

SEMINAR 4

INTRODUCTION TO REVERSING



A primer on analyzing programs



JOSÉ MANUEL REDONDO LÓPEZ "S-81 ISAAC PERAL" PROJECT V4.0



Departamento de Informática
Universidad de Oviedo



Universidad de Oviedo

Logo: @creative_vanesa (Instagram)



INDEX

- ▶ Introduction
- ▶ Launching Ghidra & loading an executable
- ▶ Inspecting executable data
- ▶ Inspecting executable code and behavior
 - Decompiling
 - Data annotation and location
 - Call graphs
- ▶ Reversing programs created in high-level languages
- ▶ Automated executable inspection tools

[< Go to Index](#)

INTRODUCTION

What is Ghidra?



Departamento de Informática
Universidad de Oviedo



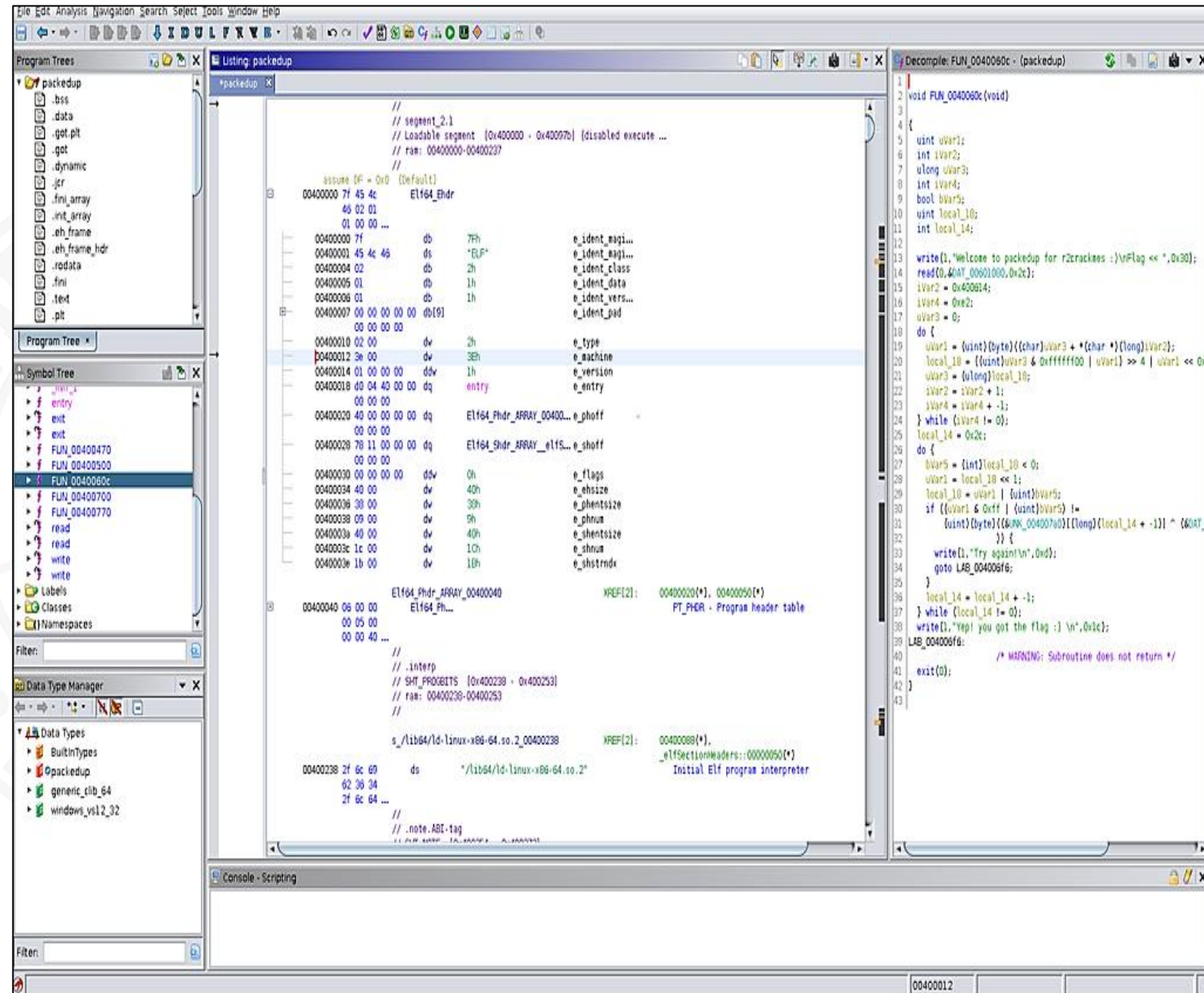
Universidad de Oviedo

- **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**

● 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 or Python (Jython) using its API



[< Go to Index](#)

LAUNCHING GHIDRA & LOADING AN EXECUTABLE

Setting up the analysis environment



Departamento de Informática
Universidad de Oviedo



Universidad de Oviedo

LOADING AN EXECUTABLE



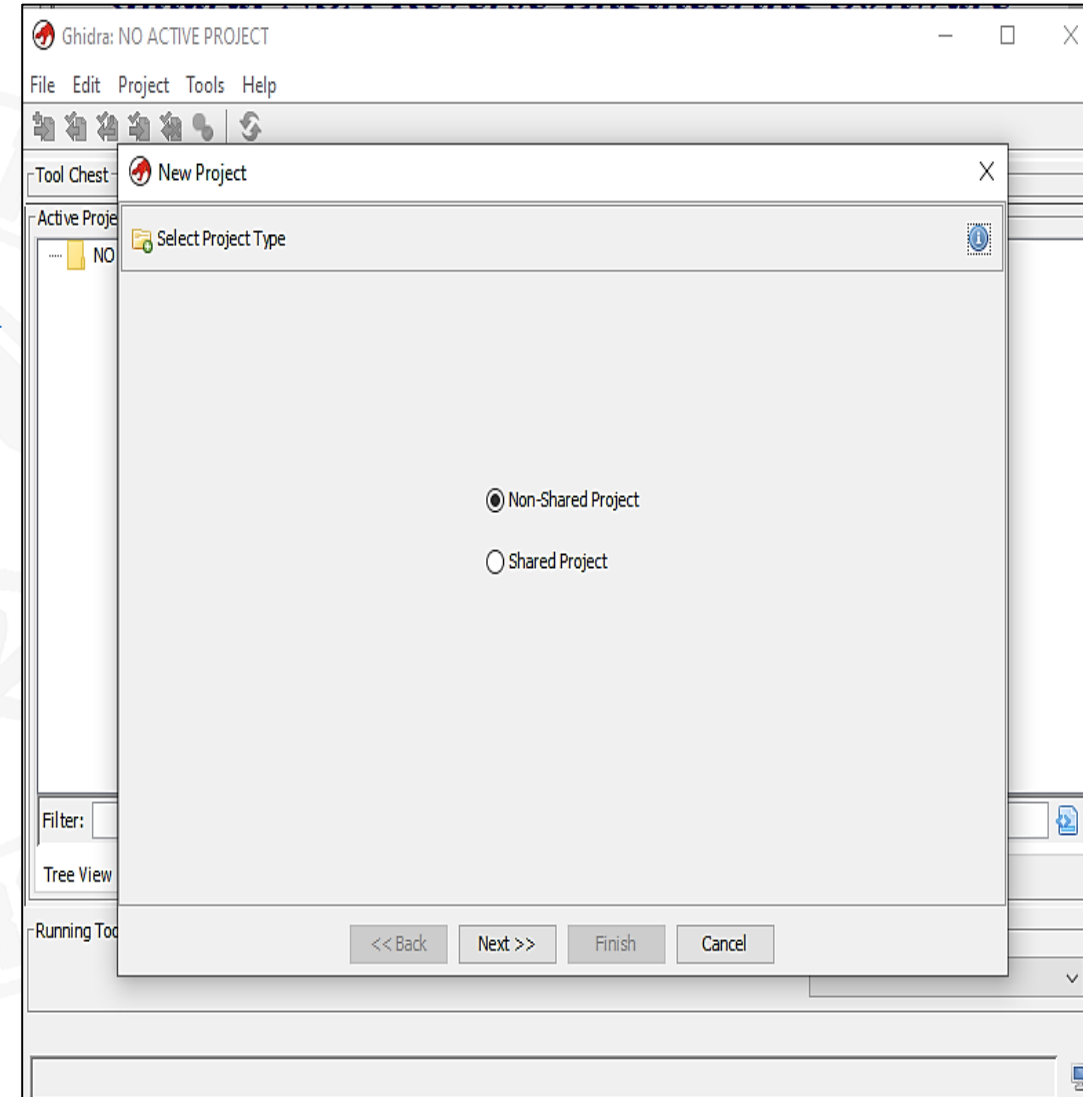
Computer Security

● Installing Ghidra requires

- Downloading and extracting a zip file and launch the batch file
- Installing **JDK/OpenJDK 17+**
 - <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



Computer Security

● 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**

```
C:\Users\redon\source\repos\Seminar4\x64\Debug\Seminar4.exe
Welcome to the SSINC Management Application!
Login: admin
Password: admin
Invalid login and/or password.
Login: root
Password: root
Invalid login and/or password.
Login: root
Password: password
Invalid login and/or password.
Login: admin
Password: lol
Invalid login and/or password.
Login: igave
Password: up
Invalid login and/or password.
Login:
```

