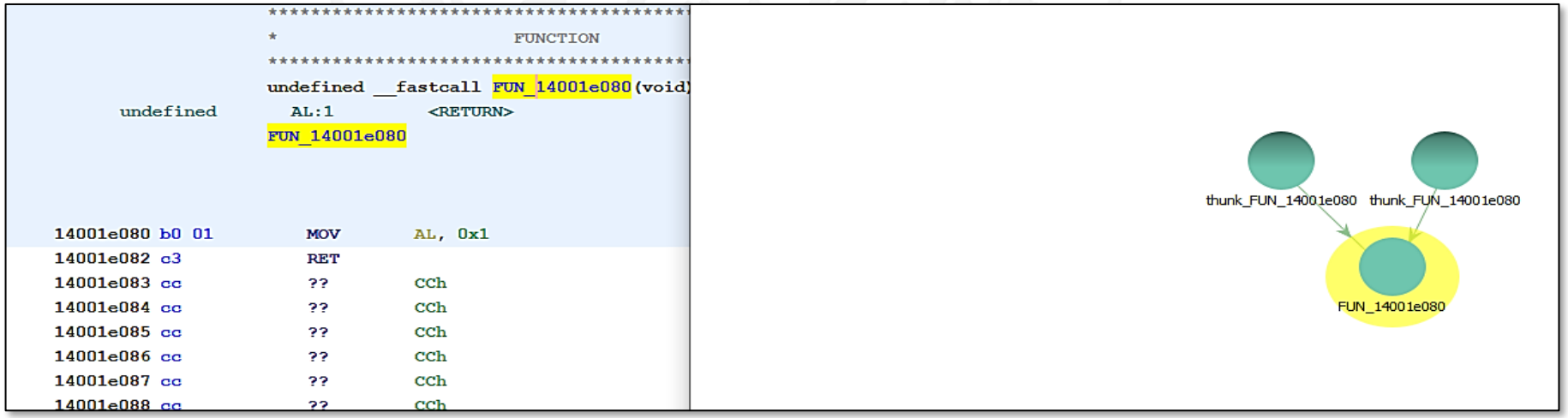
Call graphs

Tools to help us understanding the source code

FUNCTION CALL GRAPHS

José Manuel
Redondo López

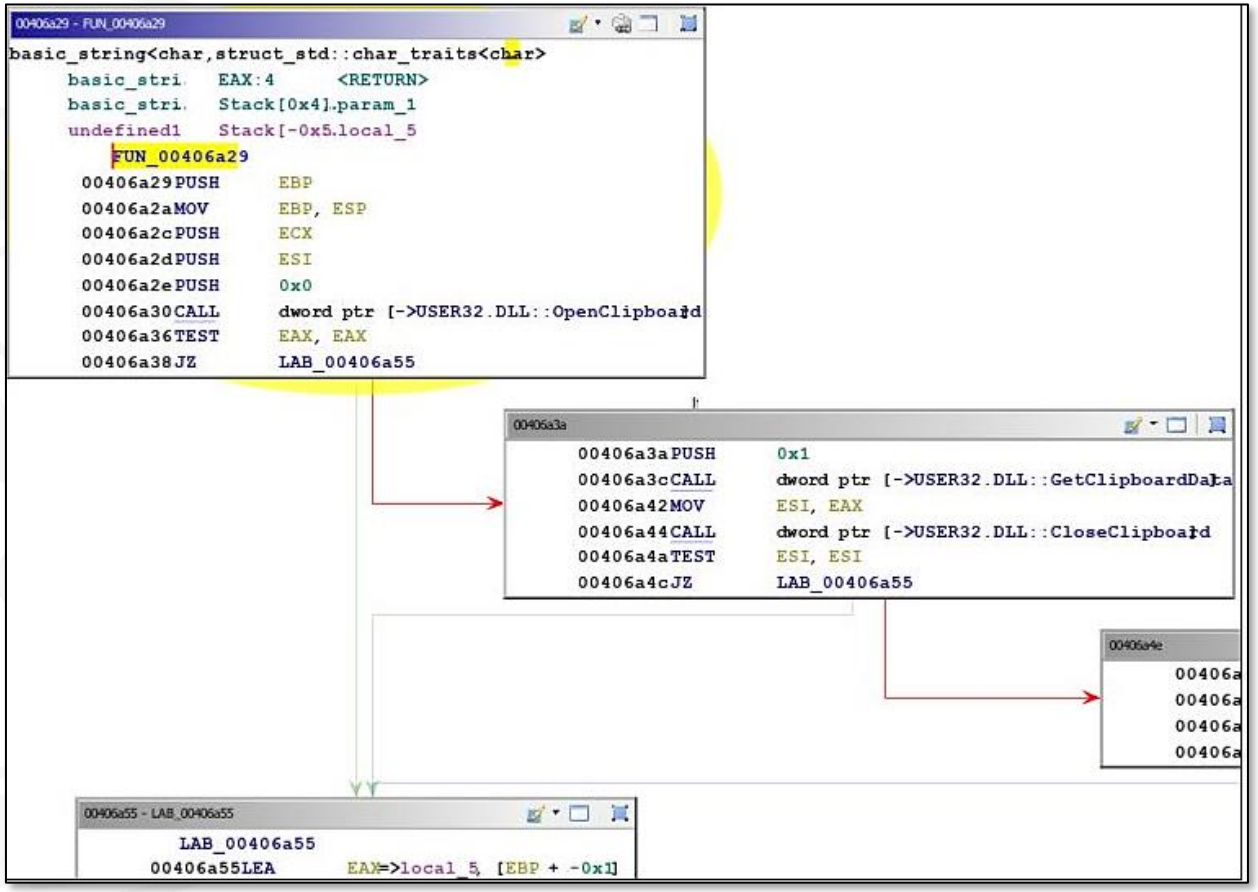
- **Once a reference is identified, we can also analyze the call sequence that ends in the code that uses it**
- **To do this, we can go to the beginning of the function that uses the string and click on Window- Function call graph**
  - This displays the relationships between functions and provides **a high-level overview of function calls**
    - Much faster than doing it manually!
  - This may allow us to discover **undocumented or unexpected calls to Windows APIs**
    - Potential malicious activity…
  - Or just to better understand the analyzed program behavior!



