SEMINARIO 4
INTRODUCCIÓN AL REVERSING

Una primera aproximación al análisis de programas

José Manuel Redondo López Proyecto *"S-81 Isaac Peral"* v4.0

Fuente: Bing Chat AI

Departamento de Informática
Universidad de Oviedo

Universidad de Oviedo

Logo: @creative_vanesa (*Instagram*)

ÍNDICE

< Ir al Índice

Fuente: Bing Chat AI

INTRODUCCIÓN

¿Qué es Ghidra?

Departamento de Informática
Universidad de Oviedo

Universidad de Oviedo

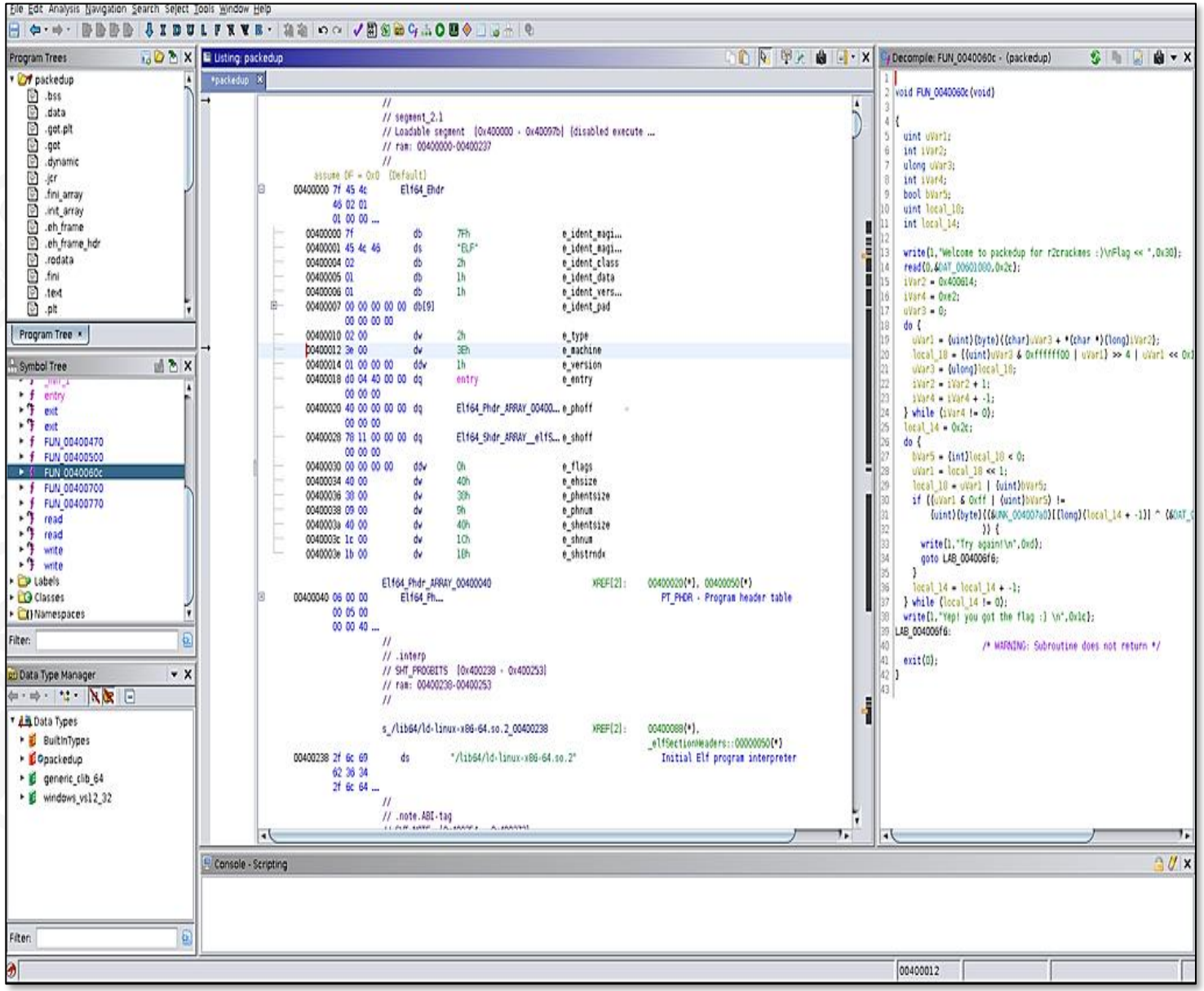
Introducción

- **Este seminario describe un procedimiento para usar Ghidra para realizar análisis de código malicioso**
  - Es sólo una breve introducción a operaciones de análisis comunes
  - Este campo es mucho más complejo
- **Ghidra es un framework de ingeniería inversa de software libre (SRE)**
  - Desarrollado por la agencia de seguridad nacional (NSA) de EEUU
  - Es una poderosa herramienta de ingeniería inversa disponible para todos de forma gratuita
  - Permite analizar malware para entenderlo mejor
    - Esto ayuda a localizar vulnerabilidades en sistemas y redes
- **Desde su lanzamiento Ghidra ha atraído a una creciente comunidad de colaboradores**

GHIDRA

https://ghidra-sre.org/

- **Alternativa gratuita a la popular IDA Pro**
- **Principales características**
  - Análisis del código compilado en Windows, Mac OS Y Linux
  - Desensamblado, ensamblado, decompilación y visualización de código
  - Acciones hacer/deshacer
  - Gran conjunto de instrucciones de CPU y formatos ejecutables soportados
  - Soporta plugins de terceros o scripts en Java o Python (Jython) utilizando su API



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

< Ir al Índice

Fuente: Bing Chat AI

ARRANCANDO GHIDRA Y CARGANDO UN EJECUTABLE

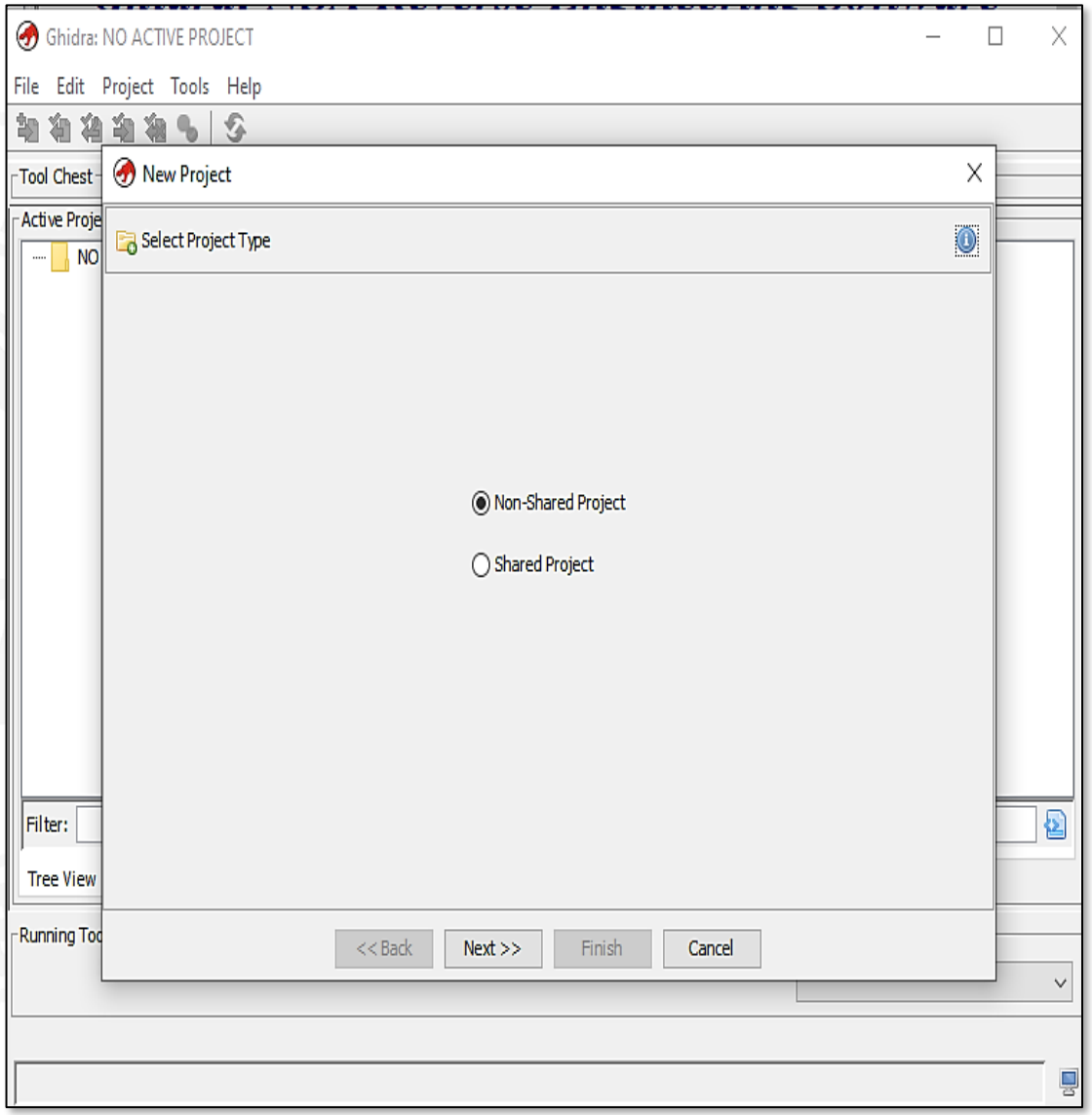
Poniendo en marcha el entorno de análisis

Departamento de Informática
Universidad de Oviedo

Universidad de Oviedo

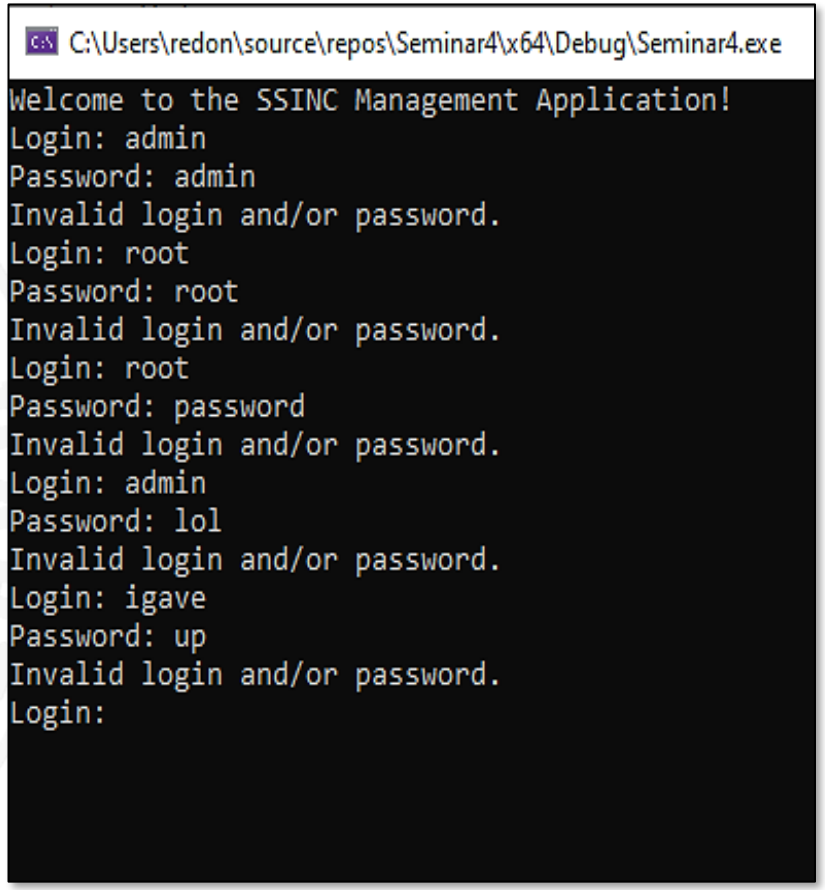
CARGANDO UN EJECUTABLE

- **La instalación de Ghidra requiere**
  - Descargar y extraer un zip y ejecutar el archivo **batch**
  - Instalar un **JDK / OpenJDK 17+**
    - https://adoptopenjdk.net/index.html?variant=openjdk11&jvmVariant=hotspot
- **Luego, crear un proyecto (File– New Project)**
  - Se suele analizar **malware** con varios módulos, por lo que se centra en proyectos y no en archivos
  - Las opciones del proyecto incluyen
    - "**Non shared**" para el análisis con un solo usuario
    - "**Shared**" para el trabajo colaborativo (varios usuarios pueden acceder a un repositorio de proyectos en un servidor)
  - **Nos centraremos en proyectos de un solo usuario**
    - El otro tipo es bueno para compartir los esfuerzos de **reversing**
  - Los proyectos necesitan un nombre