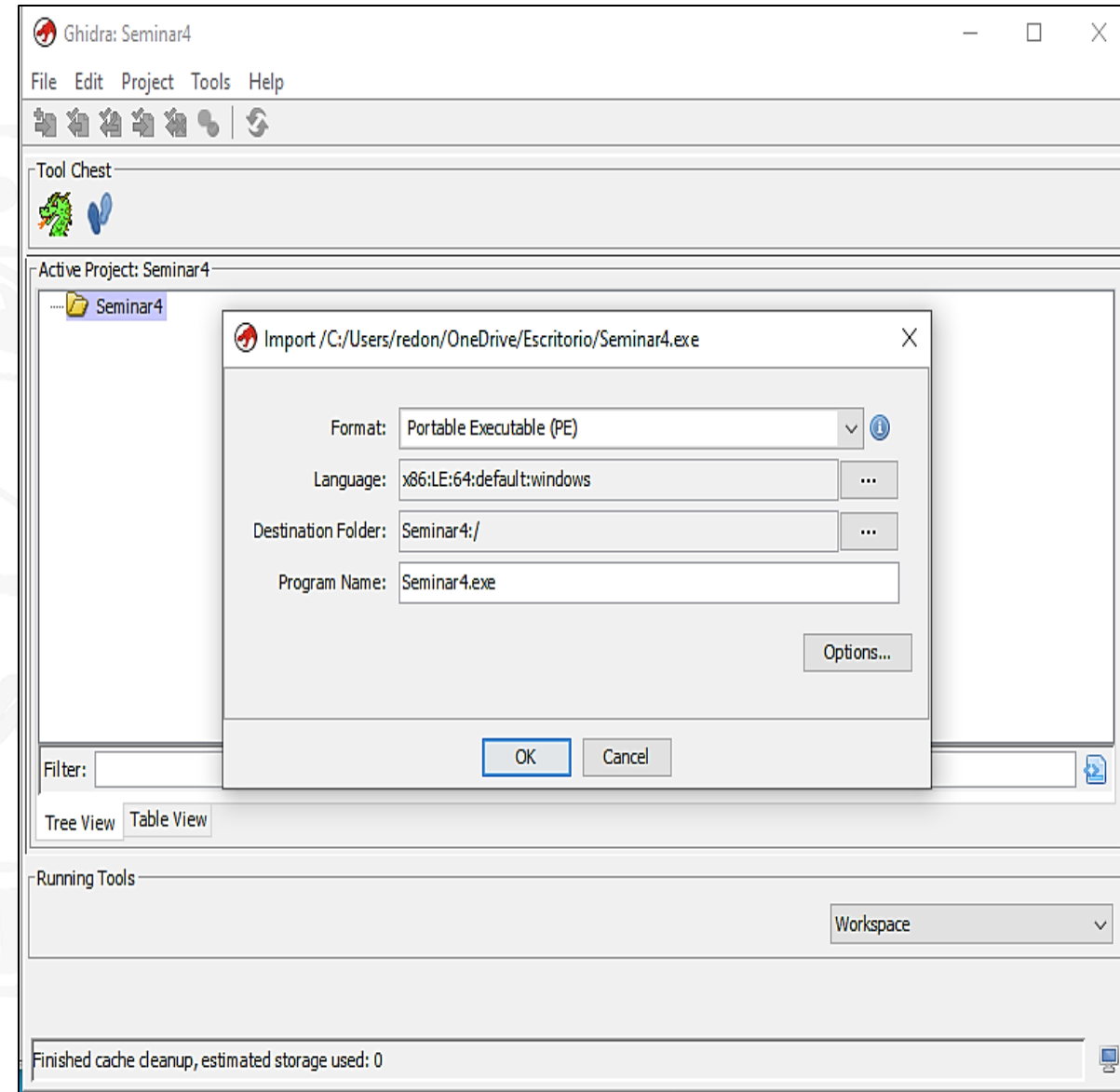
The Visual Studio Project source code is available in the virtual campus!

LOADING AN EXECUTABLE



Computer Security

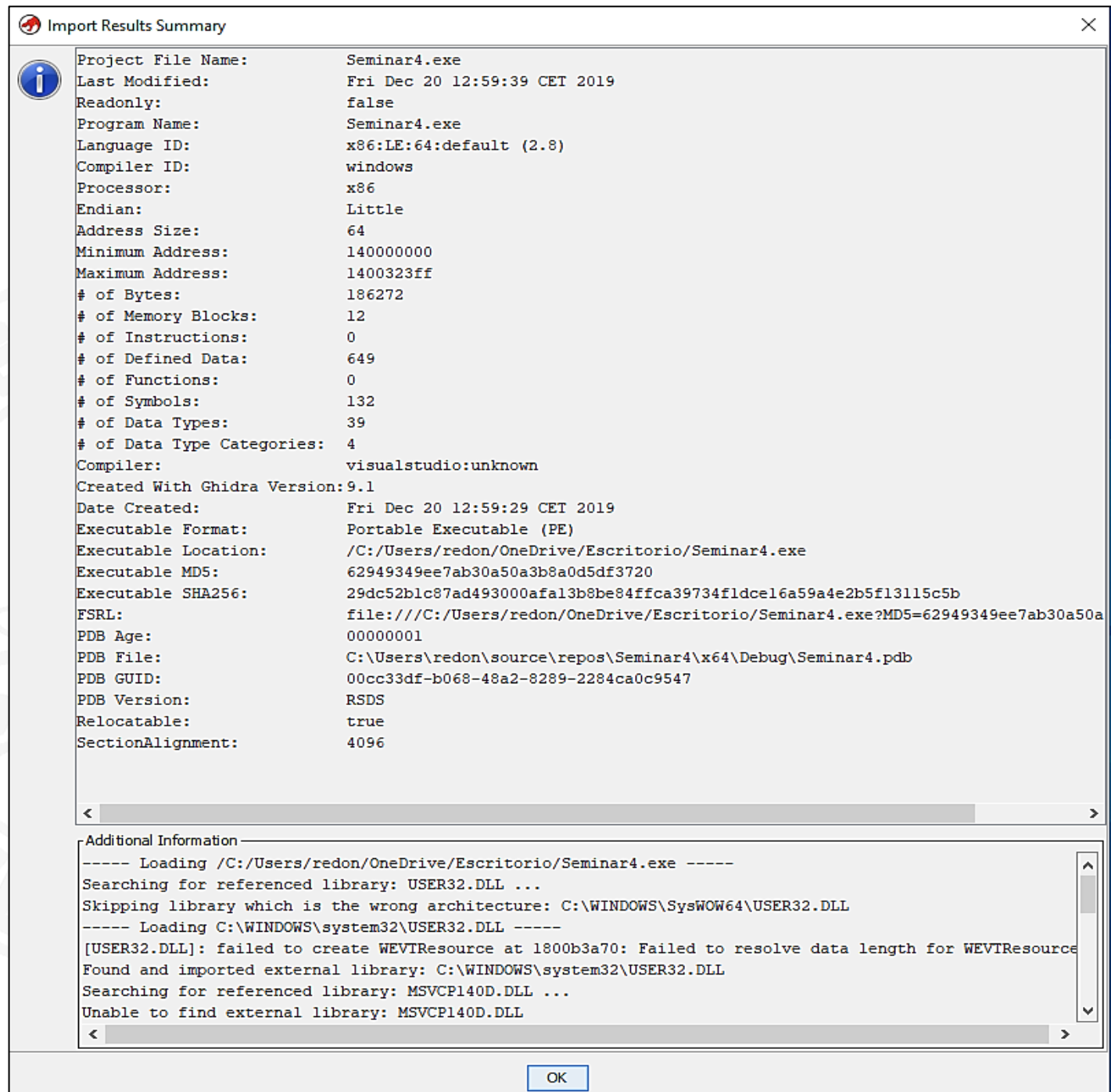
- To analyze an executable file, you need to drag and drop the `.exe` file into the project window
- This will launch a dialog box where you can accept the defaults and click OK
- If in the same folder a `.pdb` file exists, Ghidra will attempt to use it
 - These files contain the debugging symbols
 -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



LOADED EXECUTABLE INFORMATION



- 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

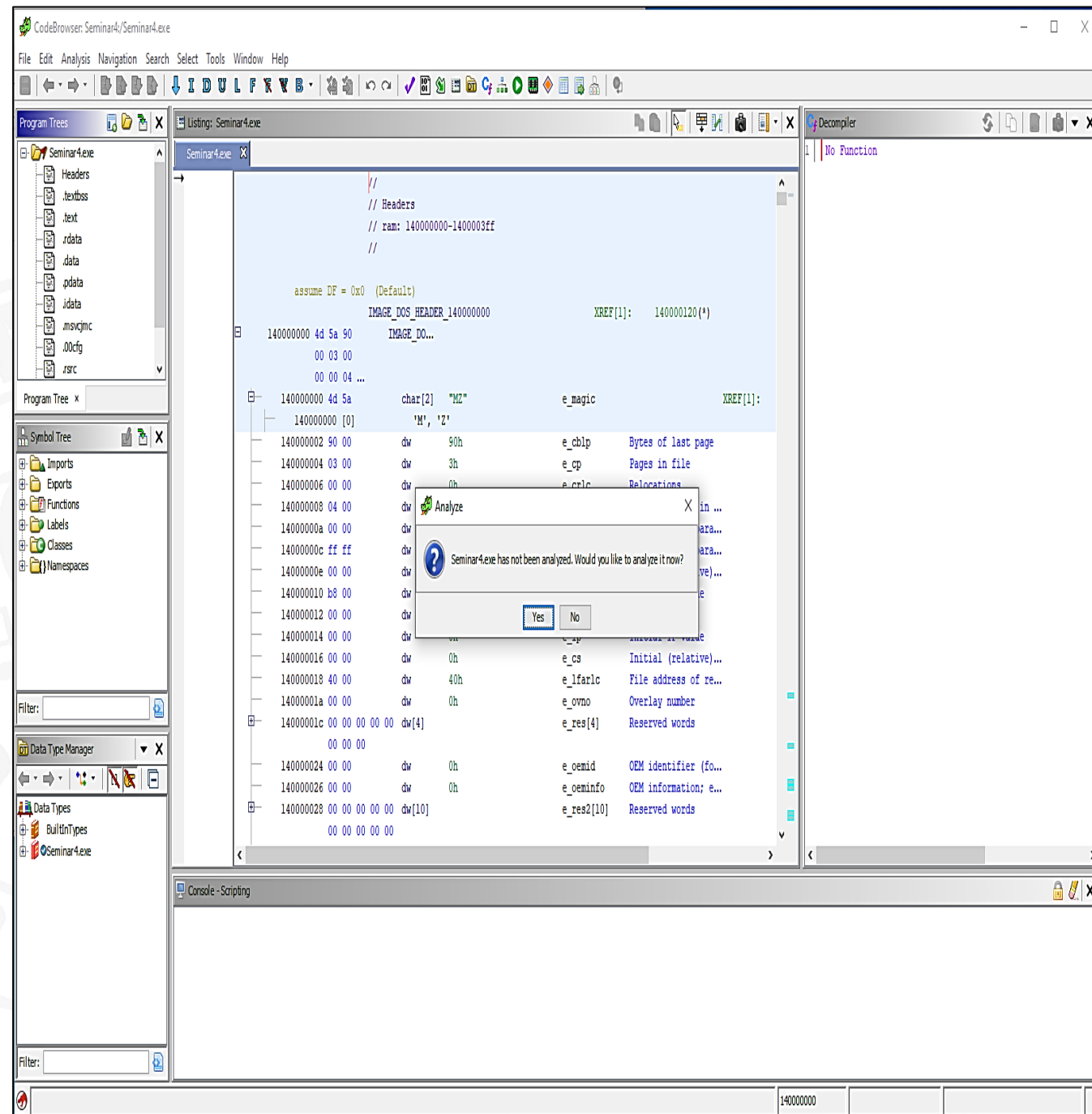


- Double-clicking the imported file in the project window runs the **Code Browser**