FUNCTION GRAPHS

José Manuel
Redondo López

- **Checking function behavior may be done reviewing the disassembly text in the decompile window**
- **However, it is often beneficial to see a visual representation of decision points within a function**
- **You can obtain it with the Window - Function Graph**
- **It displays the function's individual code blocks and the arrows representing conditional and unconditional jumps**



UNDERSTANDING ASSEMBLY

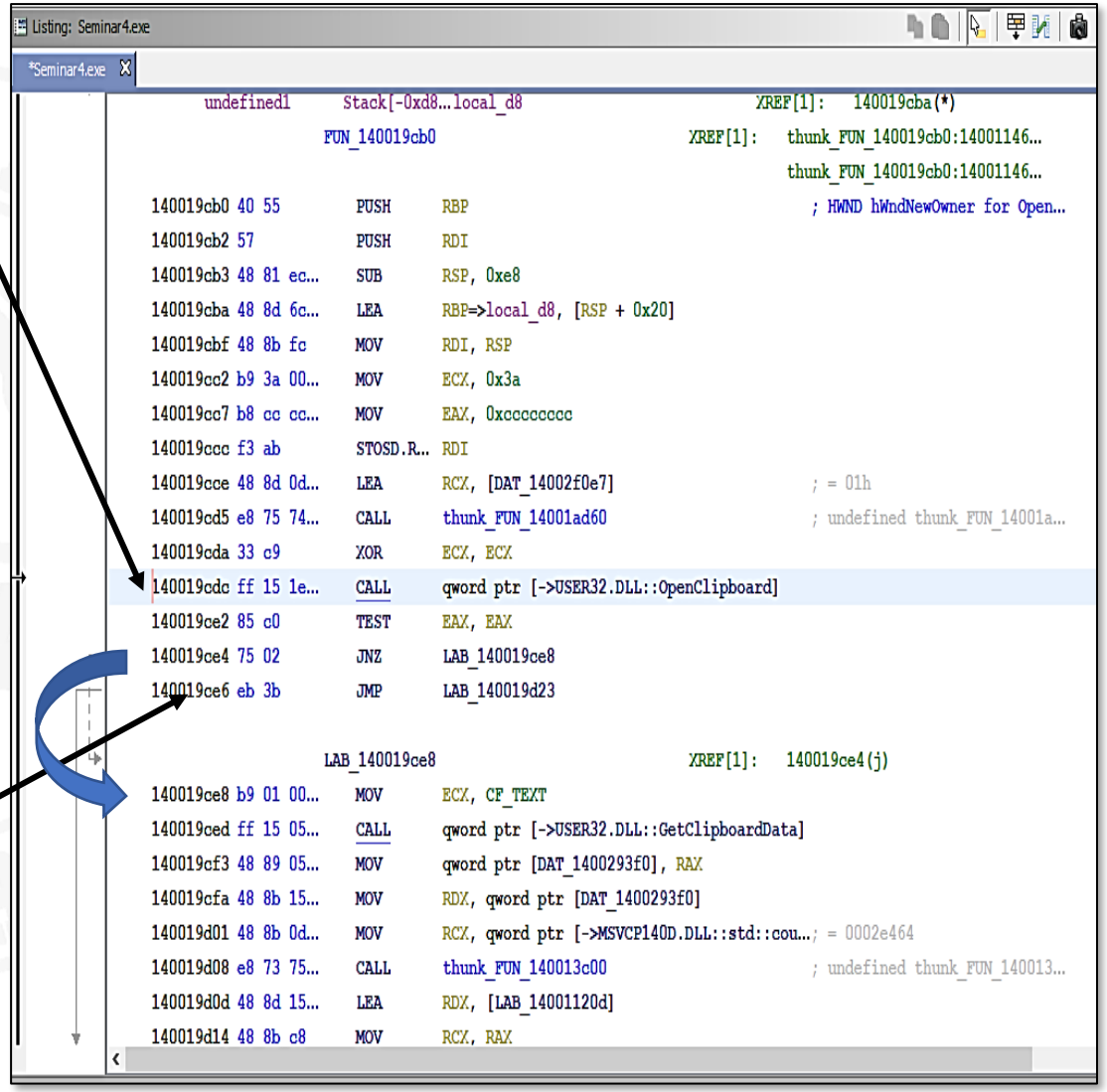
José Manuel
Redondo López

- **Finally, once we can view data, functions and call sequences, we need to understand the program code**
- **This requires a high level of assembly interpretation skills, experience and patience (you saw some in FCR)**
  - That is way beyond the objectives of this course
- **However, with practice is possible to understand most of the possible instruction patterns we may find**
  - Code translators typically use coherent code structures that we may be able to recognize
  - And don't forget that we have the valuable help of the decompiler and the function graph to better understand what the code does
- **A good starting reference is: https://www.cs.virginia.edu/~evans/cs216/guides/x86.html**
- **Let's see an example**