NUESTRO EJECUTABLE

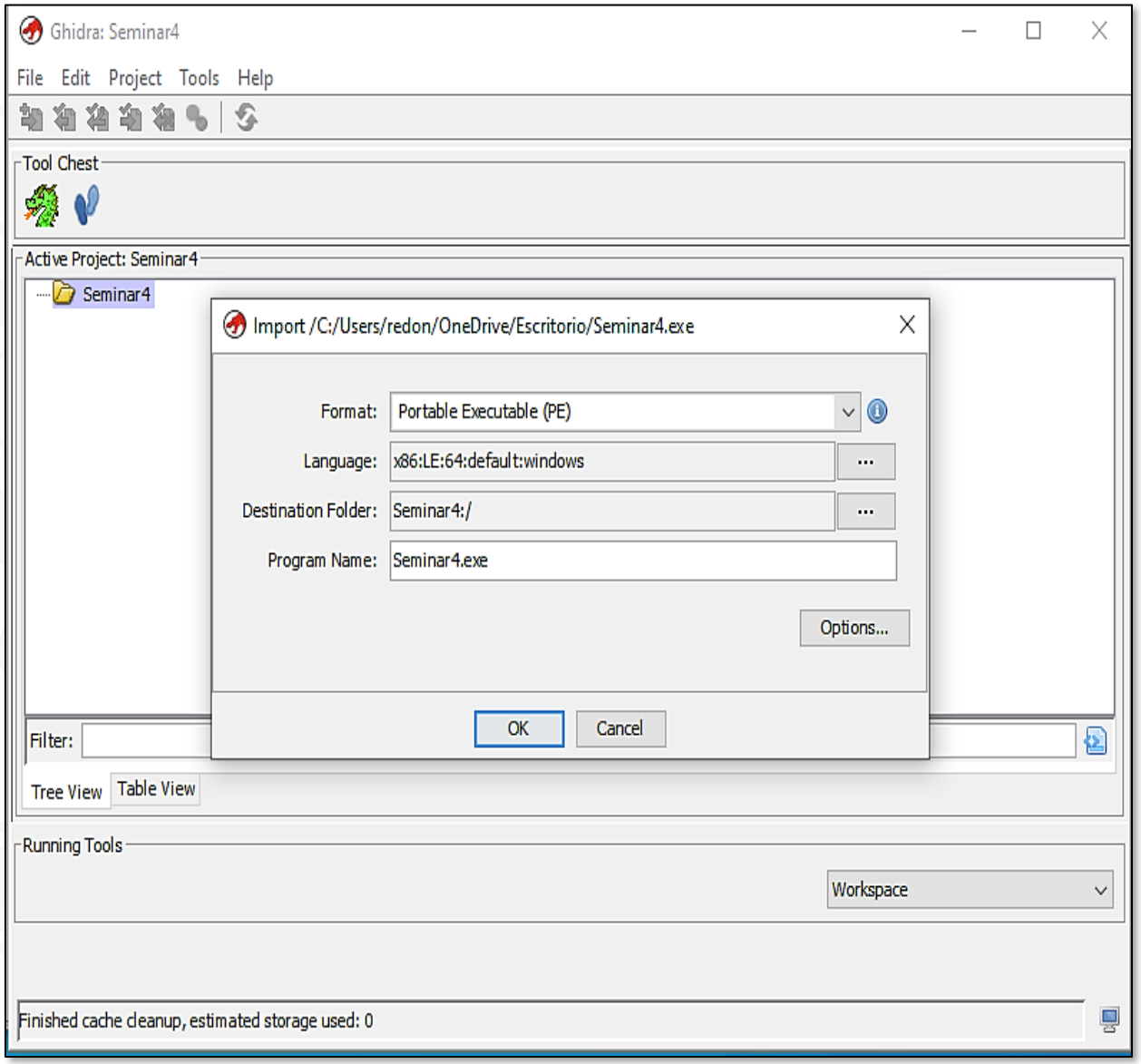
- **Vamos a realizar ingeniería inversa de un ejecutable propio que dos damos**
  - Es una aplicación de administración de línea de comandos falsa, creada con Visual Studio C++
    - Sin GUI para que el código sea pequeño
  - Compilada en modo de depuración
  - Tiene un paso inicial de inicio de sesión
  - Si se inicia sesión correctamente, muestra un menú
  - Todas las opciones están "En construcción“ ☺
  - Pero al iniciar sesión con éxito, el programa está “robando” los datos del portapapeles del usuario periódicamente
    - Sólo los muestra, no los roba de verdad
    - Pero sería muy fácil almacenarlos, enviarlo a otro sistema…
  - **Se deben copiar los archivos `.exe` y `.pdb` en la misma carpeta**



Hay un proyecto de Visual Studio Project disponible en el Campus Virtual

CARGANDO UN EJECUTABLE

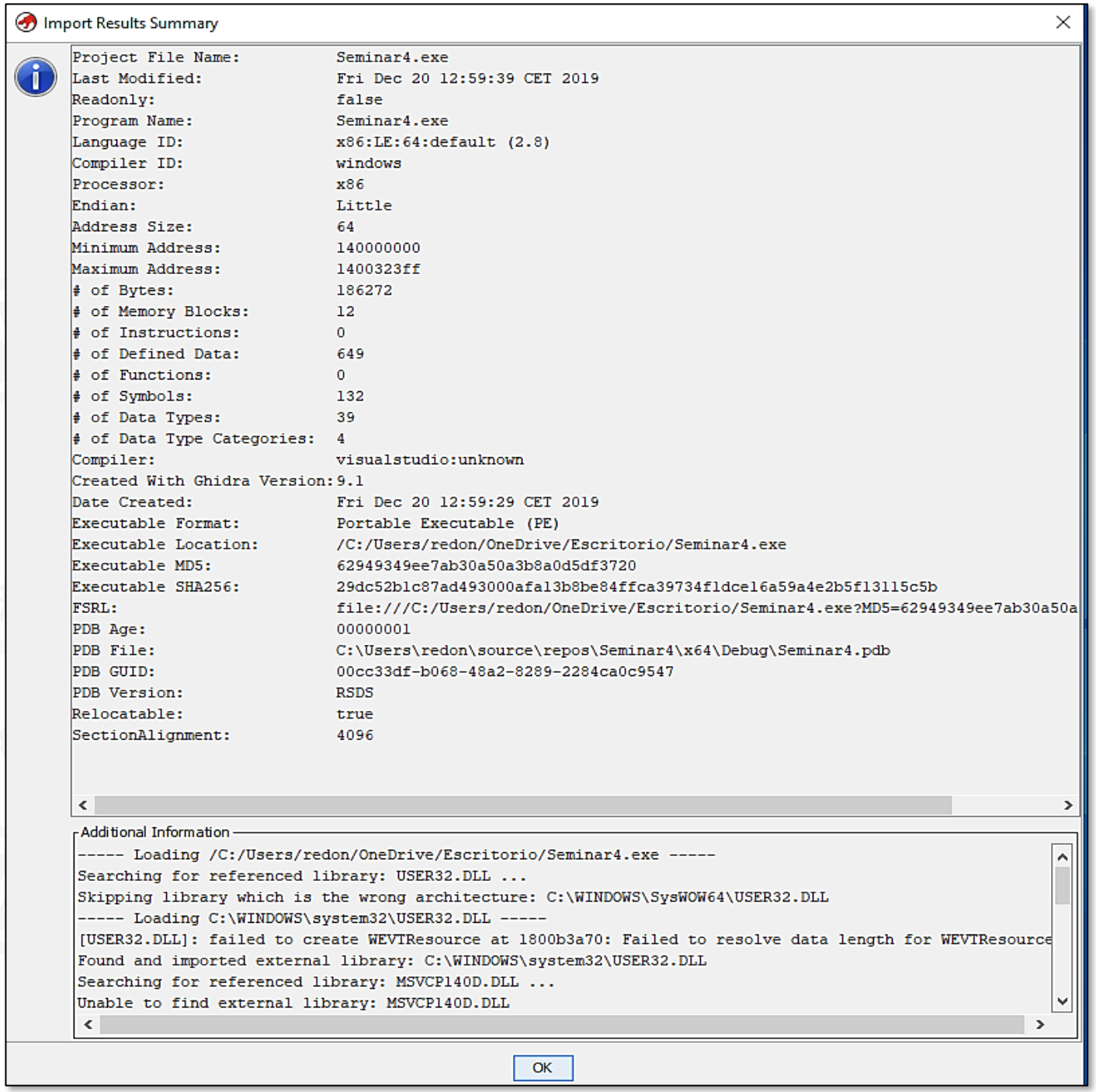
- Para analizar un ejecutable, **Ghidra** solo necesita arrastrar y soltar el archivo `.exe` en la ventana del proyecto
- Esto iniciará un cuadro de diálogo donde se pueden aceptar los valores predeterminados y hacer clic en OK
- Si en la misma carpeta existe un archivo `.pdb`, **Ghidra** intentará usarlo
  - Estos archivos contienen los símbolos de depuración
  - Podrían no funcionar en algunas configuraciones (veremos esto más adelante)