- 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”

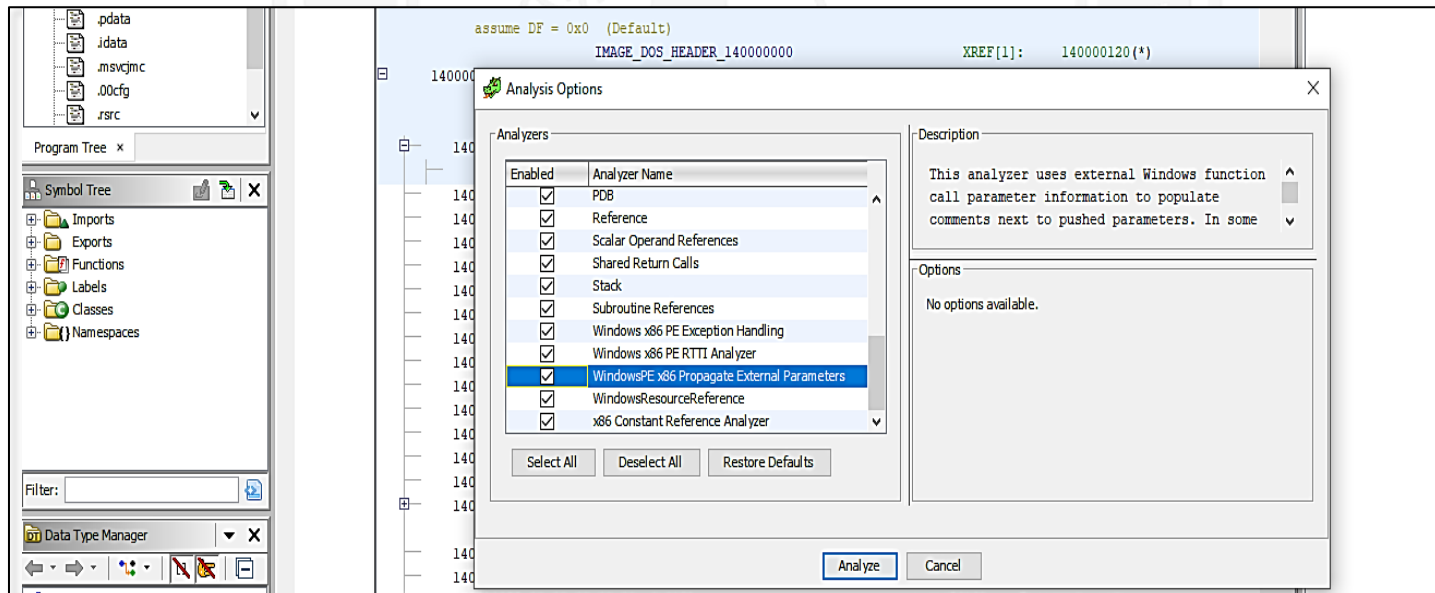
- It writes arguments of functions in the comments
- This is convenient for better understand the analyzed code



ANALYZING AN EXECUTABLE



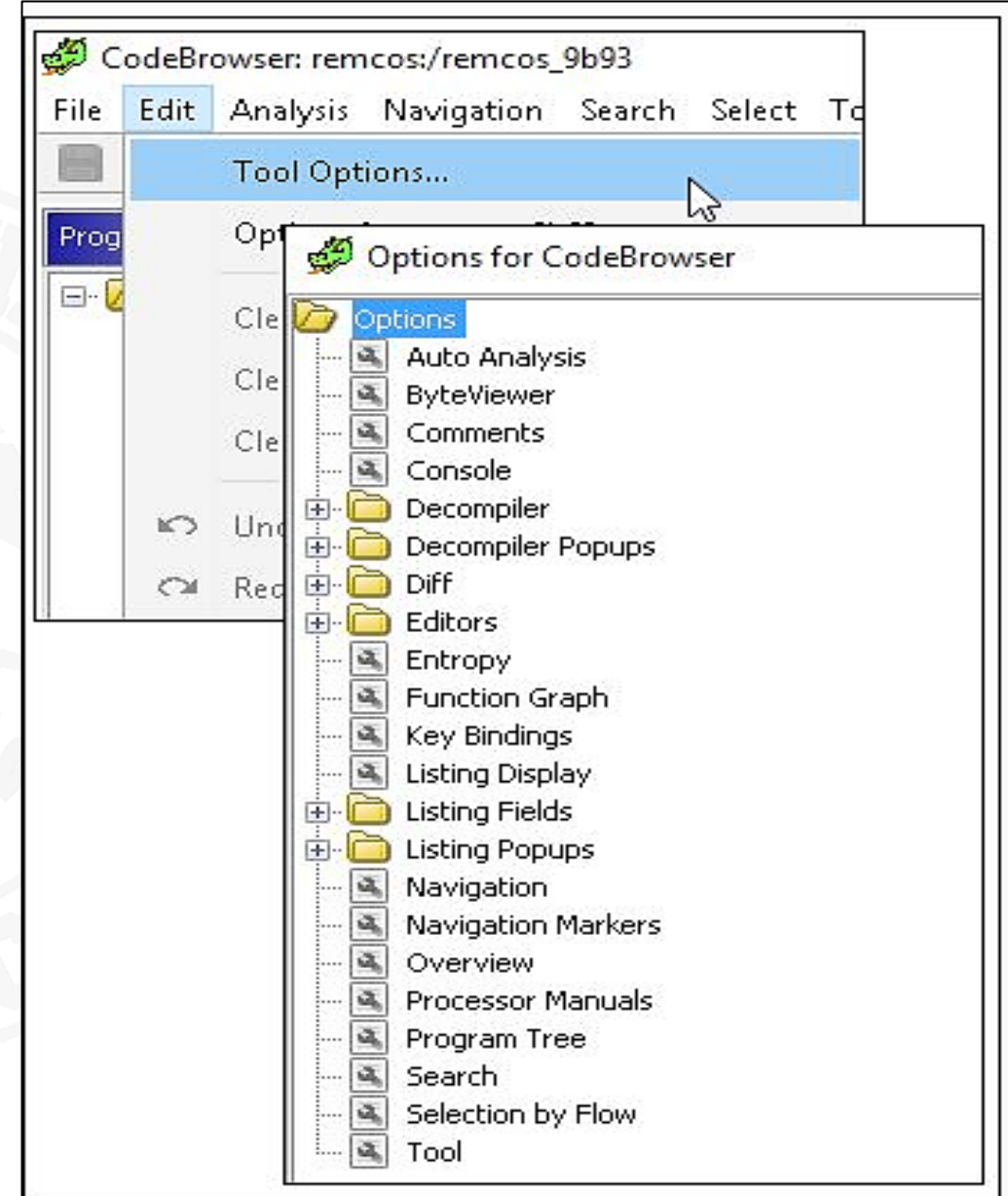
- To start the analysis, let Ghidra collect information about our `.exe`
 - As we are in Windows and a `.pdb` is provided, Ghidra should load it and complete the exe information
 - However, it could complain about not finding **the DIA SDK** that is used to fully load `.pdb` files
 - This SDK is installed 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



- 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](#)

INSPECTING EXECUTABLE DATA

Analyzing strings within the executable



Departamento de Informática
Universidad de Oviedo



Universidad de Oviedo

INVESTIGATING STRING REFERENCES



- 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



Computer Security

- 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?

Location	String Value	String Representation	Data Type
140000298			char[8]
1400002c0			char[8]
1400002e8			char[8]
140000310			char[8]
140000338			char[8]
140000360			char[8]
140023de0	_Lock	"_Lock"	ds
140023fe0	_New_array	"_New_array"	ds
1400240e0	_Alloc_max	"_Alloc_max"	ds
140024160	_Masked	"_Masked"	ds
1400241e0	_Lock	"_Lock"	ds
1400241f0	_Psave_guard	"_Psave_guard"	ds
140024300	login	"login"	ds
140024444	_Meta	"_Meta"	ds
140024500	_New_ptr	"_New_ptr"	ds
140024580	_New_ptr	"_New_ptr"	ds
140024618	Unknown exception	"Unknown exception"	ds
140024670	bad array new length	"bad array new length"	ds
140024690	invalid argument	"invalid argument"	ds
1400246b0	C:\Program Files (x8...	"C:\Program Files (x...	ds
140024730	C:\Program Files (x8...	u"C:\Program Files (...	unicode
140024830	std::_Adjust_manual...	u"std::_Adjust_manu...	unicode
140024888	"invalid argument"	u"\"invalid argument\""	unicode
1400248d8	bad cast	"bad cast"	ds
1400248e8	testUser	"testUser"	ds
1400248f8	test123...	"test123..."	ds
140024908	Login:	"Login:"	ds
140024918	Password:	"Password: "	ds
140024928	Invalid login and/or p...	"Invalid login and/or ...	ds
140024950	invalid string position	"invalid string position"	ds
140024970	string too long	"string too long"	ds
140024990	std::_Allocate_manu...	u"std::_Allocate_ma...	unicode



HIDDEN? DATA WITHIN AN EXECUTABLE

- 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!**

```
C:\Users\redon\source\repos\Seminar4\Debug\Seminar4.exe
Welcome to the SSINC Management Application!
Login: root
Password:
a
Invalid login and/or password.
Login: testUser
Password: test123...
Please wait while the application is loading...
Loading sucessfull!
This is the content of the clipboard!
This is the content of the clipboard!
This is the content of the clipboard!
This is the content of the clipboard!
EMPLOYEE MANAGEMENT
-----
1. Add an employee
2. Modify an employee
3. Delete an employee
4. Exit
```

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

[< Go to Index](#)

[Decompiling >](#)

[Data >](#)

[Code >](#)

INSPECTING EXECUTABLE CODE AND BEHAVIOR

See what our program does



Departamento de Informática
Universidad de Oviedo



Universidad de Oviedo

“FOLLOWING” A PROGRAM



● 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 (it is like the typical **main** function of languages like Java or C#)

● We may find **thunked functions**

- Functions used to **inject additional calculations** into another ones
- 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 phase
- Most of the times we can just **follow them ignoring their contents**

```
Symbol Tree
+ .F...
+ .g...
+ .init...
+ .f...
+ .se...
+ .w...
+ .x...
+ 'scalar_deleting_destructor'
+ 'configure_argv'
+ entry
+ exit...
+ Fl...

Disassembly
*****
* THUNK FUNCTION
*****
thunk undefined __fastcall entry(void)
Thunked-Function: FUN_14001b4d0
AL:1 <RETURN>
entry XREF[2]: Entry
140011046 e9 85 a4... JMP FUN_14001b4d0
14001104b e9 ?? 79h
14001104c 20 ?? 20h
```