UNDERSTANDING ASSEMBLY

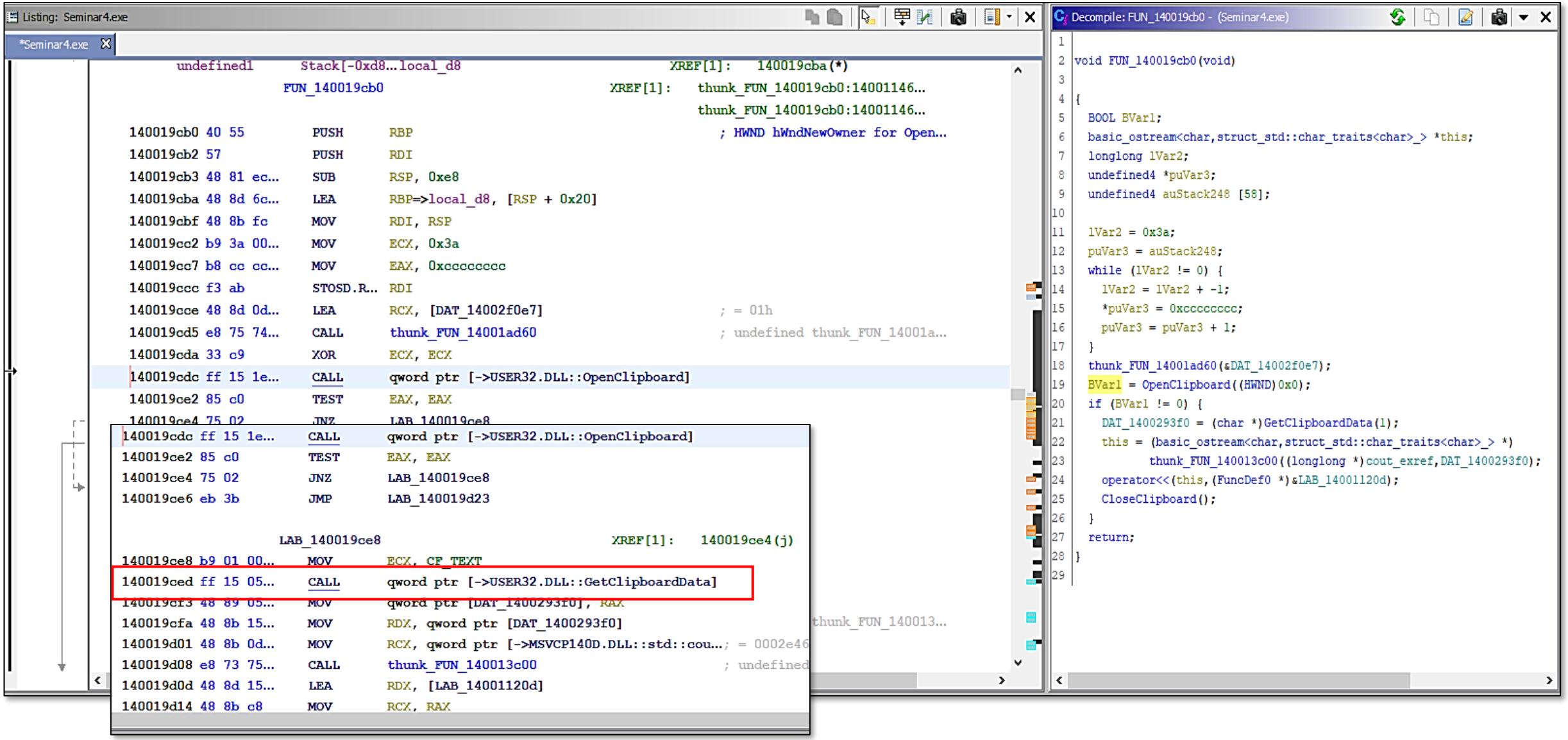
José Manuel
Redondo López

- **In this code block we call to `OpenClipboard`**
  - Followed by `TEST` and `JNZ` (jump if not zero) instructions
  - These evaluate the return value stored in `EAX`
- **`JNZ` indicates that execution should jump to `LAB_140019ce8` if `EAX` is not `0`**
  - A return value of zero means a failed call to `OpenClipboard`
- **If this jump is performed, the code block where other API calls obtain clipboard data will be run**
- **If not, the execution will `JMP` (unconditional jump) to another address**
  - If the clipboard opens, obtain its data
  - Otherwise, do not execute code to gather clipboard information



UNDERSTANDING ASSEMBLY

José Manuel
Redondo López



More information

José Manuel
Redondo López

- **Ghidra has become one of the most popular reversing tools**
- **There are many tutorials teaching how to use Ghidra and introducing you into the reversing world**
  - **(YouTube tutorial channel)** https://www.youtube.com/playlist? list=PLRAe18TJ_NTE9cr18OPphn82WS8gVv-te
  - **An Introduction To Code Analysis with Ghidra**: https://threatvector.cylance.com/en_us/home/an-introduction-to-code-analysis-with-ghidra.html
  - **Using OOAnalyzer to reverse engineering object-oriented code with Ghidra**: https://insights.sei.cmu.edu/sei_blog/2019/07/using-ooanalyzer-to-reverse-engineer-object-oriented-code-with-ghidra.html
  - **Practical case: Crack Me 0x01**: https://maxkersten.nl/binary-analysis-course/assembly-basics/practical-case-crack-me-0x01/
- **Outside Ghidra, you can also use this resources**
  - **Analyzing Modern Malware Techniques**
    - Part 1: https://0x00sec.org/t/analyzing-modern-malware-techniques-part-1/18663
    - Part 2: https://0x00sec.org/t/analyzing-modern-malware-techniques-part-2/18765
  - **University of Cincinnati course CS6038/CS5138: Malware Analysis**: https://class.malware.re/