Información del ejecutable cargada

Seguridad de Sistemas Informáticos

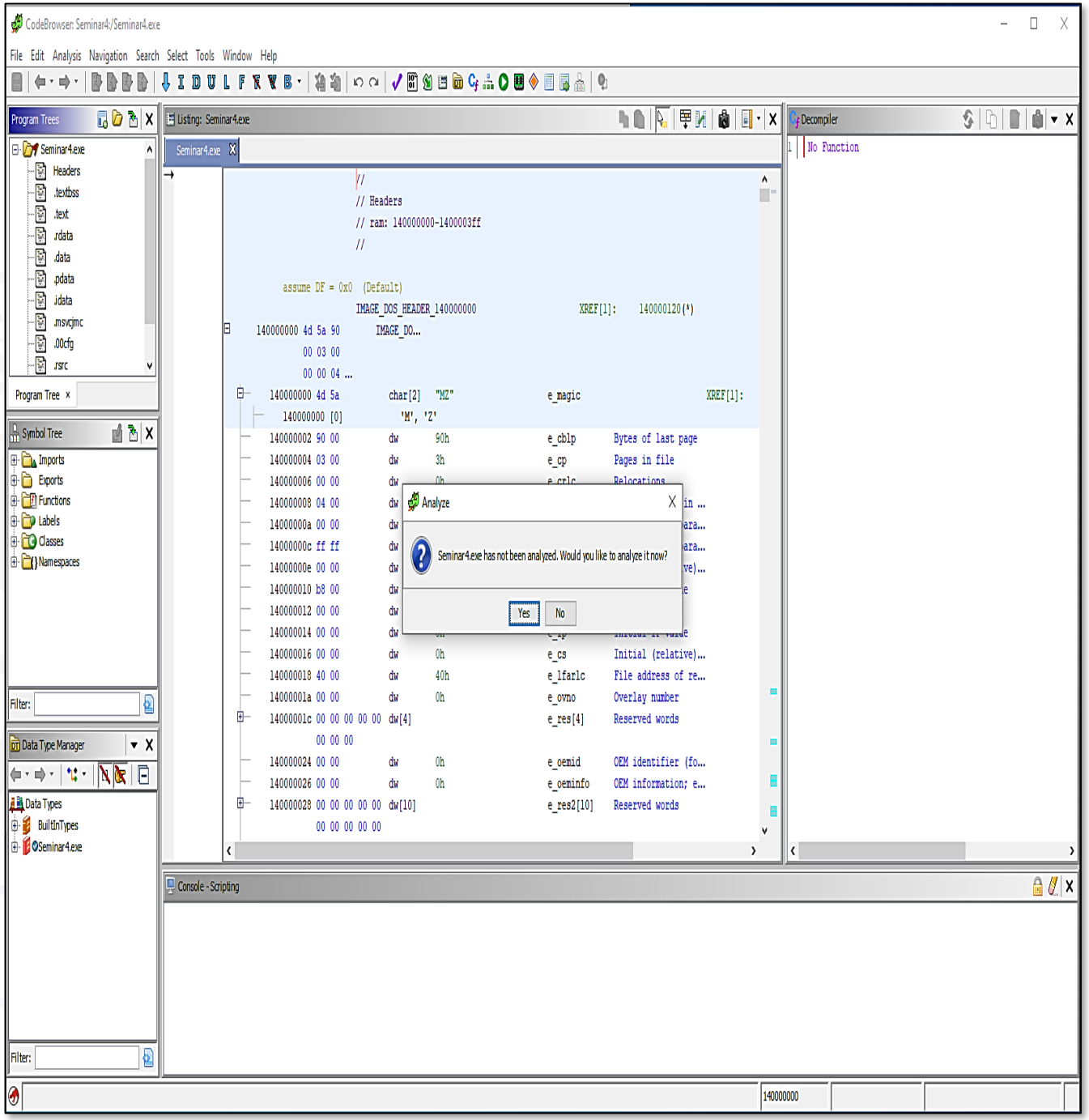
- Cuando se agrega por primera vez un archivo a un proyecto Ghidra, aparece una ventana de información especial
- Contiene una gran cantidad de información útil sobre el archivo como
  - Varios tamaños
  - Compilador utilizado para crearlo
  - Marcas de tiempo
  - Hashes de ejecutable
  - Arquitectura



ANALIZANDO UN EJECUTABLE

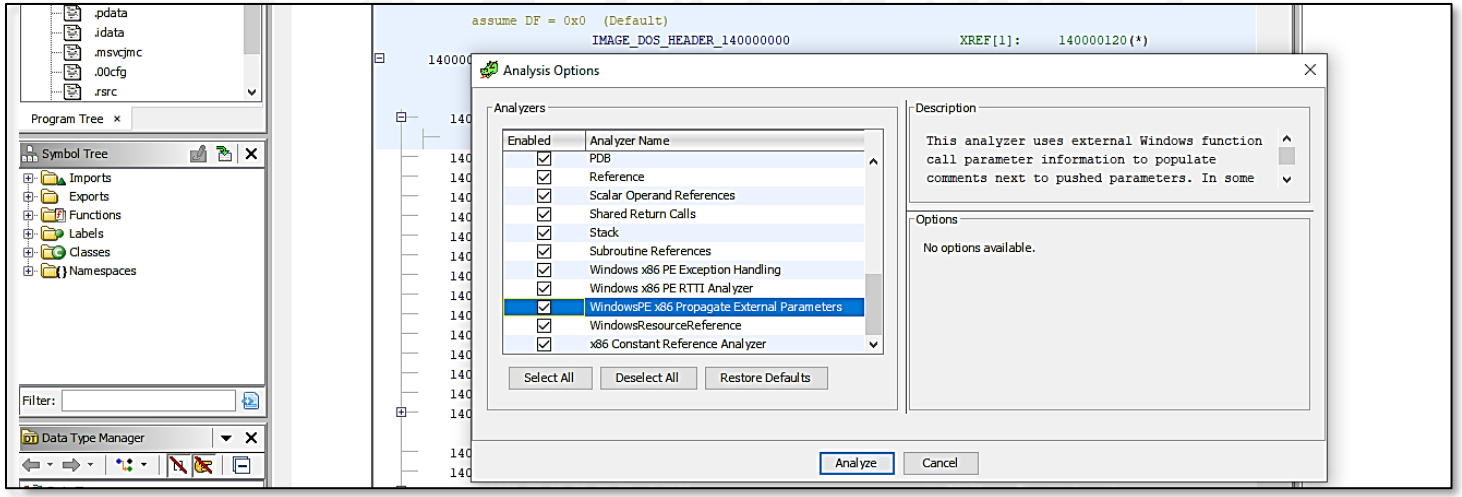
Seguridad de Sistemas Informáticos

- **Si se hace doble clic en el archivo importado en la ventana del proyecto se ejecuta el Code Browser**
  - Si no se hizo antes, la herramienta te pregunta si analiza el archivo ahora
  - **Esto es necesario** para continuar el proceso de reversing
  - El análisis se puede hacer / rehacer manualmente en el menú "Analysis"
- **Activa la opción "WindowsPE x86 propagate external parameters"**
  - Escribe los argumentos de las funciones en comentarios
  - Esto es conveniente para entender mejor el código analizado



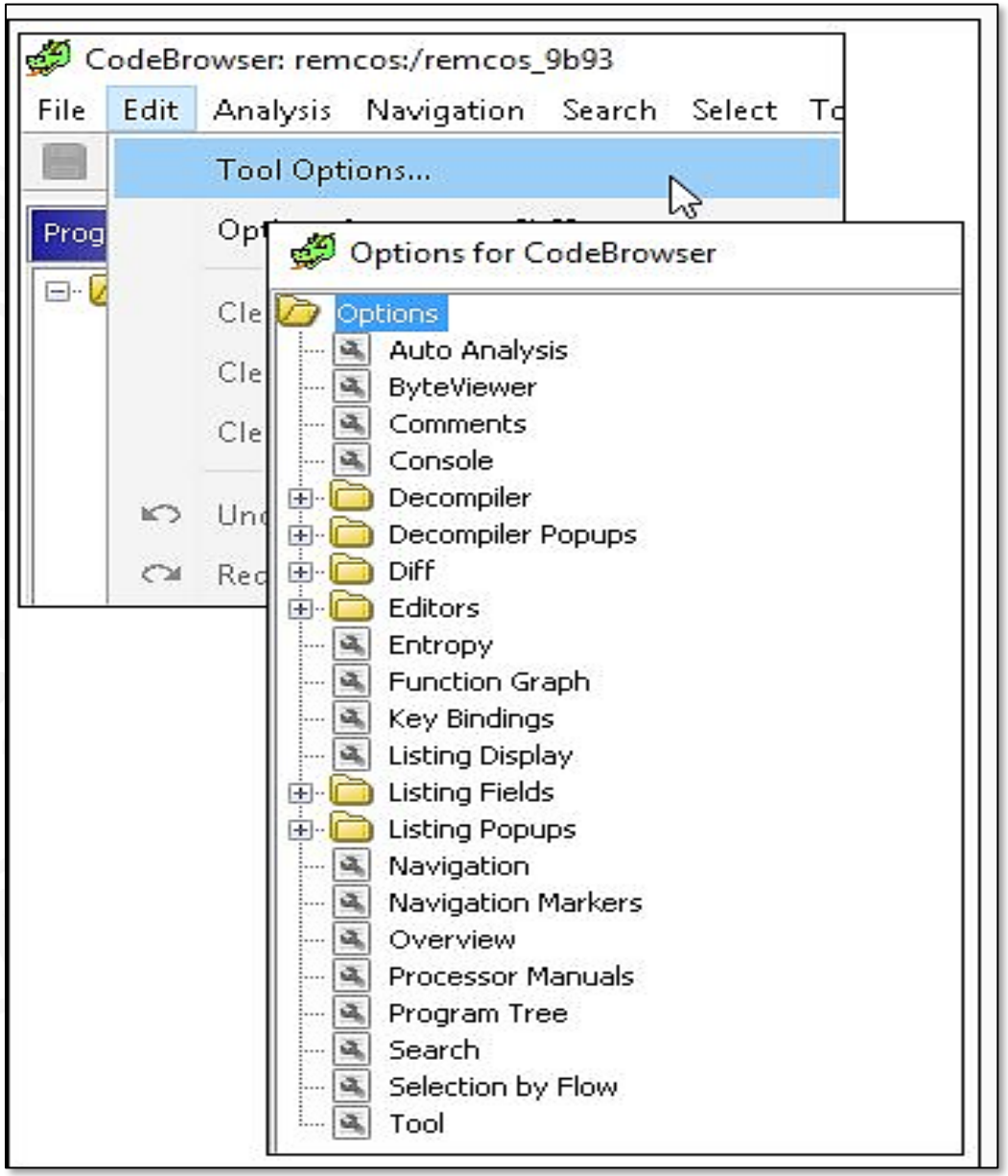
ANALIZANDO UN EJECUTABLE

- **Podemos comenzar el análisis y que Ghidra recopile información del `.exe`**
  - Como estamos en Windows y se proporciona un archivo `.pdb`, Ghidra debería ser capaz de cargar este archivo y completar la información del exe
  - Sin embargo, podría quejarse de no encontrar el SDK DIA que se usa para cargar archivos `.pdb`
    - Este SDK se instala con Visual Studio (incluso en versiones gratuitas como Visual Studio Community 2019)
    - En teoría se podría buscar el archivo `msdia140.dll` en el disco duro y registrarlo con `regsvr32` desde una terminal con privilegios de `Administrador`
    - **Esto no funciona en algunos casos**: no podemos ver información ampliada sobre algunos elementos (como los nombres de función)
    - Pero vamos a proceder de todos modos, ya que tener un archivo `.pdb` no es común en contextos reales



MODIFICANDO LOS ELEMENTOS MOSTRADOS

- **Conviene hacer algunos cambios a través de Edit – Tool options para facilitar la lectura**
  - **Listing Display** : Aumentar el tamaño de fuente y habilitar el formato en negrita
  - **Listing Fields: Bytes Field**: Cambiar *"Maximum Lines to Display"* a 1
  - **Listing Fields: Cursor Text Highlight**: Cambiar *"Mouse Button to Activate*" a LEFT
  - **Listing Fields: EOL Comments Field** : Marcar *"Show Semicolon at Start of Each Line"*
  - **Listing Fields: Operands Field** : Marcar *"Add Space After Separator"*