Double click

```
Disassembly
*****
* FUNCTION
*****
undefined __fastcall FUN_14001b4d0(void)
AL:1 <RETURN>
FUN_14001b4d0 XREF[1]:
14001b4d0 48 83 ec... SUB RSP, 0x28
14001b4d4 e8 07 fd... CALL FUN_14001b1e0
14001b4d9 48 83 c4... ADD RSP, 0x28
14001b4dd c3 RET
14001b4de cc ?? Cch

Decompile: FUN_14001b4d0 - (Seminar4.exe)
1 void FUN_14001b4d0(void)
2
3
4 {
5     FUN_14001b1e0();
6     return;
7 }
8
```

Double click

```
Disassembly
undefined __fastcall FUN_14001b1e0(void)
AL:1 <RETURN>
FUN_14001b1e0
14001b1e0 48 83 ec... SUB RSP, 0x28
14001b1e4 e8 06 65... CALL thunk_FUN_14001d0a0
14001b1e9 e8 12 00... CALL FUN_14001b200
14001b1ee 48 83 c4... ADD RSP, 0x28
14001b1f2 c3 RET

Decompile: FUN_14001b1e0 - (Seminar4.exe)
1 void FUN_14001b1e0(void)
2
3 {
4     thunk_FUN_14001d0a0();
5     FUN_14001b200();
6     return;
7 }
8
```

REVERSING IS NOT EASY... AND WE WILL FIND OBSTACLES

- Ghidra can try to make it easier for you to see the value of a variable
 - By interpreting and displaying it by its type
 - Which sometimes leads to "easter eggs", like this image
- Companies know that someone is going to decompile their programs
- Programs often have obfuscation or counter-reversing techniques
 - They make your job harder because they don't want you to see how the program does certain things
 - E.g.: Check that it is not a pirated copy 😊
 - Therefore, don't despair if there are very difficult to understand things

```

001b6835 30      ??      30h      0
001b6836 20      ??      20h      0
001b6837 35      ??      35h      5
001b6838 30      ??      30h      0
001b6839 20      ??      20h      0
001b683a 31      ??      31h      1
001b683b 36      ??      36h      6
001b683c 20      ??      20h      0
001b683d 31      ??      31h      1
001b683e 00      ??      00h      0
                                JPEG_001b683f+23300
                                JPEG_001b683f+23861
001b683f ff d8 ff      JPG
                                e0 00 10
                                4a 46 49 ...

001bc742 00      ??      00h
                                GIF_001bc743+385
001bc743 47 49 46      GIF
                                38 39 61
                                97 00 28 ...

001bc973 00      ??      00h
001bc974 b7      ??      B7h
001bc975 c8      ??      C8h
001bc976 1b      ??      1Bh
001bc977 00      ??      00h
001bc978 a1      ??      A1h
001bc979 c8      ??      C8h
001bc97a 1b      ??      1Bh
001bc97b 00      ??      00h
001bc97c 8b      ??      8Bh

```

XREF[0,2]: FUN_000291f8:0002
FUN_0002a248:0002



XREF[0,1]: FUN_000291f8:0002



? -> 001bc8f
? -> 001bc8e
? -> 001bc8d

Decompiling

Obtaining readable source code



“FOLLOWING” A PROGRAM: DECOMPILING

- Following the call sequence, we will arrive to functions with the actual program code
- 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

```

ulonglong __fastcall FUN_14001b200(void)
ulonglong     RAX:8     <RETURN>
undefined8     Stack[-0x18...local_18]     XREF[2]: 14001b2dc(W), 14001b2eb(R)
undefined8     Stack[-0x20...local_20]     XREF[2]: 14001b2d2(W), 14001b2d7(R)
undefined8     Stack[-0x30...local_30]     XREF[4]: 14001b2fb(W)
undefined8     Stack[-0x38...local_38]

```

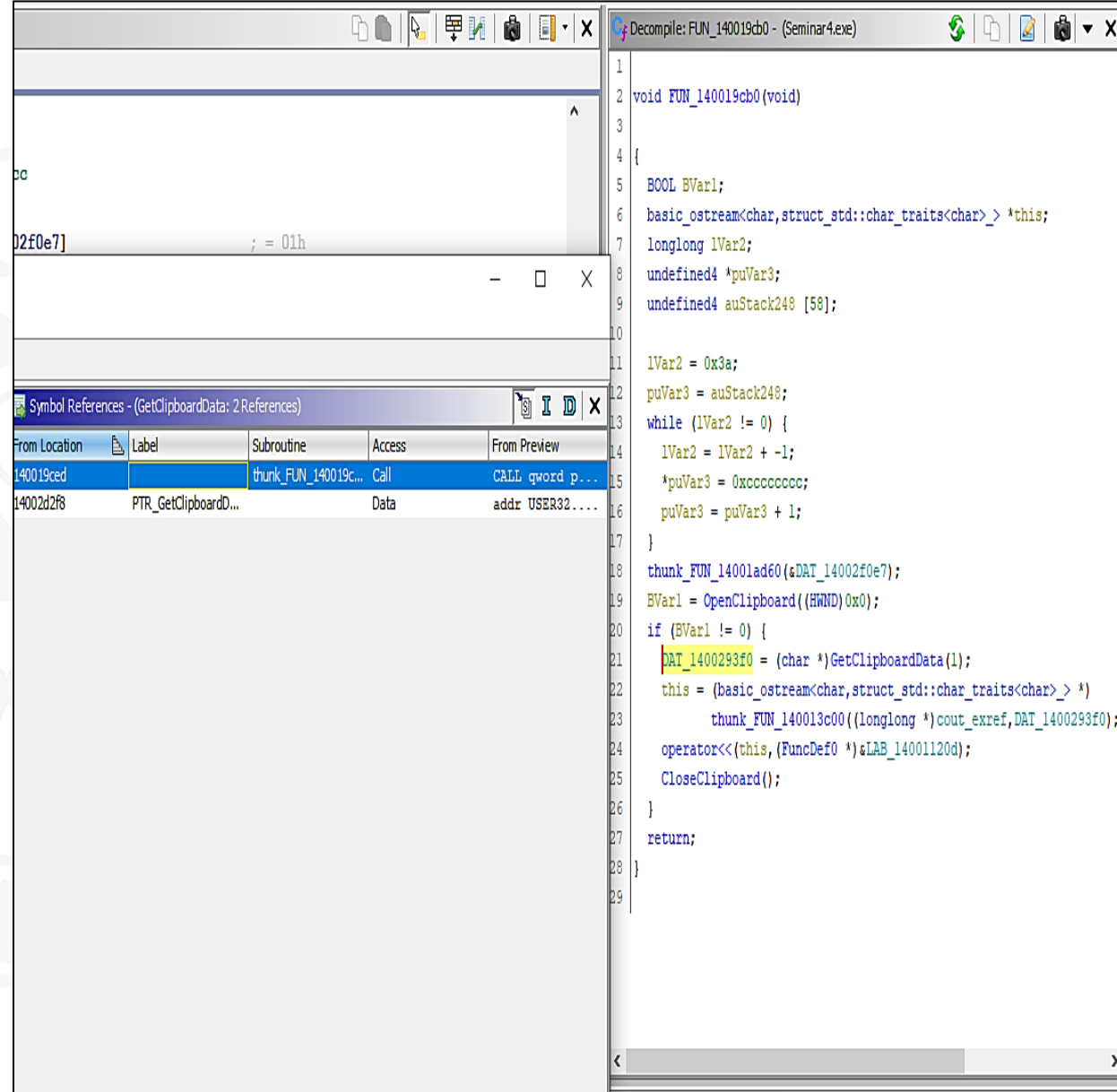
```

Decompile: FUN_14001b200 - (Seminar4.exe)
1
2 /* WARNING: Function: _guard_dispatch_icall replaced with inject
3 /* WARNING: Globals starting with '_' overlap smaller symbols at
4
5 ulonglong FUN_14001b200(void)
6
7 {
8     bool bVar1;
9     int iVar2;
10    uint _Code;
11    ulonglong uVar3;
12    code **ppcVar4;
13    code **ppcVar5;
14
15    uVar3 = __srt_initialize_crt(1);
16    if ((uVar3 & 0xff) == 0) {
17        __srt_fastfail(7);
18    }
19    bVar1 = false;
20    uVar3 = __srt_acquire_startup_lock();
21    if (_DAT_140029418 == 1) {
22        __srt_fastfail(7);
23    }
24    else {
25        if (_DAT_140029418 == 0) {
26            _DAT_140029418 = 1;
27            iVar2 = _initterm_e(&DAT_140023558,&DAT_140023888);
28            if (iVar2 != 0) {
29                return 0xff;
30            }
31            _initterm();
32            _DAT_140029418 = 2;
33        }
34    }
35

```

VIEW DECOMPILE OUTPUT

- 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



- 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 reference on the right-hand side
 - This populates the Listing view with the appropriate disassembly (ASM + C code)