< Go to Index

Source: zona-militar.com

Reversing programs created in high-level languages

Inspecting .Net or Java programs

Reversing programs created in high-level languages

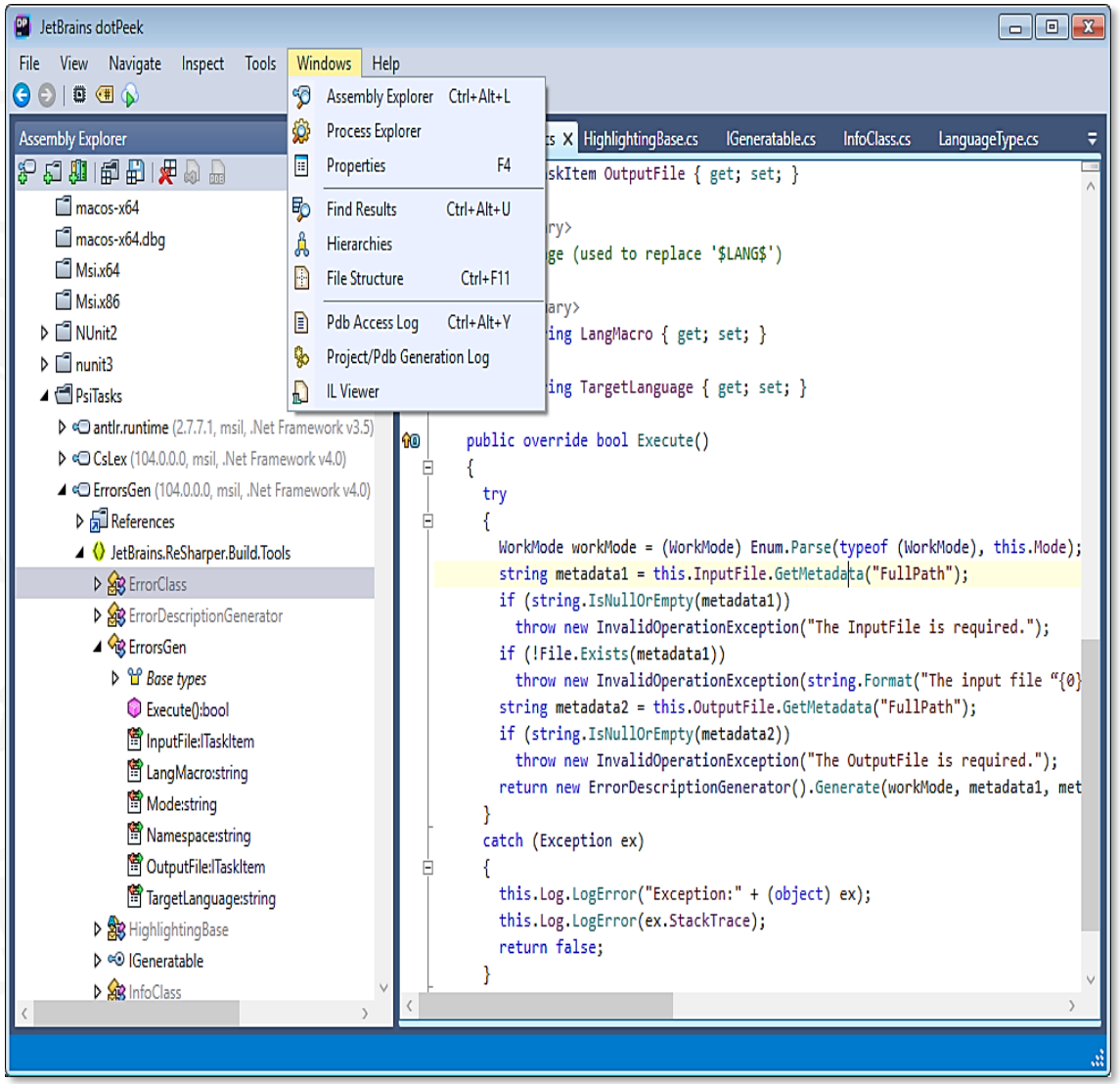
José Manuel
Redondo López

- **We have seen a tool for decompiling and analyzing programs made in low/medium level languages (C/C++)**
- **The same is possible with those coded in C#, Java, or other high-level languages**
  - So, you can analyze programs created in courses like TPP, EDI, MP, etc. even if you don't have the source code
- **In fact, these programs are usually much easier to reverse and analyze**