< Ir al Índice

Fuente: Bing Chat AI

Inspeccionando los datos de la aplicación

Analizando strings dentro de un ejecutable

Departamento de Informática
Universidad de Oviedo

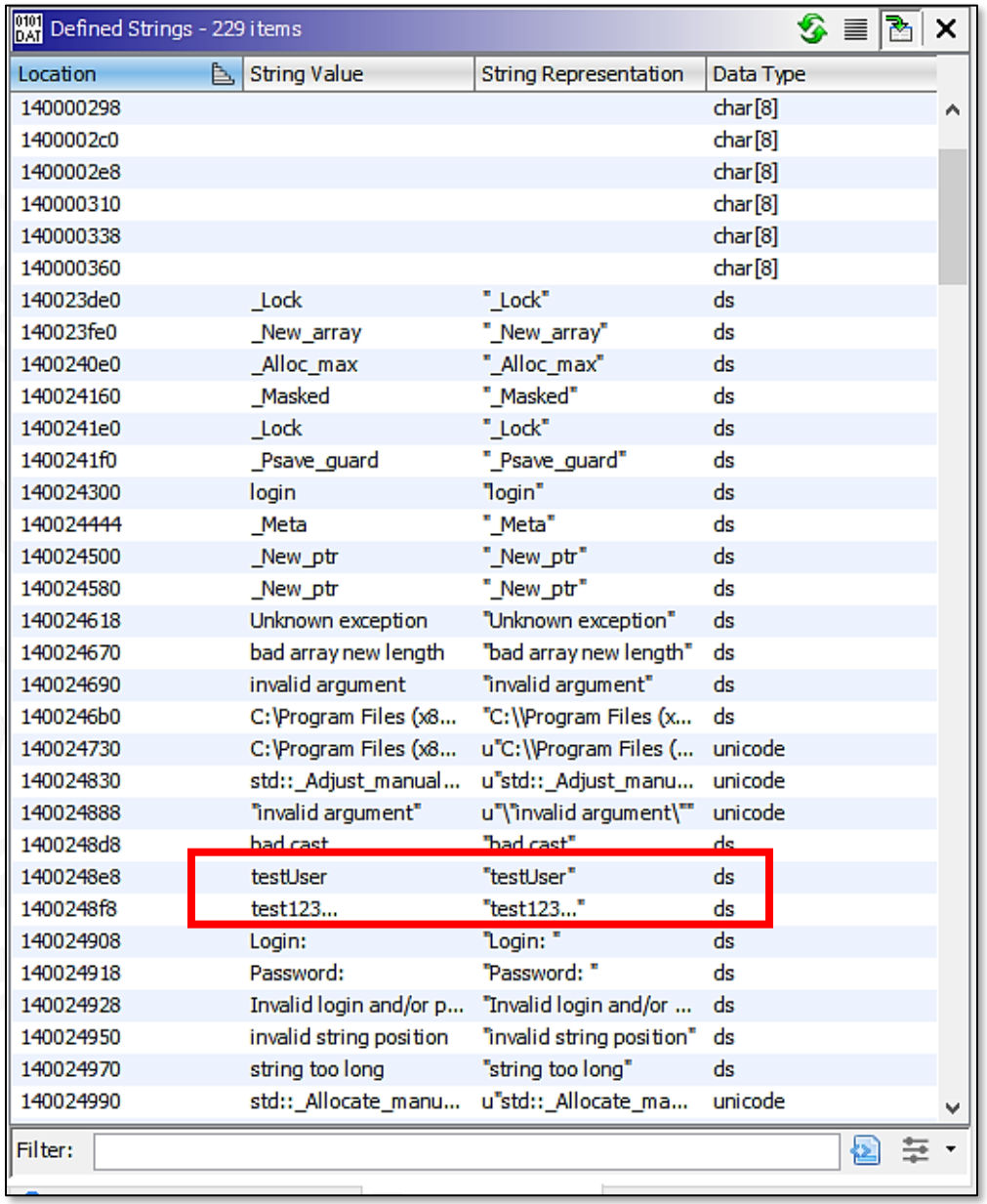
Universidad de Oviedo

Investigando referencias a String

- Una de las cosas de las que podríamos no ser conscientes es que **las cadenas incluidas en nuestro ejecutable son visibles** con herramientas como Ghidra
- Esto significa que la creación de constantes con claves, IDs importantes, nombres de usuario, contraseñas, etc. **siempre es una mala idea**
  - Las cadenas deben ser ofuscadas
  - O, mucho mejor, se deben cargar desde fuentes cifradas externas
- Los datos que residen en el código pueden ser localizados y utilizados en nuestra contra
  - Por ejemplo, no tenemos idea de cómo eludir el mecanismo de autenticación de aplicaciones
  - ¿Qué sucede si el código de la aplicación tiene algún tipo de información de autenticación "oculta"?
- Si sólo queremos buscar cadenas codificados en un ejecutable, la herramienta de línea de comando `strings` puede hacer el mismo trabajo más fácilmente

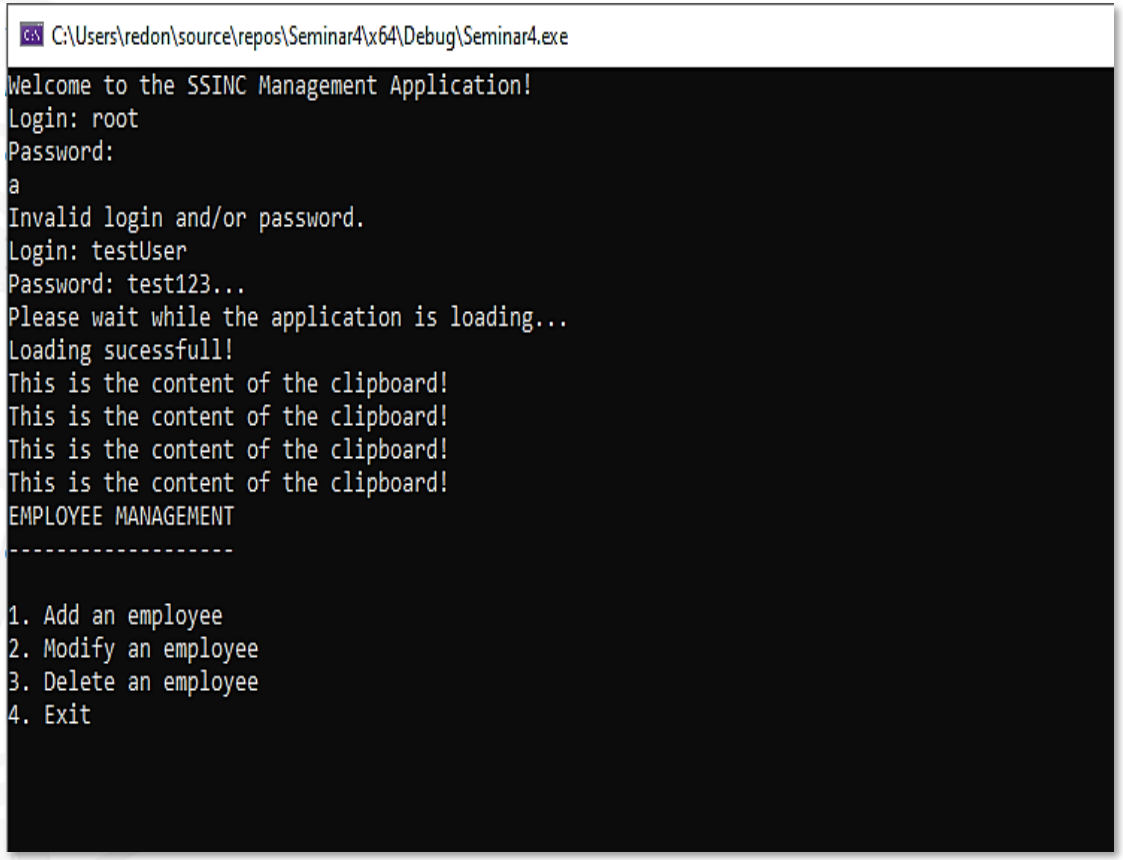
Investigando referencias a String

- **Para ver las cadenas incrustadas en un ejecutable, vamos a Window - Defined Strings**
  - En esta ventana aparece una lista de todas las cadenas definidas
  - Incluida su dirección dentro del código, el valor y la representación de cadena
  - Vemos que hay una gran cantidad de cadenas que pertenecen a los mensajes estándar de la aplicación
  - Pero también localizamos 2 sospechosas "`testuser`" y "`test123...`" que parecen ser un nombre de usuario / contraseña
- **¿qué pasa si probamos esta combinación?**



Datos dentro de un ejecutable...¿ocultos?

- **Adivinamos que se trataba de una combinación de usuario / contraseña válida**
- **¿Es común esta situación?**
  - Desgraciadamente, no es raro encontrar puertas traseras / información privilegiada codificada en un archivo ejecutable
  - Nombres de usuario válidos
  - Contraseñas
  - API Keys
  - Identificadores
  - URLs desconocidas / IPs / rutas
  - Nombres de funcionalidades indocumentadas
  - ...
- **¡Ahora sabemos que hacer esto no es seguro!**



Nota: nuestra aplicación de ejemplo se bloquea si el contenido del portapapeles no es una cadena

< Ir al Índice

Fuente: Bing Chat AI

Decompilar >

Datos >

Código >

Inspeccionando el Comportamiento de la Aplicación

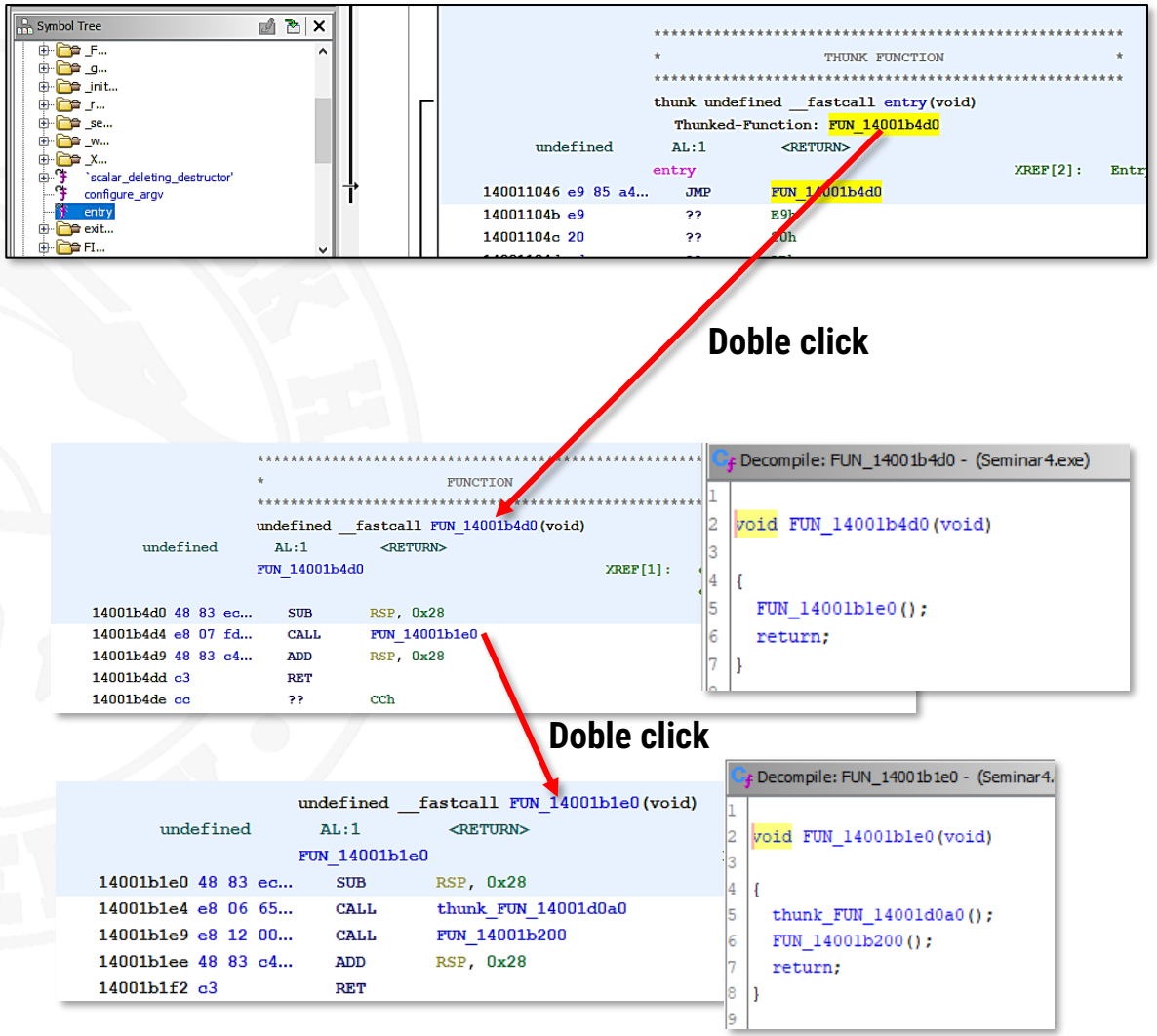
Ver lo que hace nuestro programa

Departamento de Informática
Universidad de Oviedo

Universidad de Oviedo

"Siguiendo" a un programa

- El comportamiento de un programa se inspecciona analizando sus llamadas desde su **punto de entrada**
  - Los puntos de entrada tienen un nombre especial en el árbol de símbolos (`entry`)
  - A partir de él podemos seguir la ejecución (es como la función `main` típica de lenguajes como Java o C#)
- Podemos encontrar **thunked functions**
  - Funciones que **inyectan cálculos adicionales** en otras
  - Usadas para retrasar un cálculo hasta que se necesite su resultado
  - O para insertar operaciones al principio o al final de una función
  - Comúnmente usadas por los compiladores en la fase de generación de código
  - Normalmente podemos simplemente ir a la a la que llaman, **ignorando su contenido**