INVESTIGATING AN API REFERENCE



Computer Security

```
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]
```

```
9  undefined4 auStack248 [58];
10
11  lVar2 = 0x3a;
12  puVar3 = auStack248;
13  while (lVar2 != 0) {
14      lVar2 = lVar2 + -1;
15      *puVar3 = 0xffffffff;
16      puVar3 = puVar3 + 1;
17  }
18  thunk_FUN_14001ad60(&DAT_14002f0e7);
19  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...	Offset 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 contained inside source code



DATA ANNOTATION



Computer Security

● 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...**"

The screenshot displays a debugger's listing view with assembly code and a right-click context menu.

Assembly Listing:

LAB_140019ce8			
140019ce8	b9 01 00...	MOV	ECX, 0
140019ced	ff 15 05...	CALL	qword p
140019cf3	48 89 05...	MOV	qword p
140019cfa	48 8b 15...	MOV	RDX, q
140019d01	48 8b 0d...	MOV	RCX, q
140019d08	e8 73 75...	CALL	thunk_1
140019d0d	48 8d 15...	LEA	RDX, [
140019d14	48 8b c8	MOV	RCX, R
140019d17	ff 15 53...	CALL	qword p
140019d1d	ff 15 cd...	CALL	qword p

LAB_140019d23			
140019d23	48 8d a5...	LEA	RSP=>1c
140019d2a	5f	POP	RDI
140019d2b	5d	POP	RBP

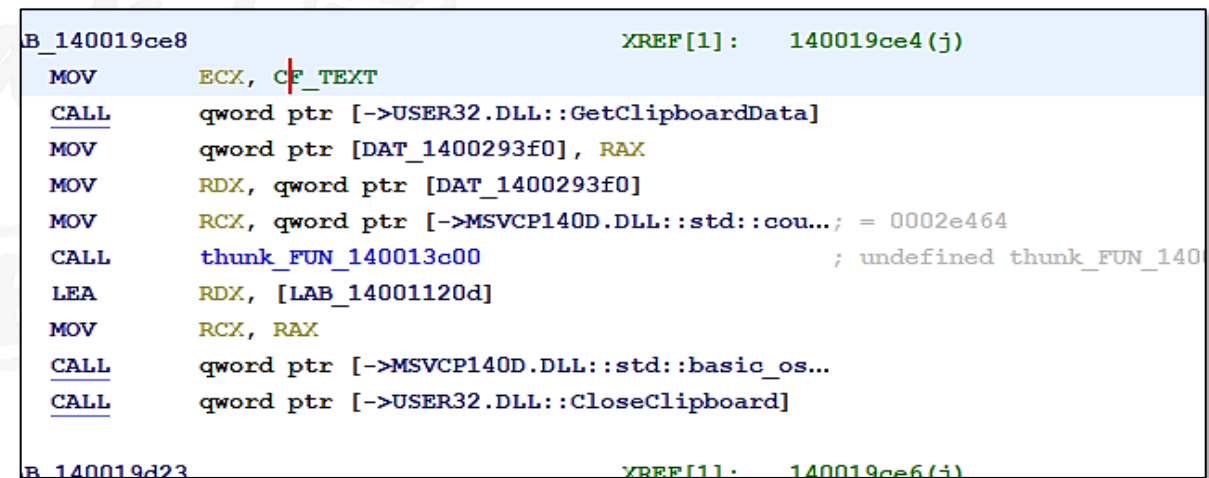
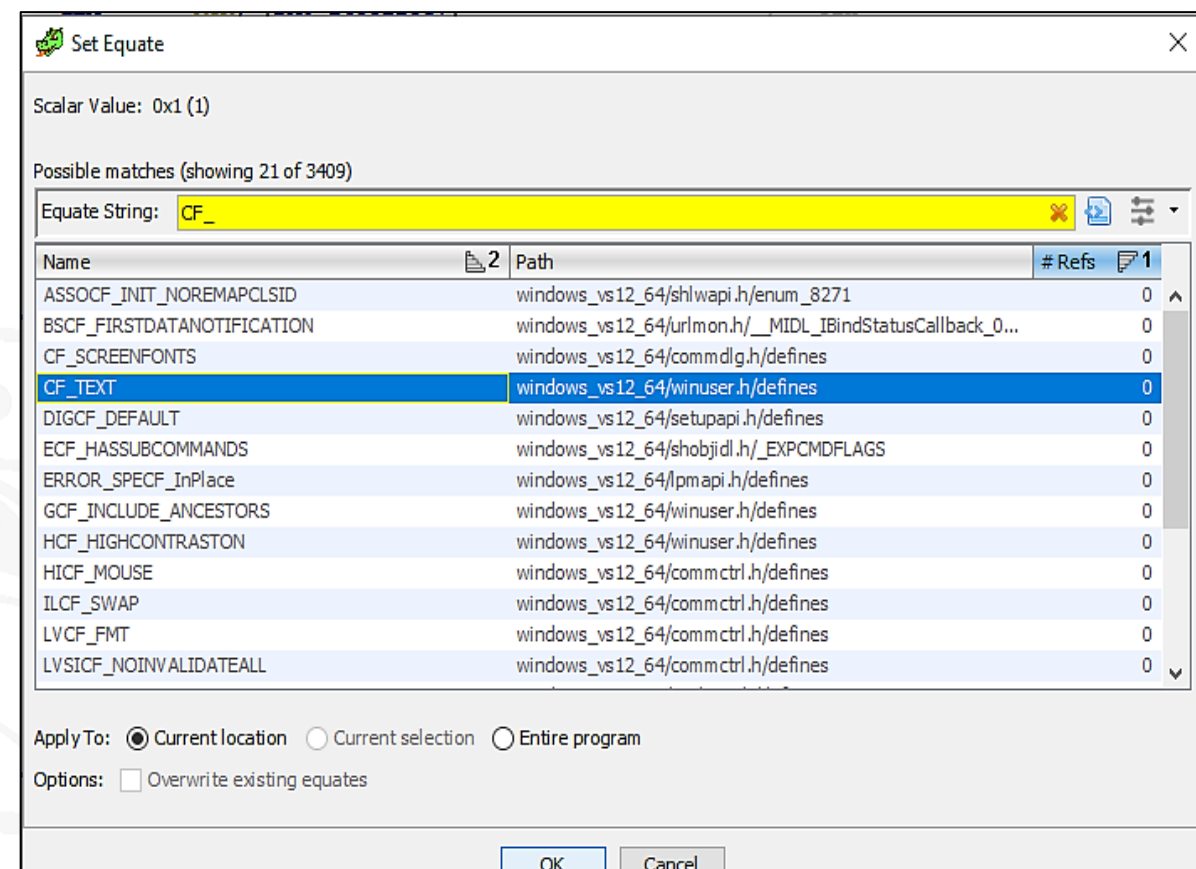
Right-click Context Menu:

- Copy Special...
- Paste (Ctrl+V)
- Comments >
- Instruction Info...
- Patch Instruction (Ctrl+Shift+G)
- Processor Manual...
- Processor Options...
- Create Function (F)
- Create Thunk Function
- Function >
- Compare Selected Functions...
- Add Label... (L)
- Show Label History... (H)
- Clear Register Values...
- Set Register Values... (Ctrl+R)
- Colors >
- Convert >
- Set Equate... (E)

DATA ANNOTATION



- 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!





LOCATING CODE FROM STRING REFERENCES

- 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

Location	String Value	String Representation	Data Type
140000298			char[8]
1400002c0			char[8]
1400002e8			char[8]
140000310			char[8]
140000338			char[8]
140000360			char[8]
140023de0	_Lock	"_Lock"	ds
140023fe0	_New_array	"_New_array"	ds
1400240e0	_Alloc_max	"_Alloc_max"	ds
140024160	_Masked	"_Masked"	ds
1400241e0	_Lock	"_Lock"	ds
1400241f0	_Psave_guard	"_Psave_guard"	ds
140024300	login	"login"	ds
140024444	_Meta	"_Meta"	ds
140024500	_New_ptr	"_New_ptr"	ds
140024580	_New_ptr	"_New_ptr"	ds
140024618	Unknown exception	"Unknown exception"	ds
140024670	bad array new length	"bad array new length"	ds
140024690	invalid argument	"invalid argument"	ds
1400246b0	C:\Program Files (x8...	"C:\Program Files (x...	ds
140024730	C:\Program Files (x8...	u"C:\Program Files (...	unicode
140024830	std::_Adjust_manual...	u"std::_Adjust_manu...	unicode
140024888	"invalid argument"	u"\"invalid argument\""	unicode
1400248d8	bad cast	"bad cast"	ds
1400248e8	testUser	"testUser"	ds
1400248f8	test123...	"test123..."	ds
140024908	Login:	"Login: "	ds
140024918	Password:	"Password: "	ds
140024928	Invalid login and/or p...	"Invalid login and/or ...	ds
140024950	invalid string position	"invalid string position"	ds