- **We just must look for a suitable tool for the language in which the program (or the part we want to study) is created**

Analyzing .Net programs

José Manuel
Redondo López

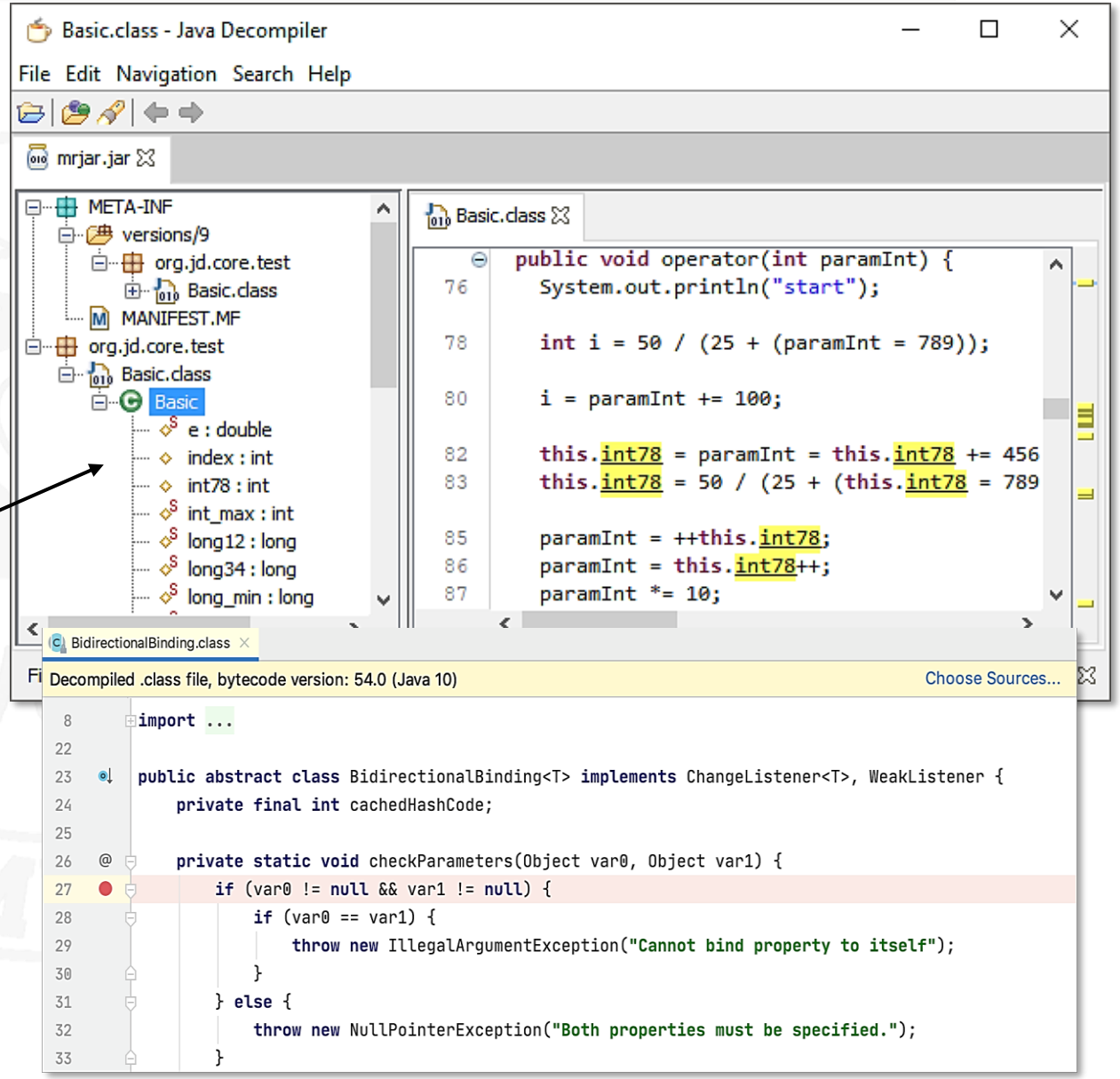
- **There are several decompilers on the market that can do this**
- **A very powerful one is DotPeek**
  - https://www.jetbrains.com/es-es/decompiler/
  - Creating a JetBrains account with a university email account normally gives you free access to all its tools for one year (renewable)
- **Loads `.exe` or `.dll` assemblies and displays the code of their classes**
  - In an understandable and easily analyzable way
- **Therefore, knowing what a program whose sources are not available does gets simpler**