El reversing no es sencillo...y encima hay obstáculos

- **Ghidra puede intentar facilitar que veas el valor de una variable**
  - Interpretando y mostrándola según su tipo
  - Lo cual a veces hace que te encuentres con "easter eggs", como esta imagen
- **No creas que las empresas no saben que alguien va a descompilar sus programas**
- **Los programas suelen tener técnicas de ofuscación o contra reversing**
  - Dificultan tu labor porque no les interesa que veas cómo hace el programa ciertas cosas
    - Ej.: Comprobar que no es una copia pirata ☺
  - Por ello, no desesperes si hay cosas que son muy difíciles de entender

```
001b6835 30              ??         30h    0
001b6836 20              ??         20h
001b6837 35              ??         35h    5
001b6838 30              ??         30h    0
001b6839 20              ??         20h
001b683a 31              ??         31h    1
001b683b 36              ??         36h    6
001b683c 20              ??         20h
001b683d 31              ??         31h    1
001b683e 00              ??         00h
                    JPEG_001b683f+23300                    XREF[0,2]:   FUN_000291f8:0002
                    JPEG_001b683f+23861                                 FUN_0002a248:0002
001b683f ff d8 ff        JPG
         e0 00 10
         4a 46 49 ...
001bc742 00              ??         00h
                    GIF_001bc743+385                       XREF[0,1]:   FUN_000291f8:0002
001bc743 47 49 46        GIF
         38 39 61
         97 00 28 ...
001bc973 00              ??         00h
001bc974 b7              ??         B7h                                  ?  ->  001bc8
001bc975 c8              ??         C8h
001bc976 1b              ??         1Bh
001bc977 00              ??         00h
001bc978 a1              ??         A1h                                  ?  ->  001bc8
001bc979 c8              ??         C8h
001bc97a 1b              ??         1Bh
001bc97b 00              ??         00h
001bc97c 8b              ??         8Bh                                  ?  ->  001bc8
```

RSA™
PUBLIC KEY CRYPTOSYSTEM

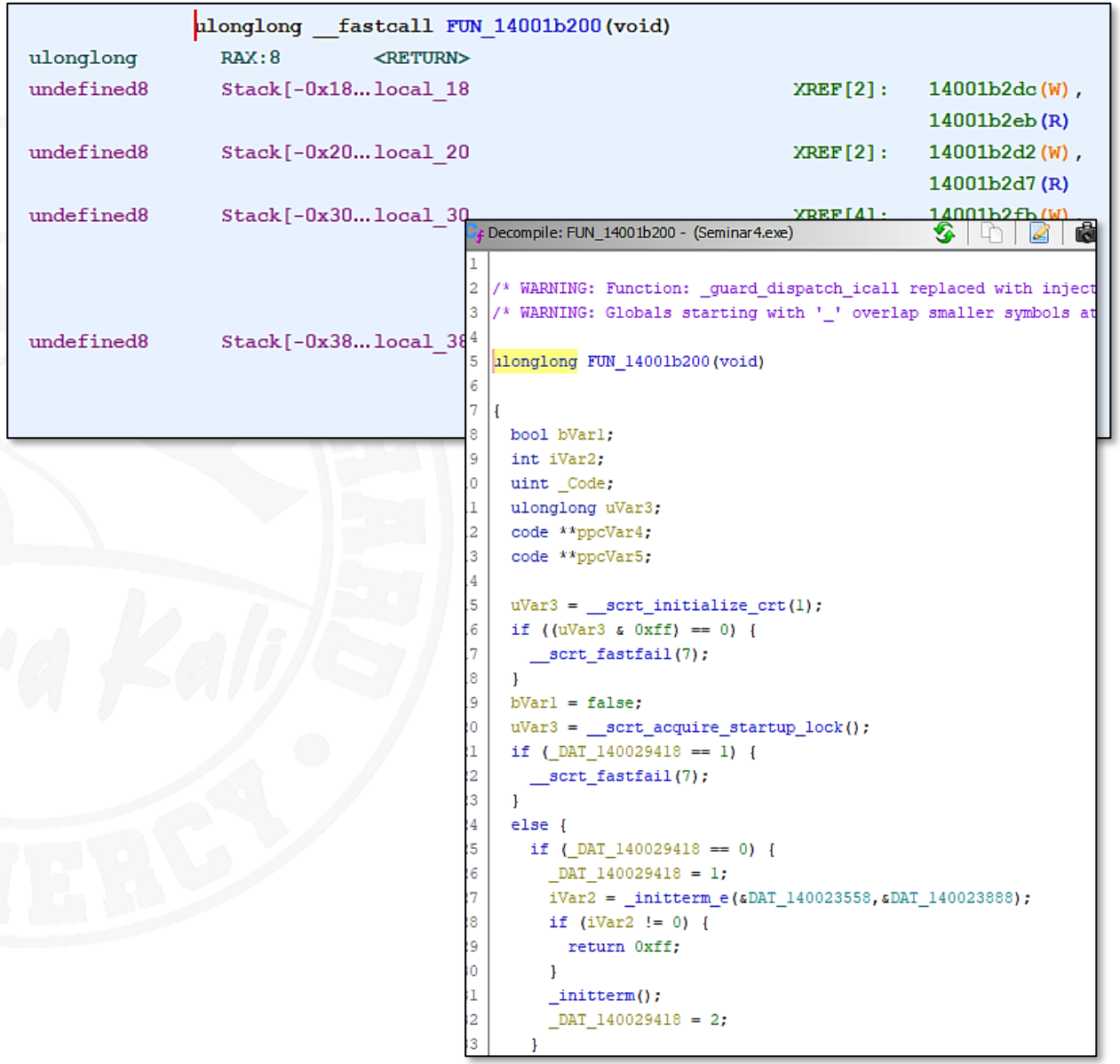
Decompilar

Obtener código fuente legible

"Siguiendo" a un programa: Decompilación

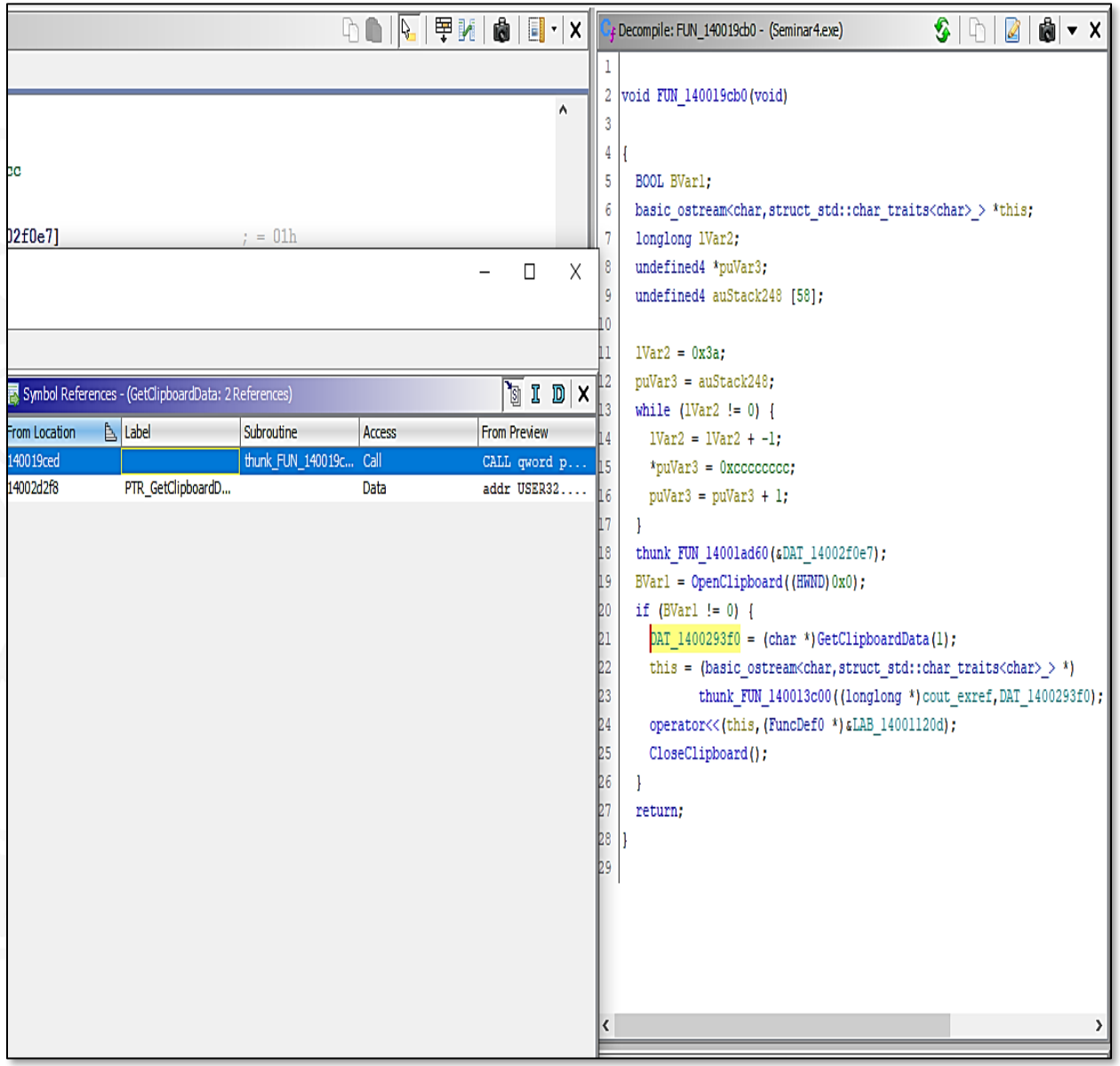
Seguridad de Sistemas Informáticos

- Siguiendo la secuencia de llamadas, llegaremos a funciones con el código real del programa
- Podemos ver el diseño del código ASM y la salida de un **decompilador**
- Este decompilador tratará de transformar el código ASM en su código equivalente en C
  - Este código es más legible, pero por desgracia **también es difícil de seguir**
  - El proceso de compilación en ensamblador x86 pierde información
    - Sobre los nombres de variables y funciones, entre otras cosas
    - Algunas construcciones de código diferentes pueden en realidad generar el mismo código ASM



VER LA SALIDA DEL DECOMPILADOR

- Esto significa que, aunque ayuda, **no podemos obtener el código fuente original exacto**
  - Y es peor aún si no podemos usar los archivos del depurador (.pdb)
- **La salida del descompilador se sincroniza con los cambios hechos en la vista listado / Listing**
  - Sí, Ghidra nos permite **realizar cambios en el código ejecutable del programa**
  - No lo hacemos porque está más allá de los objetivos de esta asignatura
  - Los cambios realizados en los nombres de función, variables o argumentos en la vista Listing o Decompile actualizan la otra ventana



Investigando una referencia a un API

- Es muy común identificar código interesante en base a referencias a la API del SO
- Para hacer eso hay que ir a **Window - Symbol References**
- Selecciona una llamada a la API que quieras investigar e identifica dónde se llama a esa función (ventana **symbol references**)
  - Por ejemplo, vale la pena investigar la llamada al API `GetClipboardData` para entender lo que hace este programa
  - Para localizar una referencia al API hay que hacer clic en una referencia en el lado derecho
  - Eso rellena la vista Listing con el desensamblado adecuado (ASM + Código C)