140024970	string too long	"string too long"	ds
140024990	std::_Allocate_manu...	u"std::_Allocate_ma...	unicode

Filter:

LOCATING CODE FROM STRING REFERENCES



Computer Security

Listing: Seminar4.exe

*Seminar4.exe

1400248e6 00 ?? 00h
1400248e7 00 ?? 00h

s_testUser_1400248e8 XREF[1]: 140011faa(*)

1400248e8
1400248f1
1400248f2
1400248f3
1400248f4
1400248f5
1400248f6
1400248f7
1400248f8
140024903
140024904
140024905
140024906
140024907
140024908
140024910
140024911
140024912
140024913
140024914

Bookmark... Ctrl+D
Clear Code Bytes C
Clear With Options...
Clear Flow and Repair...
Copy Ctrl+C
Copy Special...
Paste Ctrl+V
Comments
Data
Instruction Info...
Patch Instruction Ctrl+Shift+G
Processor Manual...
Processor Options...
Function
Compare Selected Functions...
Add Label... L
Show Label History... H

Defined Strings - 229 items

Location	String Value	String Representation	Data Type
140000298			char[8]
1400002c0			char[8]
1400002e8			char[8]
140000310			char[8]
140000338			char[8]
140000360			char[8]
140023de0	_Lock	"_Lock"	ds
140023fe0	_New_array	"_New_array"	ds
1400240e0	_Alloc_max	"_Alloc_max"	ds
140024160	_Masked	"_Masked"	ds
1400241e0	_Lock	"_Lock"	ds
1400241f0	_Psave_guard	"_Psave_guard"	ds
140024300	login	"login"	ds
140024444	_Meta	"_Meta"	ds
140024500	_New_ptr	"_New_ptr"	ds
140024580	_New_ptr	"_New_ptr"	ds
140024618	Unknown exception	"Unknown exception"	ds

References to s_testUser_1400248e8 - 1 locations [CodeBrowser: Seminar4/Seminar4.exe]

Help

References to s_testUser_1400248e8 - 1 locations

Location	Label	Code Unit	Context
140011faa		LEA RDX, [s_testUser_1400248e8]	DATA

Filter:

XREF[1]: 140023330(*)

140011f9e 48 8d 0d... LEA RCX, [DAT_14002f027] ; = 01h
140011fa5 e8 a5 f1... CALL thunk_FUN_14001ad60 ; undefined thunk_FUN_14001a.
140011faa 48 8d 15... LEA RDX, [s_testUser_1400248e8] ; = "testUser"
140011fb1 48 8d 0d... LEA RCX, [DAT_140029380]
140011fb8 e8 96 f7... CALL thunk_FUN_140015a90 ; undefined thunk_FUN_140015.
140011fbd 48 8d 0d... LEA RCX, [LAB_140021900]
140011fc4 e8 98 f3... CALL FID_conflict:atexit ; int FID_conflict:atexit(vo.
140011fc9 48 8d a5... LEA RSP, [RBP + 0xc8]
140011fd0 5f POP RDI

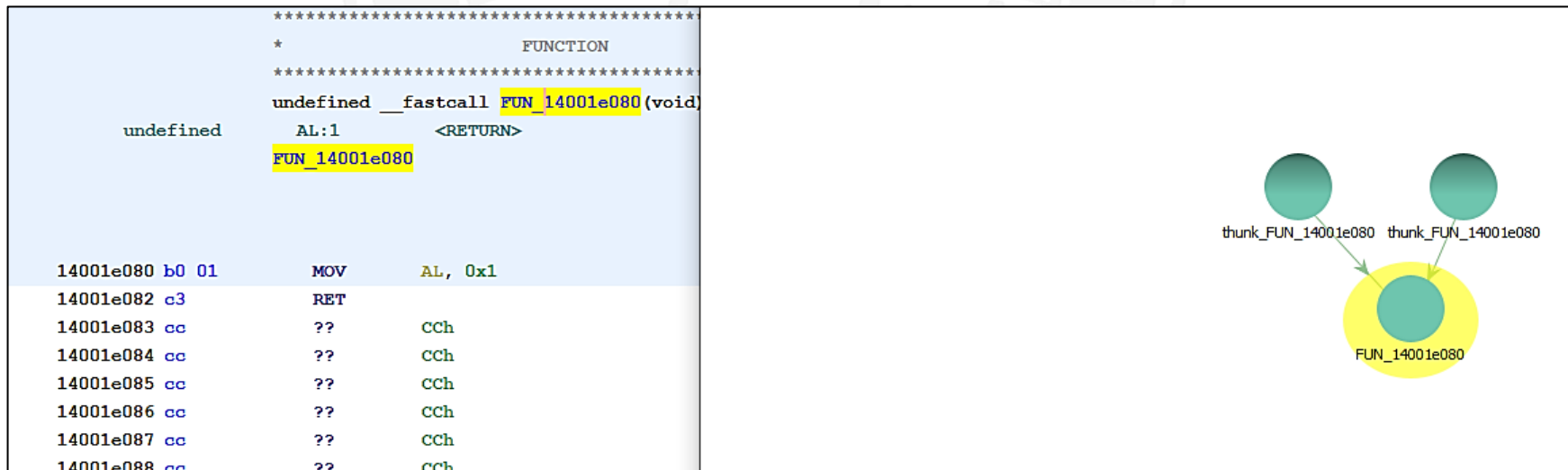
Call graphs

Tools to help us understanding the source code



FUNCTION CALL GRAPHS

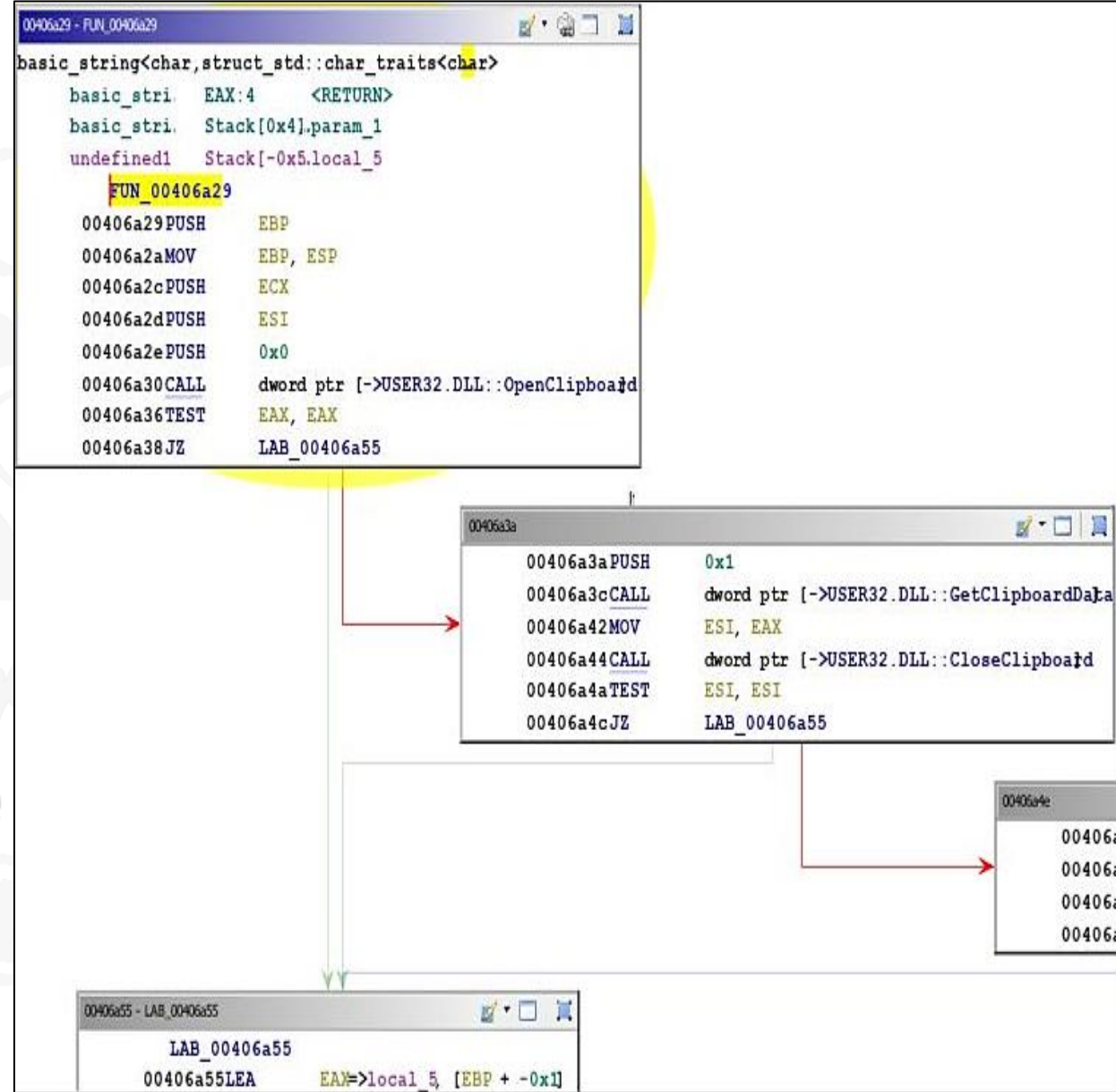
- Once a reference is identified, we can also analyze the call sequence that ends in the code that uses it
- To do this, we can go to the beginning of the function that uses the string and click on **Window- Function call graph**
 - This displays the relationships between functions and provides a **high-level 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



- 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**
 - The arrows represent **conditional and unconditional jumps**



UNDERSTANDING ASSEMBLY



Computer Security

- 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



Computer Security

- 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