Analyzing Java programs

José Manuel
Redondo López

- Java programs can be decompiled with very good results too
- There are several tools for this, but one of the most popular open-source ones is **JD-GUI**
  - https://github.com/java-decompiler/jd-gui
- The **JetBrains** IDE for Java, **IntelliJ**, also has that feature built in if you open a `.class`
  - https://blog.jetbrains.com/idea/2020/03/java-bytecode-decompiler/



< Go to Index

Source: zona-militar.com

Automated executable inspection tools

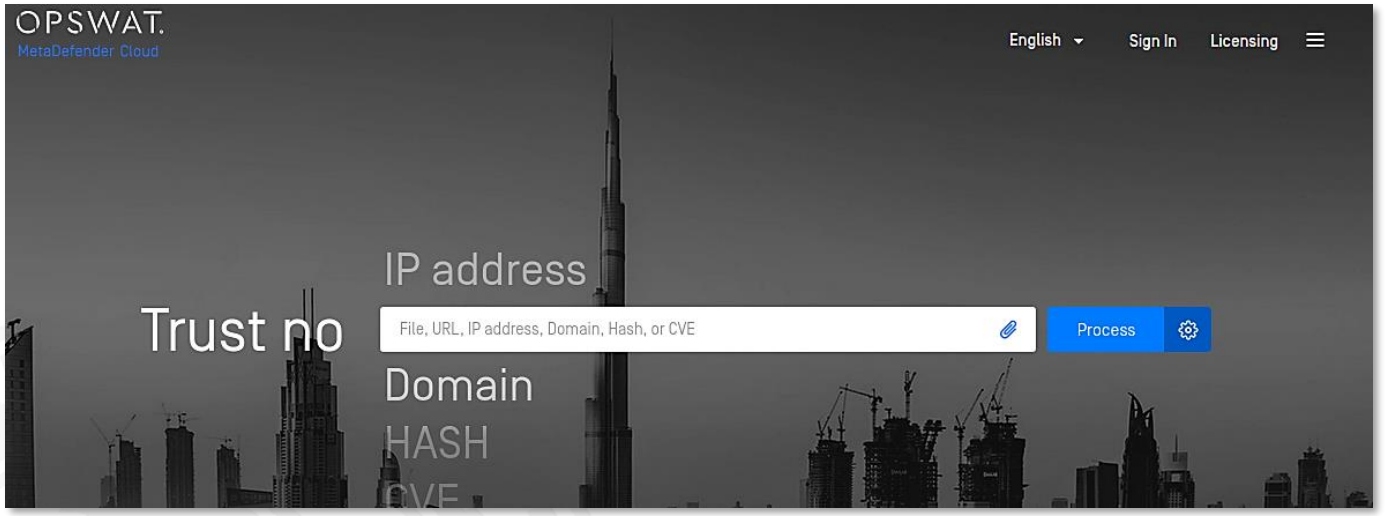
Automated reversing