INVESTIGANDO UNA REFERENCIA A UN API

Seguridad de Sistemas Informáticos

```
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 = 0xcccccccc;
16    puVar3 = puVar3 + 1;
17  }
18  thunk_FUN_1400lad60(&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...	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)

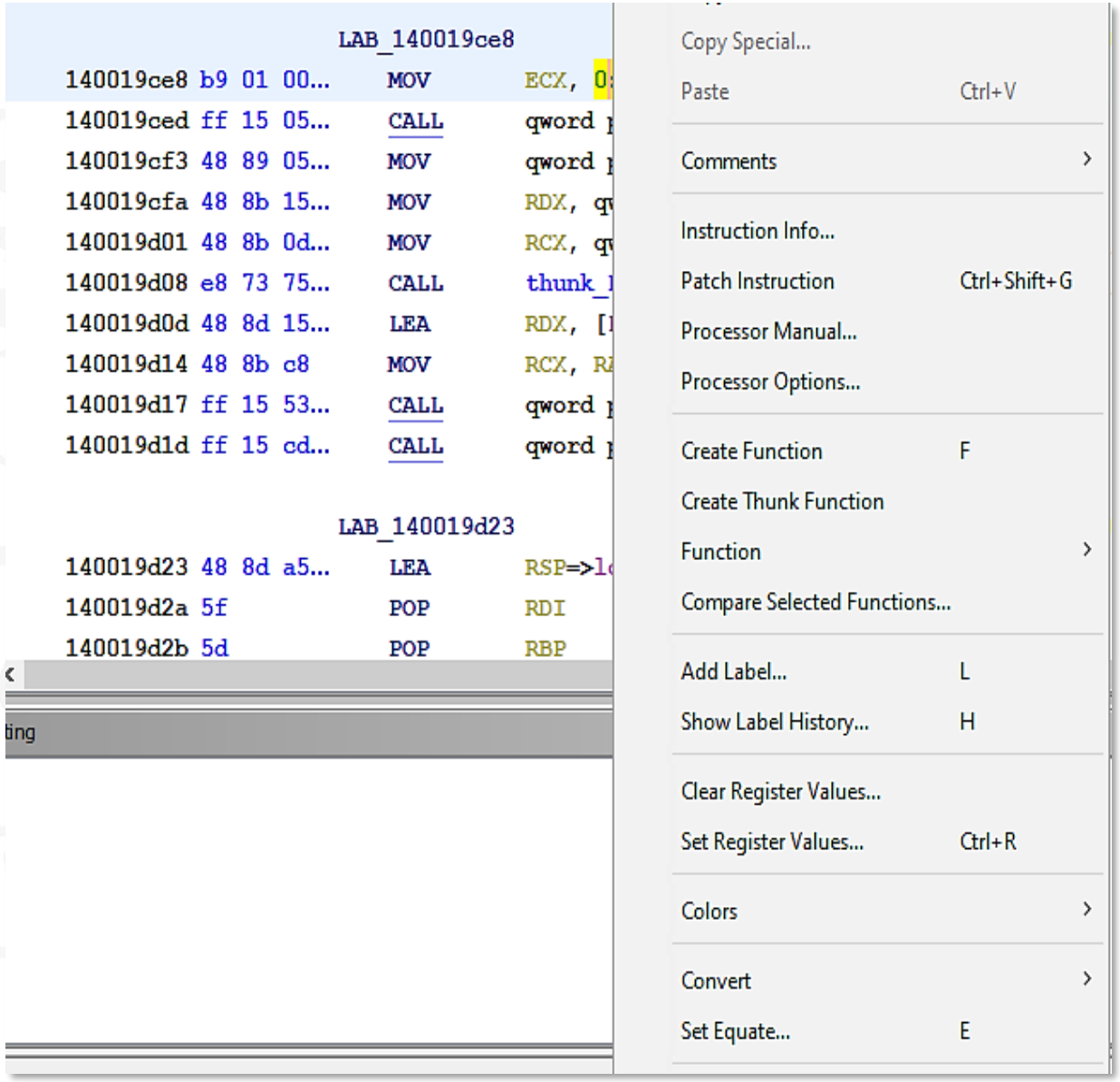
From Location	Label	Subroutine	Access	From Preview
140019ced		thunk_FUN_140019c...	Call	CALL qword p...
14002d2f8	PTR_GetClipboardD...		Data	addr USER32....

Anotación y localización de datos

Dando significado a los datos contenidos en el código fuente

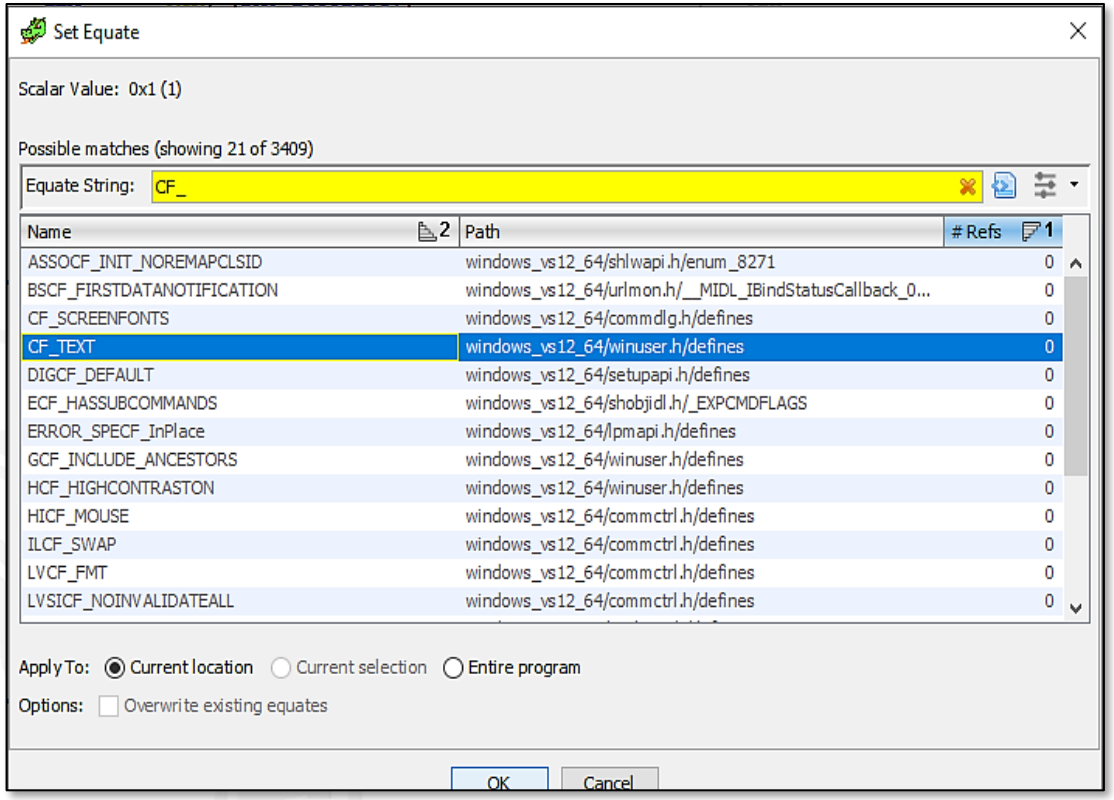
ANOTACIÓN DE DATOS

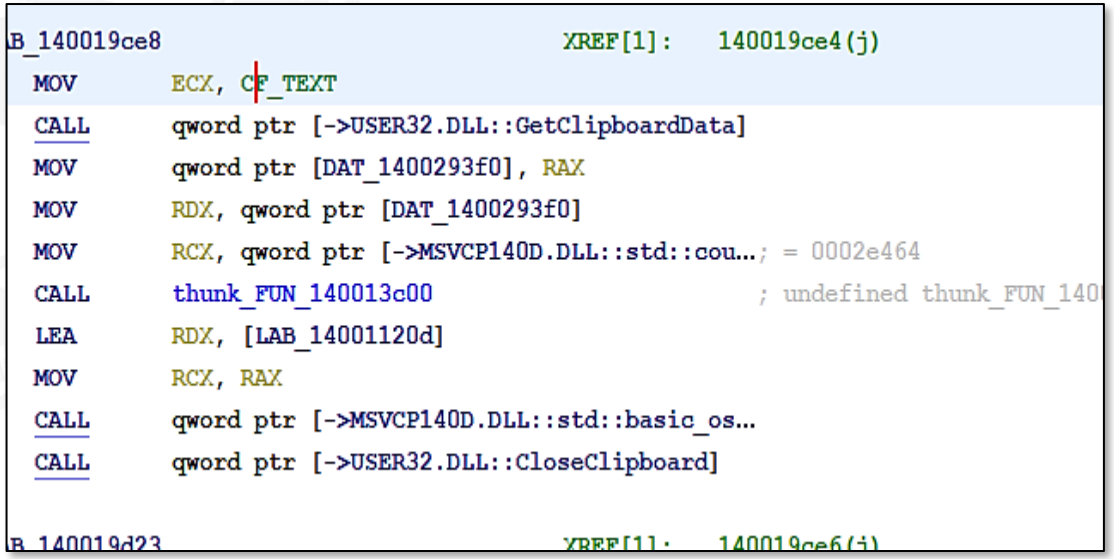
- **En la vista de listado vemos**
  - La llamada inicial (`CALL`) a la función de la API
  - Las instrucciones `PUSH` antes de la llamada `CALL` son los argumentos de la función (si hay)
  - A la derecha, comentarios autogenerados que identifican los parámetros de la función
    - Como se describen en la web de **Microsoft**
- **El único parámetro de `GetClipboardData` es el formato de datos esperado del portapapeles**
  - https://docs.microsoft.com/enus/windows/win32/api/winuser/nf-winusergetclipboarddata
  - El valor `0x1` es una constante simbólica
  - Podemos convertirla en algo **más legible**
    - Haciendo clic con el botón derecho en el valor y seleccionando " **Set Equate...**"



ANOTACIÓN DE DATOS

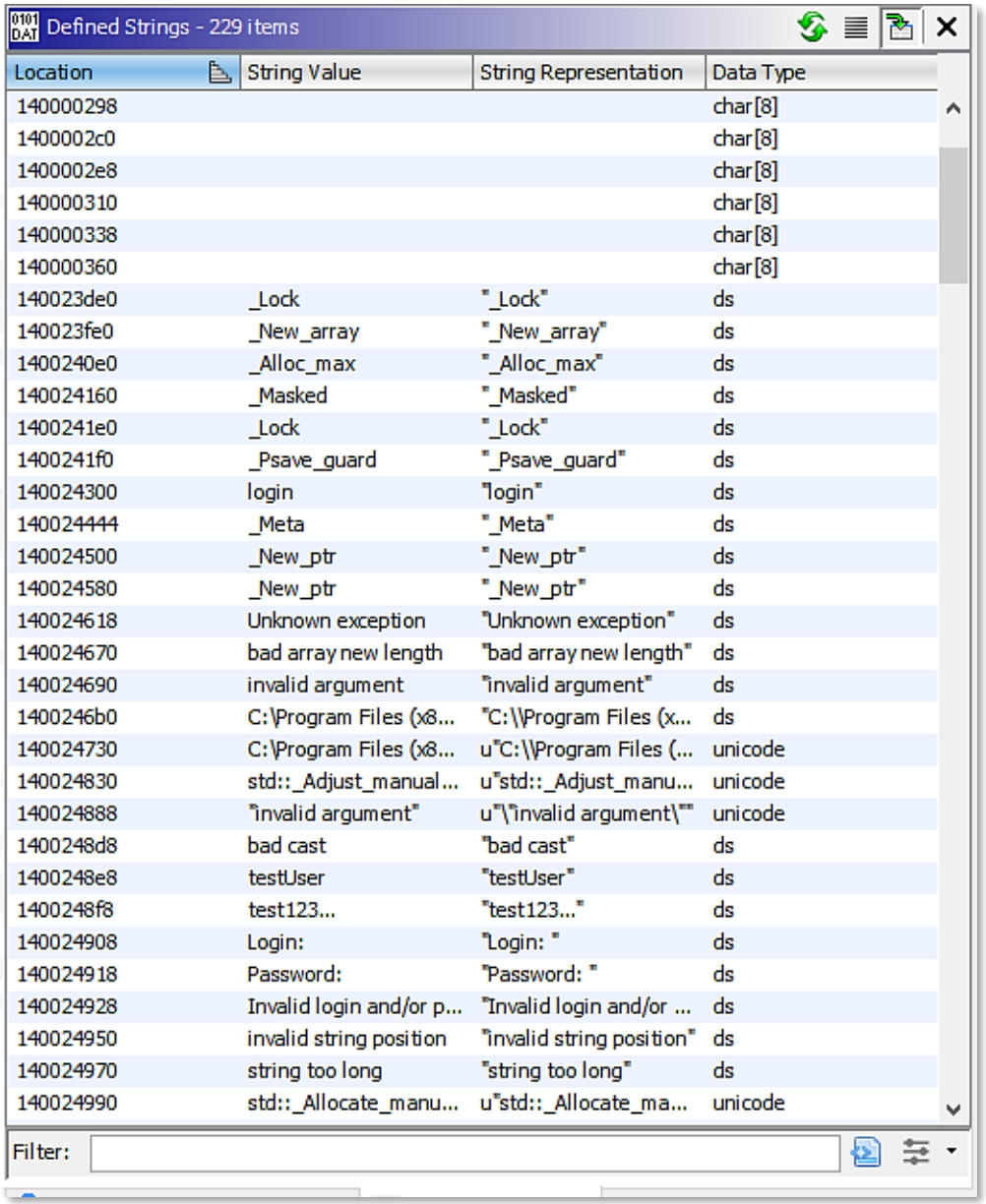
- La ventana resultante muestra varias opciones para el valor hexadecimal elegido
- Para conocer la correcta, debemos consultar la documentación de la llamada al API
  - https://docs.microsoft.com/eses/windows/win32/dataxchg/clipboard-formats
  - La mayoría de los formatos del portapapeles empiezan con CF_
  - Escribiendo esto en el campo "Equate String" nos deja con una única opción compatible: CF_TEXT
  - Significa que se espera que el portapapeles contenga texto
    - https://docs.microsoft.com/es-es/windows/win32/dataxchg/clipboard-formats
  - Selecciona este valor y pulsar OK