```
Listing: Seminar4.exe
*Seminar4.exe X
undefined1 Stack[-0xd8...local_d8] XREF[1]: 140019cba(*)
FUN_140019cb0 XREF[1]: thunk_FUN_140019cb0:14001146...
thunk_FUN_140019cb0:14001146...
; Hwnd hWndNewOwner for Open...

140019cb0 40 55 PUSH RBP
140019cb2 57 PUSH RDI
140019cb3 48 81 ec... SUB RSP, 0xe8
140019cba 48 8d 6c... LEA RBP=>local_d8, [RSP + 0x20]
140019cbf 48 8b fc MOV RDI, RSP
140019cc2 b9 3a 00... MOV ECX, 0x3a
140019cc7 b8 cc cc... MOV EAX, 0xffffffff
140019ccc f3 ab STOSD.R.. RDI
140019cce 48 8d 0d... LEA RCX, [DAT_14002f0e7] ; = 01h
140019cd5 e8 75 74... CALL thunk_FUN_14001ad60 ; undefined thunk_FUN_14001a...
140019cda 33 c9 XOR ECX, ECX
140019cde 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, CF_TEXT
140019ced ff 15 05... CALL qword ptr [->USER32.DLL::GetClipboardData]
140019cf3 48 89 05... MOV qword ptr [DAT_1400293f0], RAX
140019cfa 48 8b 15... MOV RDX, qword ptr [DAT_1400293f0]
140019d01 48 8b 0d... MOV RCX, qword ptr [->MSVCPL40D.DLL::std::cou...; = 0002e464
140019d08 e8 73 75... CALL thunk_FUN_140013c00 ; undefined thunk_FUN_140013...
140019d0d 48 8d 15... LEA RDX, [LAB_14001120d]
140019d14 48 8b c8 MOV RCX, RAX
```

UNDERSTANDING ASSEMBLY



Computer Security

Listing: Seminar4.exe

*Seminar4.exe

undefined1 Stack[-0xd8...local_d8
FUN_140019cb0

XREF[1]: 140019cba(*)
XREF[1]: thunk_FUN_140019cb0:14001146...
thunk_FUN_140019cb0:14001146...
; HWND hWndNewOwner for Open...

140019cb0 40 55 PUSH RBP
140019cb2 57 PUSH RDI
140019cb3 48 81 ec... SUB RSP, 0xe8
140019cba 48 8d 6c... LEA RBP=>local_d8, [RSP + 0x20]
140019cbf 48 8b fc MOV RDI, RSP
140019cc2 b9 3a 00... MOV ECX, 0x3a
140019cc7 b8 cc cc... MOV EAX, 0xffffffff
140019ccc f3 ab STOSD.R... RDI
140019cce 48 8d 0d... LEA RCX, [DAT_14002f0e7] ; = 01h
140019cd5 e8 75 74... CALL thunk_FUN_14001ad60 ; undefined thunk_FUN_14001a...
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
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, CF TEXT
140019ced ff 15 05... CALL qword ptr [->USER32.DLL::GetClipboardData]
140019cf3 48 89 05... MOV qword ptr [DAT_1400293f0], RAX
140019cfa 48 8b 15... MOV RDX, qword ptr [DAT_1400293f0]
140019d01 48 8b 0d... MOV RCX, qword ptr [->MSVCPP140D.DLL::std::cou...; = 0002e46
140019d08 e8 73 75... CALL thunk_FUN_140013c00 ; undefined
140019d0d 48 8d 15... LEA RDX, [LAB_14001120d]
140019d14 48 8b c8 MOV RCX, RAX

Decompile: FUN_140019cb0 - (Seminar4.exe)

1
2 void FUN_140019cb0(void)
3
4 {
5 BOOL BVar1;
6 basic_ostream<char,struct_std::char_traits<char>_> *this;
7 longlong lVar2;
8 undefined4 *puVar3;
9 undefined4 auStack248 [58];
10
11 lVar2 = 0x3a;
12 puVar3 = auStack248;
13 while (lVar2 != 0) {
14 lVar2 = lVar2 + -1;
15 *puVar3 = 0xffffffff;
16 puVar3 = puVar3 + 1;
17 }
18 thunk_FUN_14001ad60(&DAT_14002f0e7);
19 BVar1 = OpenClipboard((HWND)0x0);
20 if (BVar1 != 0) {
21 DAT_1400293f0 = (char *)GetClipboardData(1);
22 this = (basic_ostream<char,struct_std::char_traits<char>_> *)
23 thunk_FUN_140013c00((longlong *)cout_exref,DAT_1400293f0);
24 operator<<(this,(FuncDef0 *)&LAB_14001120d);
25 CloseClipboard();
26 }
27 return;
28 }
29

MORE INFORMATION



- 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_NTE9cr180Pphn82WS8gVv-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/>
 - Repository with lot of reversing resources: <https://github.com/xiosec/Reverse-engineering>

[< Go to Index](#)

REVERSING PROGRAMS CREATED IN HIGH-LEVEL LANGUAGES

Inspecting .Net or Java programs



Departamento de Informática
Universidad de Oviedo



Universidad de Oviedo

REVERSING PROGRAMS CREATED IN HIGH-LEVEL LANGUAGES



Computer Security

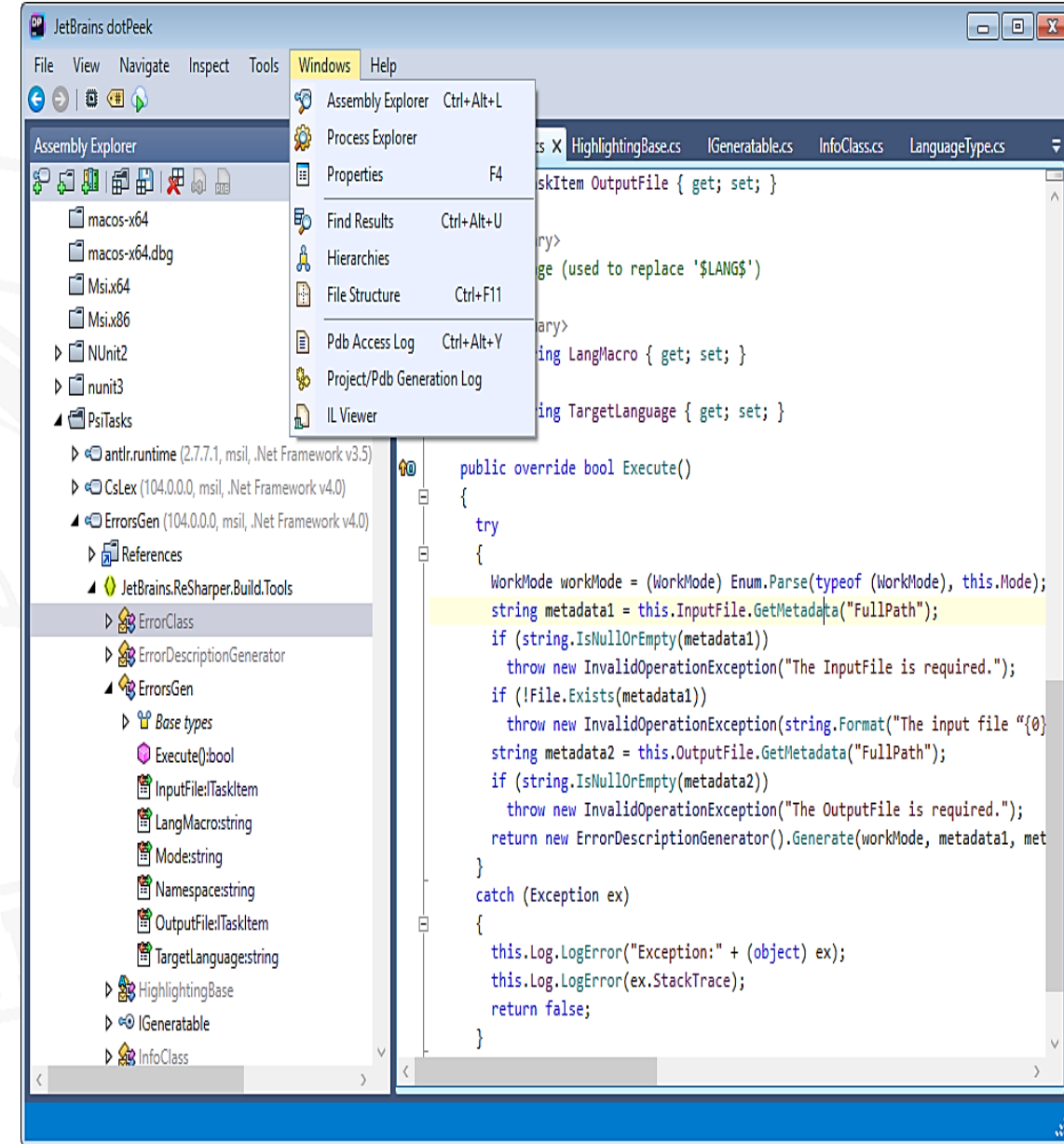
- We have seen a tool for decompiling and analyzing programs made in low/medium level languages (C/C++)
- The same is possible with those coded in C#, Java, or other high-level languages
 - So, you can analyze programs created in courses like [TPP](#), [EDI](#), [MP](#), etc. even if you don't have the 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



Computer Security

- 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

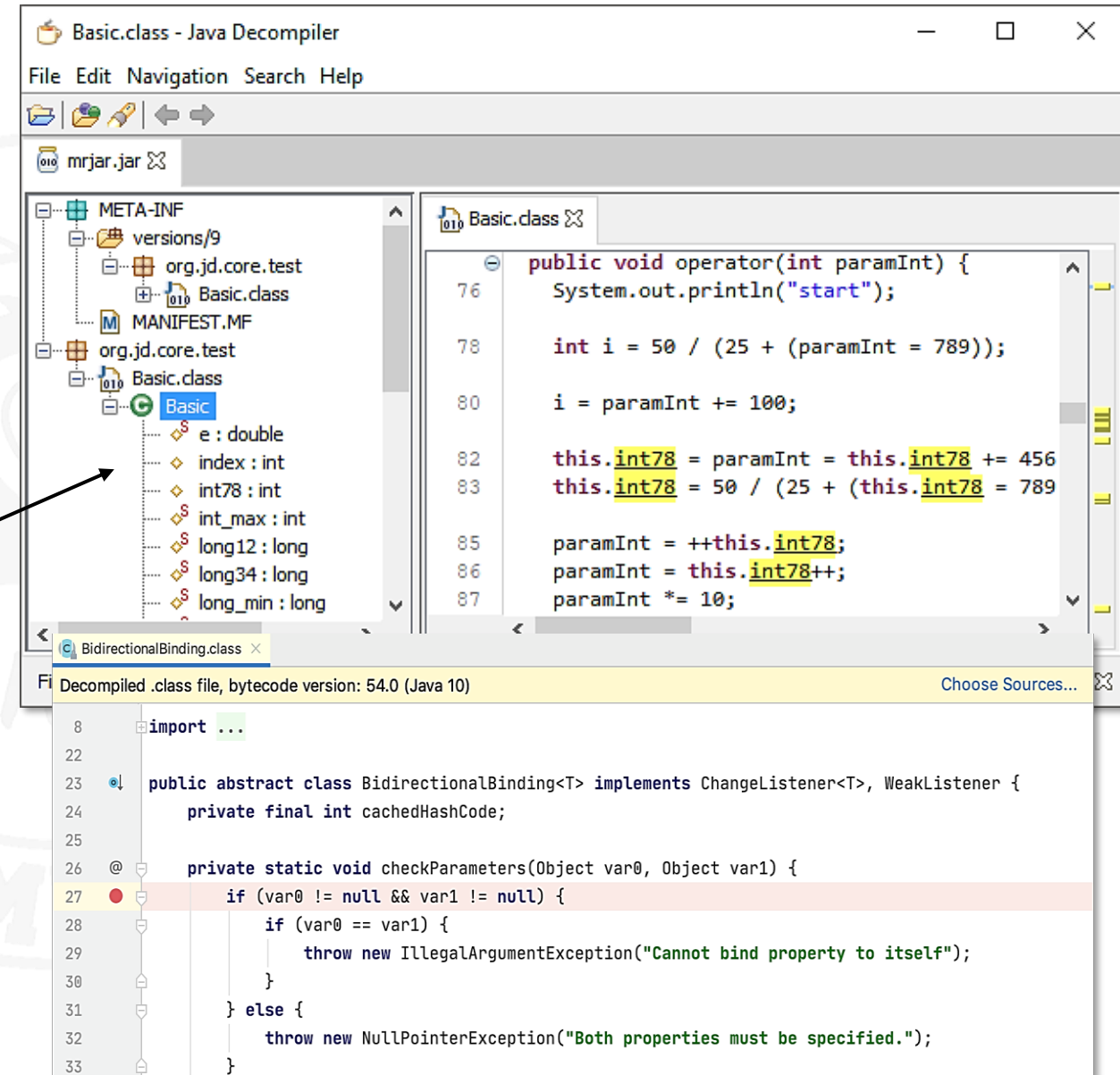


ANALYZING JAVA PROGRAMS



Computer Security

- 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](#)

AUTOMATED EXECUTABLE INSPECTION TOOLS

Automated reversing

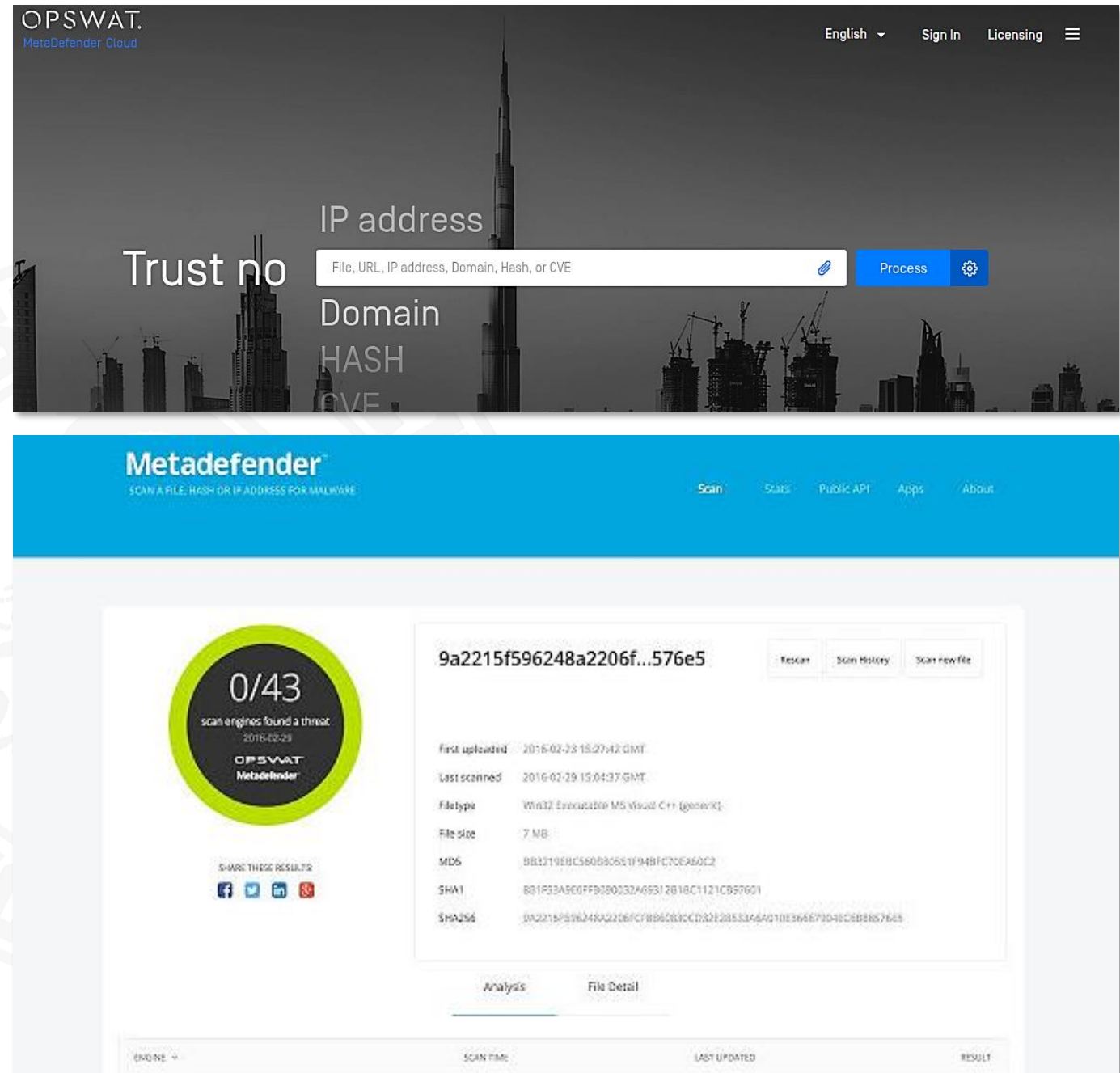


Departamento de Informática
Universidad de Oviedo



Universidad de Oviedo

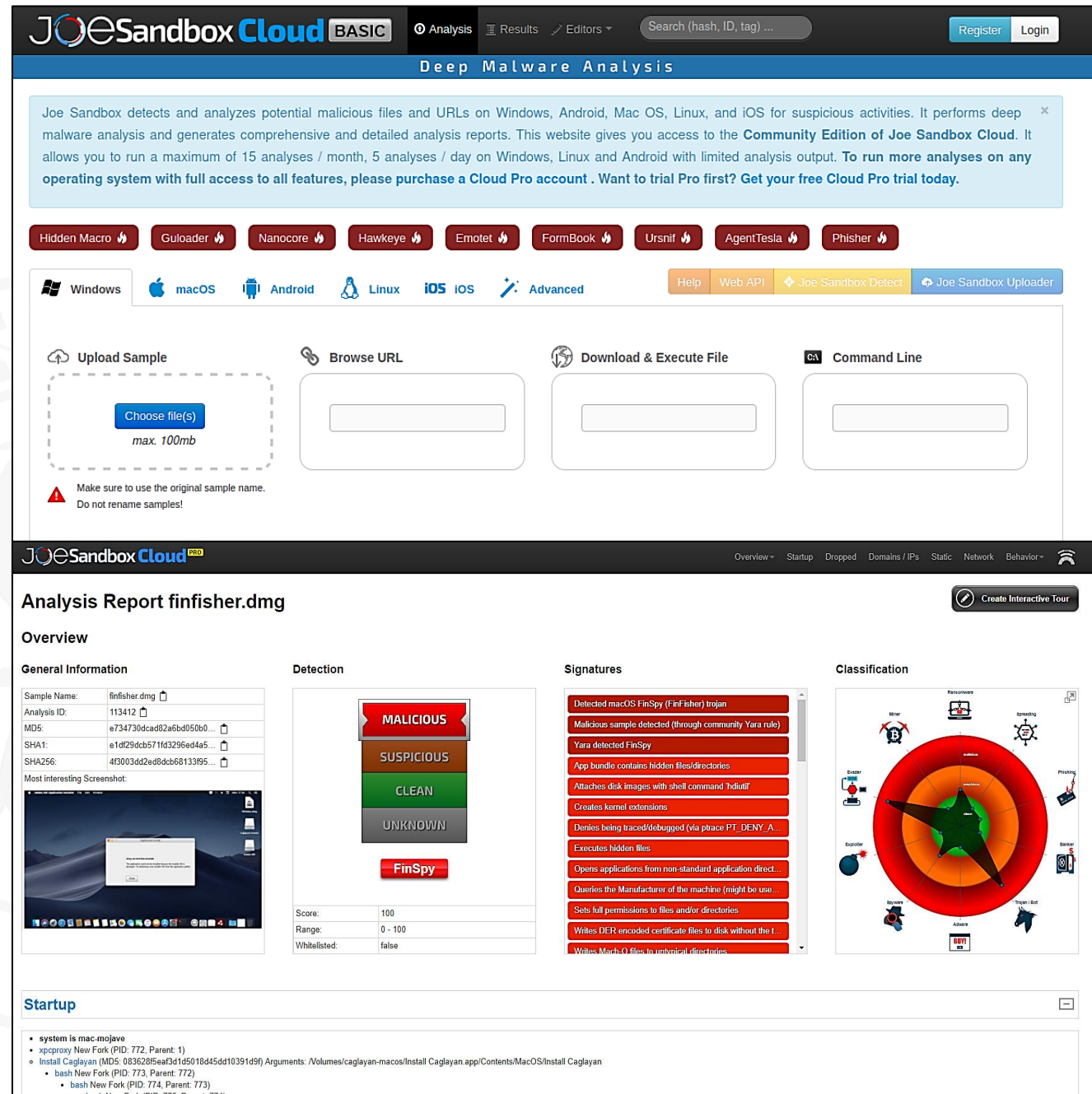
- Many times, we will only want to know if an executable behaves strangely to avoid malware
 - In that case, doing a manual analysis may be too much
- There are tools that allow us to automate this process
 - We can use one that checks if an executable is detected as malware
- Metadefender is a tool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