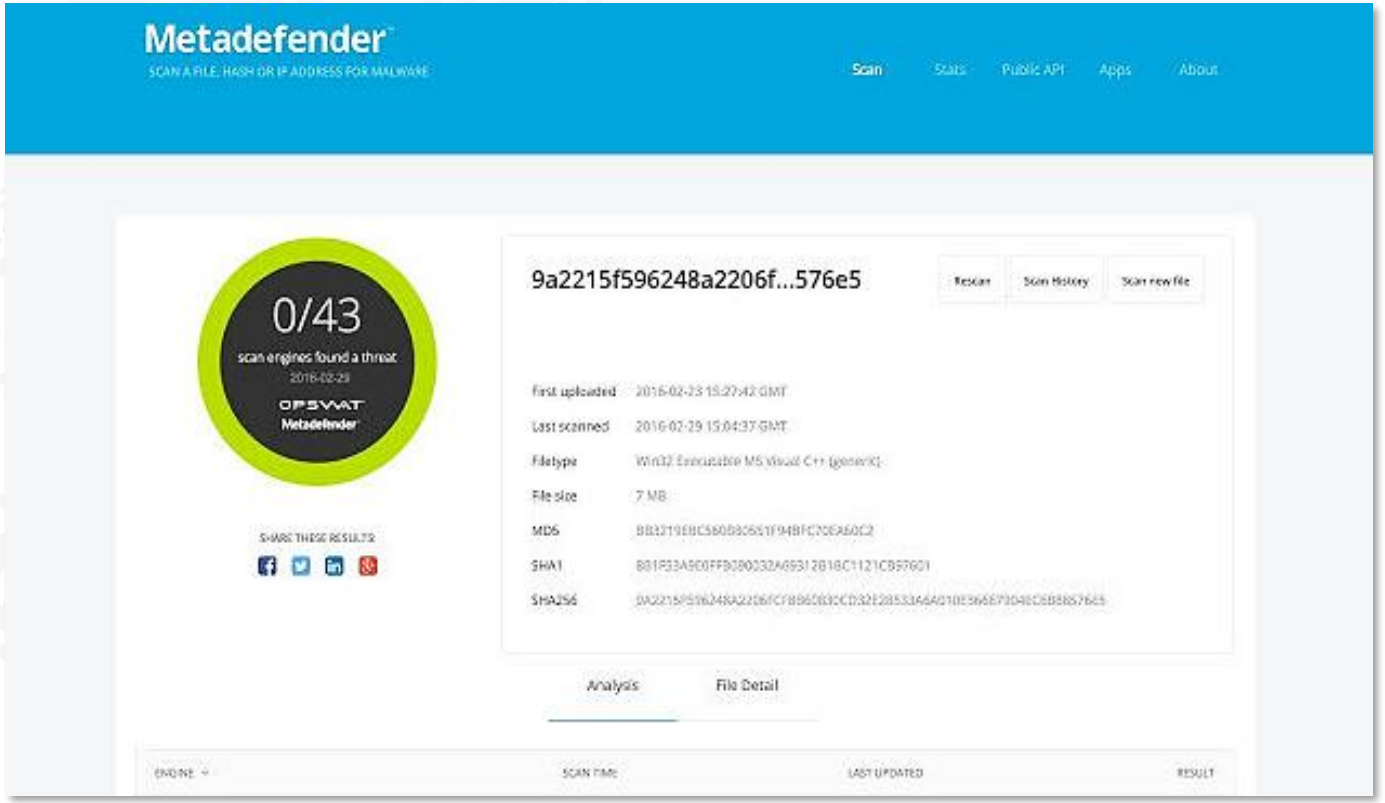
METADEFENDER CLOUD

https://metadefender.opswat.com/

José Manuel
Redondo López

- **Many times, we will only want to know if an executable behaves strangely to avoid malware**
  - In that case, doing a manual analysis may be too much
- **There are tools that allow us to automate this process**
  - We can use one that checks if an executable is detected as malware
- **Metadefender is a tool that analyzes executables to find out precisely that**
  - Another option is Virustotal, seen at the beginning of the course