- El ensamblador ahora muestra la representación de texto en lugar del valor hexadecimal
- Ya sabemos cómo anotar flags y valores en el código y hacerlos persistentes





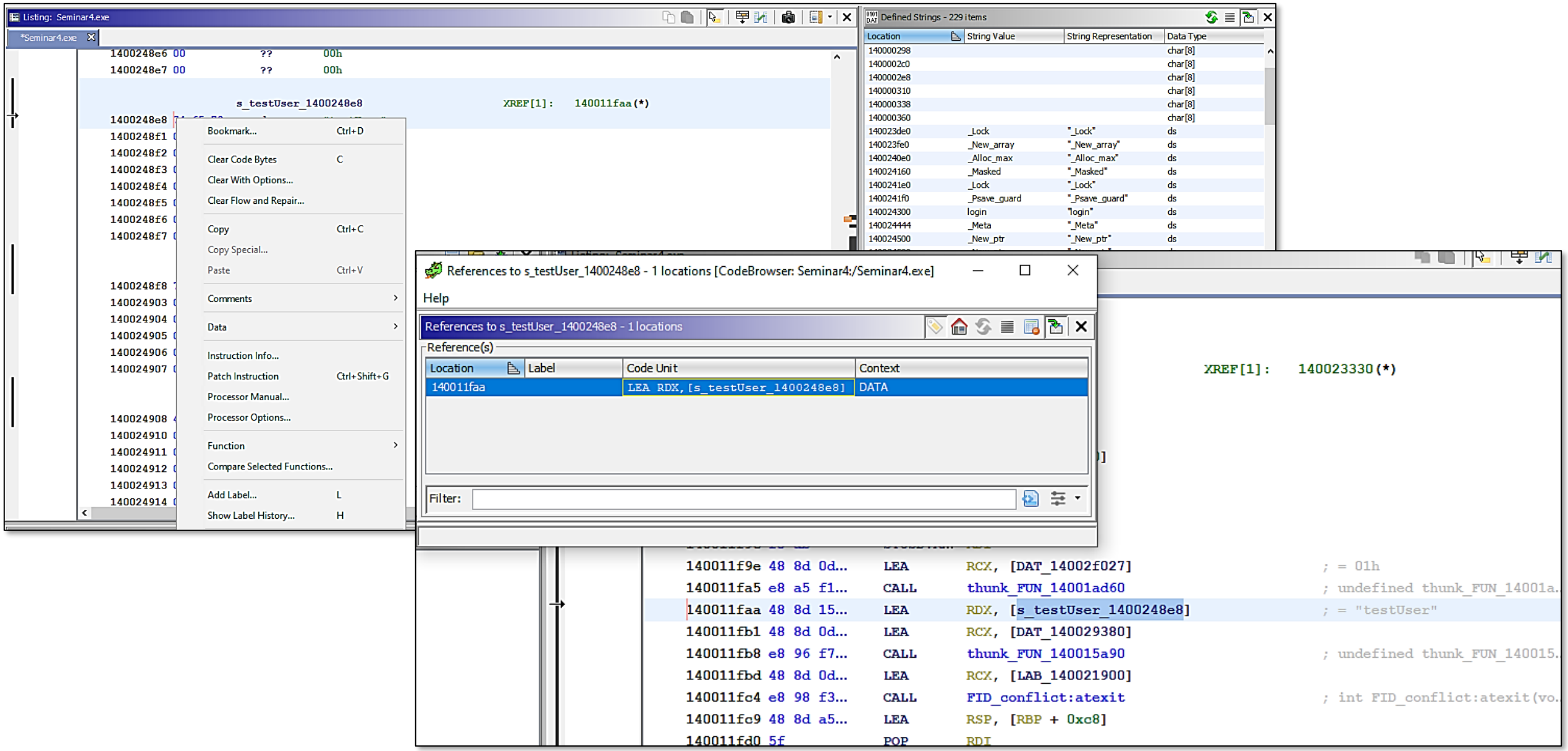
Localizando código a partir de referencias a string

- El análisis de cadenas no sólo sirve para adivinar información oculta en el ejecutable
  - También para **rastrear dónde se utilizan estas cadenas en el código fuente**
- Al hacer clic en la fila asociada a una cadena, se rellena la ventana **Listing** con los datos en su dirección
- Para identificar referencias a esta cadena podemos hacer clic con el botón derecho en el área azul - **References - Show References to Address**
- Vamos a ver qué pasa con la cadena **"testUser"** que localizamos



LOCALIZANDO CÓDIGO A PARTIR DE REFERENCIAS A STRING

Seguridad de Sistemas Informáticos

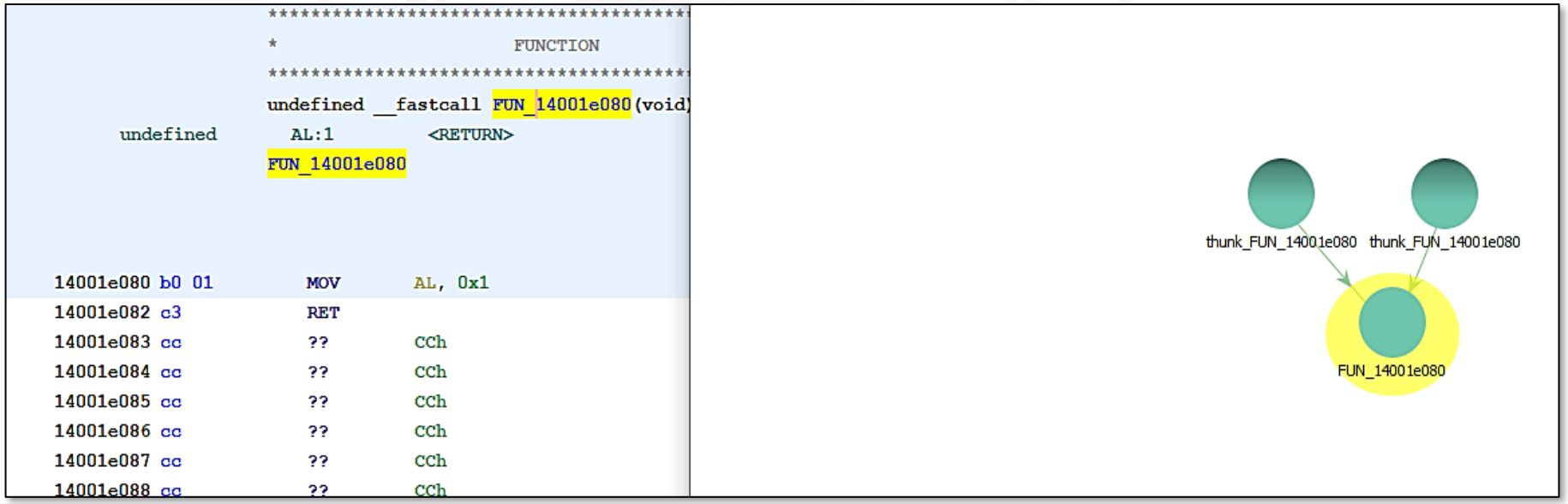


Grafos de llamadas

Herramientas de ayuda para entender el comportamiento del código fuente

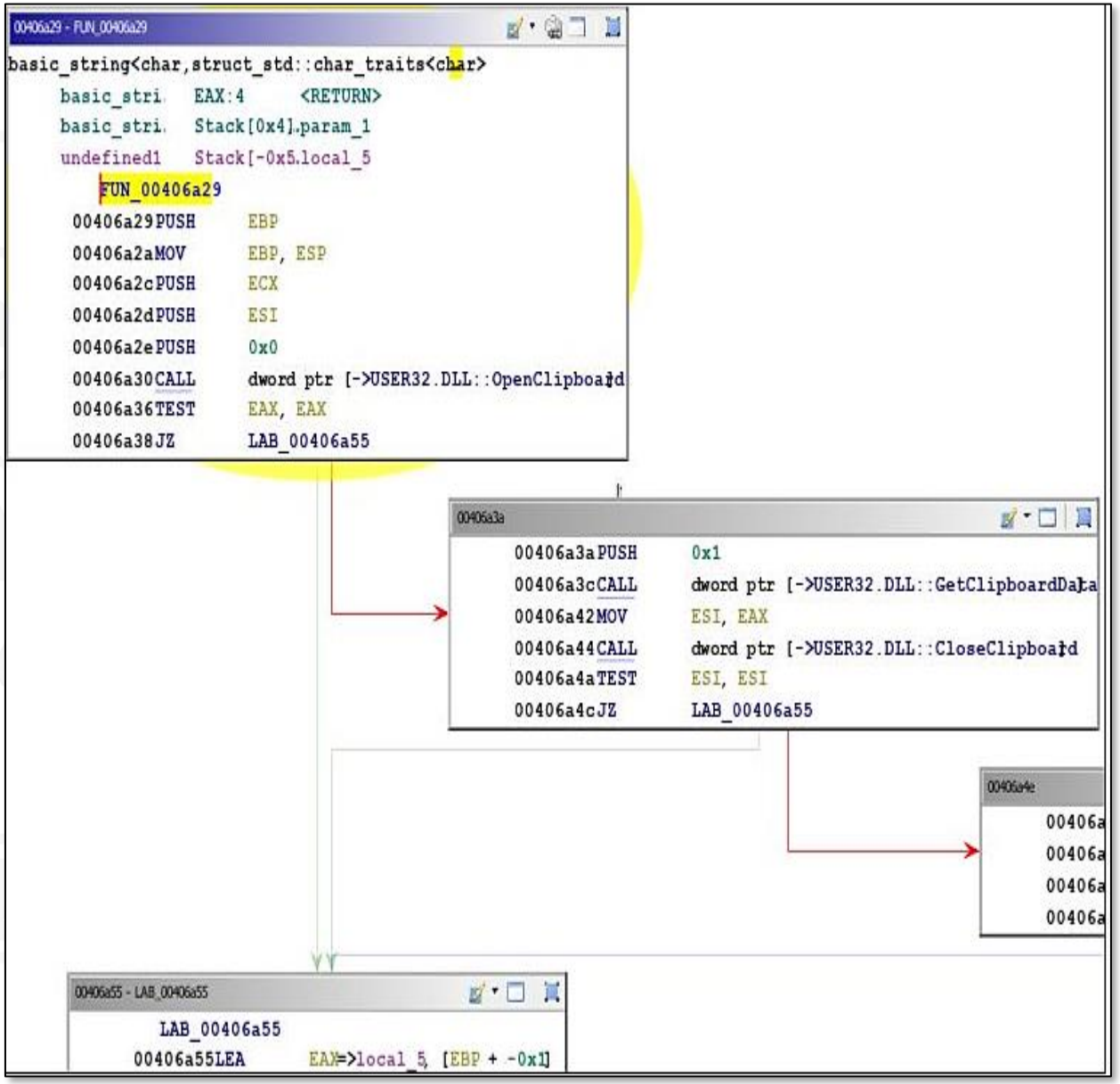
GRAFOS DE LLAMADAS A FUNCIONES

- **Una vez que se identifica una referencia, también podemos analizar la secuencia de llamadas que termina en el código que la utiliza**
- **Para ello, podemos ir al principio de la función que utiliza la cadena y hacer clic en Window-Function Call Graph**
  - Esto muestra las relaciones entre las funciones y proporciona **una visión general de alto nivel de las llamadas a funciones**,
    - ¡Mucho más rápido que hacerlo manualmente!
  - Esto puede permitirnos descubrir **llamadas indocumentadas o inesperadas a las API del SO**
    - Podrían significar actividades maliciosas...
  - O simplemente para entender mejor el comportamiento del programa analizado



GRAFOS DE FUNCIONES

- El análisis del comportamiento de una función podría hacerse revisando el texto desensamblado en la ventana de decompilación
- Sin embargo, también ayuda **tener una representación visual** de los puntos de decisión dentro de una función
- Esto se puede hacer con **Window – Function Graph**
- Muestra los **bloques de código individuales** de la función