The screenshot displays the Metadefender Cloud interface. At the top, there's a navigation bar with 'English', 'Sign In', and 'Licensing' links. Below this, a large banner area features a city skyline background and a search bar labeled 'Trust no IP address'. The search bar has a placeholder text 'File, URL, IP address, Domain, Hash, or CVE' and a 'Process' button. Below the banner, the main content area has a blue header with the 'Metadefender' logo and the tagline 'SCAN A FILE, HASH OR IP ADDRESS FOR MALWARE'. The main content area shows a scan result for a file with the hash '9a2215f596248a2206f...576e5'. On the left, a circular progress indicator shows '0/43' and 'scan engines found a threat'. On the right, a table lists file details: 'First uploaded', 'Last scanned', 'Filetype', 'File size', 'MD5', 'SHA1', and 'SHA256'. At the bottom, there are tabs for 'Analysis' and 'File Detail'.

ENGINE	SCAN TIME	LAST UPDATED	RESULT
0/43			

- If we want a 2nd opinion, we can use a **sandbox**
- Run a program and analyze its runtime behavior
 - Looking for "suspicious" behavior
 - System calls known as problematic, attempts to access restricted parts, etc.
 - So, we can extract conclusions about whether the program is trusted
 - Returns us a report of its actions and the "dangerousness" they have
- Although it is not free, there is a limited "Community" version
 - <https://www.joesandbox.com/#windows>



The screenshot displays the JoeSandbox Cloud interface. At the top, there's a navigation bar with 'JOESandboxCloud BASIC', 'Analysis', 'Results', and 'Editors' tabs. A search bar and 'Register'/'Login' buttons are also present. Below this is a 'Deep Malware Analysis' section with a blue banner explaining the service. A row of analysis tools (Hidden Macro, Guloader, Nanocore, Hawkeye, Emotet, FormBook, Ursnif, AgentTesla, Phisher) is shown. The main interface has tabs for Windows, macOS, Android, Linux, iOS, and Advanced. Below these are four analysis methods: Upload Sample, Browse URL, Download & Execute File, and Command Line. The 'Upload Sample' method is selected, showing a 'Choose file(s)' button and a warning to use the original sample name. Below this is the 'Analysis Report finfisher.dmg' section. It includes an 'Overview' tab with 'General Information' (Sample Name, Analysis ID, MD5, SHA1, SHA256, and a screenshot), 'Detection' (MALICIOUS, SUSPICIOUS, CLEAN, UNKNOWN, and FinSpy), 'Signatures' (a list of detected behaviors), and 'Classification' (a radar chart). The 'Startup' section at the bottom shows system information and process details.

WILL AI REPLACE REVERSING?



- Today, generative AI is being successfully experimented with to analyze executables
 - They allow us to know what an executable does in "human" language
- It will not replace 100% reversing
 - But it will make it **much more accessible** to a lot of people!
- An example is Code Insight by Virustotal
 - <https://blog.virustotal.com/2023/04/introducing-virustotal-code-insight.html>
 - <https://assets.virustotal.com/reports/2023-ai>

The screenshot displays the Virustotal Code Insight interface. At the top, a green circle with '0 / 59' indicates the community score. A green checkmark and text state: 'No security vendors and no sandboxes flagged this file as malicious'. Below this, the file's SHA-256 hash is shown: '552efb0dc7e62ded08c98d2e6355df1d27a1317c0a37aabeefd48667b7b1917b'. The file size is '413 B' and it was uploaded on '2023-04-10 11:06:13 UTC a moment ago'. The file type is identified as 'important.ps1' with a 'javascript' extension. The 'DETECTION' tab is active, showing a 'Code Insight' section with a warning icon. The insight text reads: 'This code snippet is malicious. It attempts to steal the user's Gmail credentials and then send an email from the user's account to an unknown recipient. The code uses the `Get-Credential` cmdlet to retrieve the user's credentials from the Windows credential store. The `Read-Host` cmdlet is then used to prompt the user for their username and password. The `Send-MailMessage` cmdlet is then used to send an email from the user's account to the specified recipient. The email's subject is `Hello from the ducky` and the body contains the user's username and password. The code is malicious because it attempts to steal the user's Gmail credentials and then send an email from the user's account to an unknown recipient. This could be used to impersonate the user or to send spam or phishing emails.'

Antivirus products do not detect anything in the executable, but the AI that analyzes it determines that it's a malicious executable by understanding its behavior. Awesome, right? It also explains it in plain words, so that anyone who has no experience in reversing can understand it

SELF-EVALUATION ACHIEVEMENT LIST: SEMINAR 4

- Understand the 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



SEMINAR 4

INTRODUCTION TO REVERSING