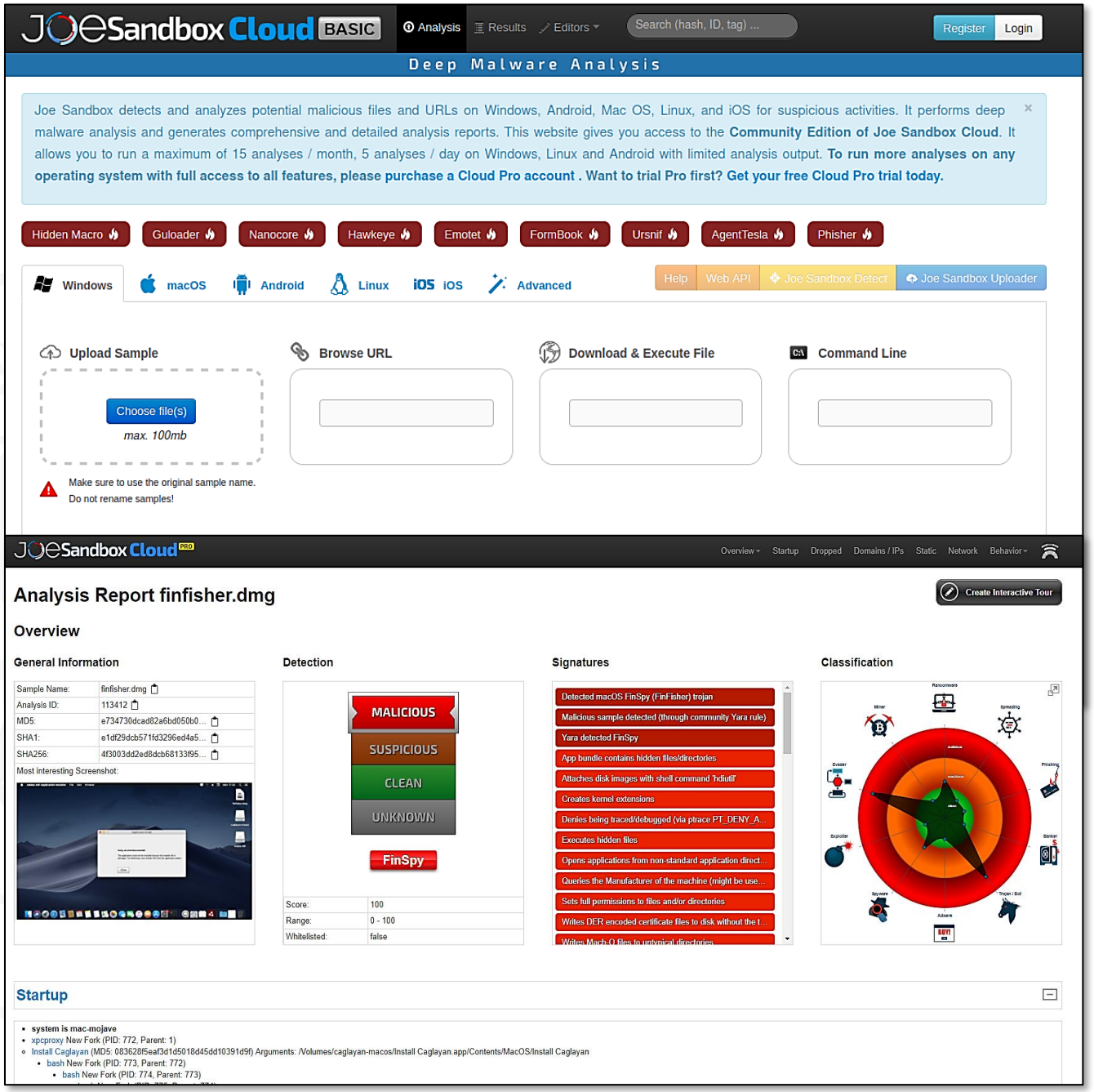


Joe SandBox Cloud

https://www.joesecurity.org/#

José Manuel
Redondo López

- **If we want a second opinion, we can use sandboxing software**
- **They run a program for us and analyze what it does for "suspicious" behavior**
  - System calls known as problematic, attempts to access restricted parts, etc.
- **Analyzes the behavior of executed code**
  - So we can extract conclusions about whether the program is legit or malware
  - Returns us a complete report of the actions and the "dangerousness" that it considers that they have
- **Although it is not free, there is a limited "Community" version**
  - https://www.joesandbox.com/#windows



Source: https://www.joesecurity.org/blog/1896065926254527264

Self-evaluation achievement list: Seminar 4

José Manuel
Redondo López

Rank	Concept
	Understand de purpose and utility of the Ghidra decompiler
	Know how to load an executable and analyze its contents
	Locate source code that uses certain string references
	Know how to display function and call graphs
	Know automated program analysis platforms and its usage
	Know how to locate and investigate string references within an executable
	Know how to follow code execution of a program
	Know how to perform decompilation of code and its limitations
	Know how to annotate data
	Apply your ASM knowledge to the output of Ghidra
	Locate and investigate an API call within an executable
	Choose the appropriate tool to decompile programs created with C# or Java
	**My treasure!: Use the appropriate tools to locate interesting information and behavior of an executable**

SEMINAR 4. INTRODUCTION TO REVERSING

Source: zona-militar.com