  - Las flechas representan saltos condicionales e incondicionales

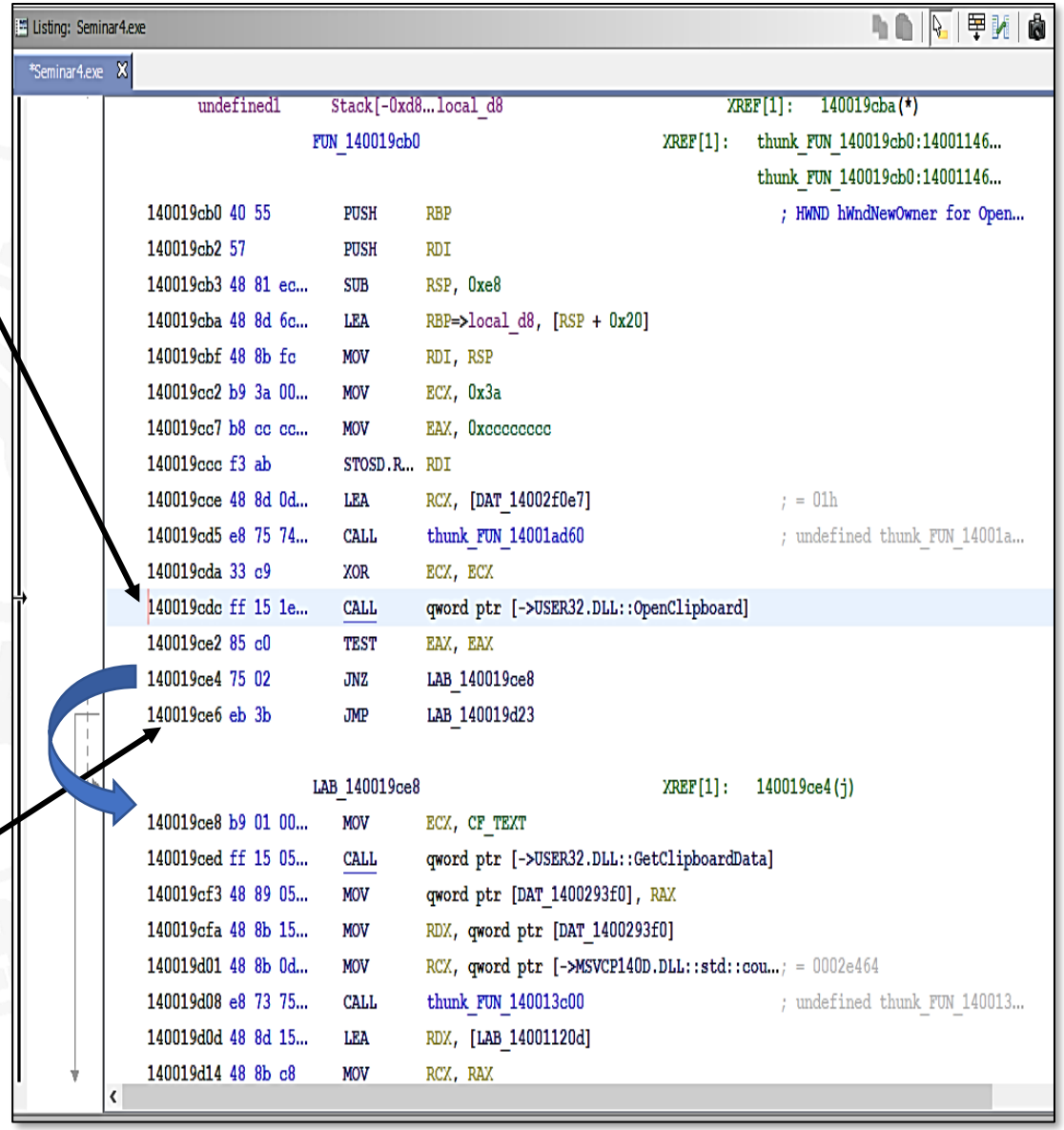


Entendiendo el lenguaje ensamblador

- Por último, una vez que podemos ver los datos, las funciones y las secuencias de llamadas, necesitamos **entender el código del programa**
- Esto requiere un **nivel de habilidad alto para interpretar el código ensamblador, experiencia y paciencia** (visteis algo en FCR)
  - Va mucho más allá de los objetivos de esta asignatura
- Sin embargo, con práctica es posible entender la mayoría de los posibles patrones de instrucciones que podamos encontrar
  - Los traductores de código suelen utilizar estructuras de código coherentes que podemos reconocer
  - Y no debemos olvidar la ayuda del descompilador y el gráfico de funciones para entender mejor lo que hace el código
- Una buena referencia para empezar es: https://www.cs.virginia.edu/~evans/cs216/guides/x86.html
- Veamos un ejemplo

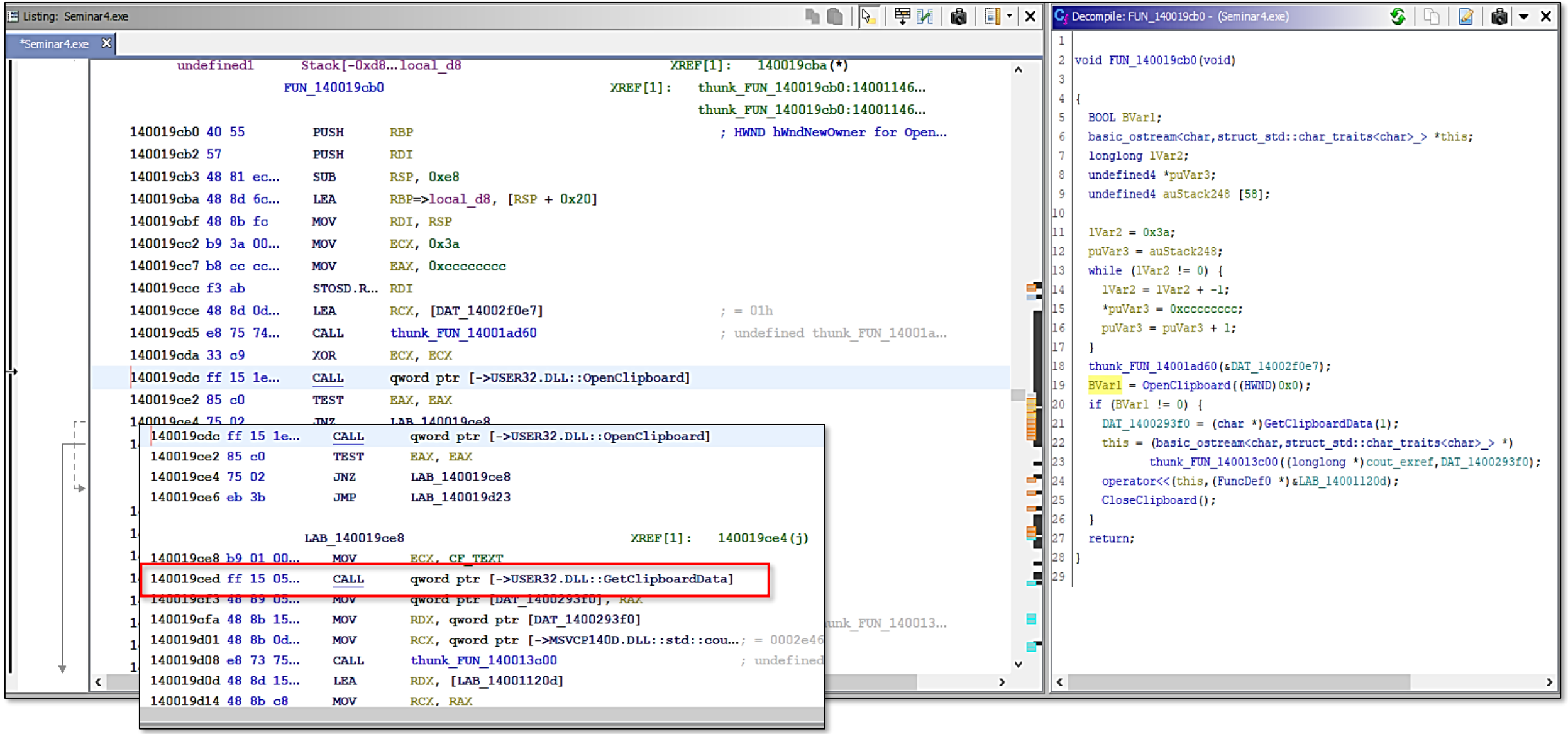
Entendiendo el lenguaje ensamblador

- **En este bloque llamamos a `OpenClipboard`**
  - Seguida de instrucciones `TEST` y `JNZ` (saltar si no es cero)
  - Estas evalúan el valor devuelto almacenado en `EAX`
- **`JNZ` indica que la ejecución debe saltar a `lab_140019ce8` si `EAX` no es 0**
  - Un valor devuelto igual a cero significa que la llamada a `OpenClipboard` ha fallado
- **Si se hace este salto, la ejecución irá al bloque de código donde más llamadas al API obtienen datos del portapapeles**
- **De lo contrario, la ejecución hará `JMP` (salto incondicional) a otra dirección de código**
  - Si se abre el portapapeles, obtiene sus datos
  - De lo contrario, no se ejecuta código para obtener información del portapapeles



ENTENDIENDO EL LENGUAJE ENSAMBLADOR

Seguridad de Sistemas Informáticos



MÁS INFORMACIÓN

- **Ghidra** es una de las herramientas de ingeniería inversa más populares
- **Hay un gran número de tutoriales** que enseñan cómo usarlo y ayudan a introducirse en el mundo de la ingeniería inversa
  - **Canal de YouTube:** https://www.youtube.com/playlist?list-PLRAe18TJ_NTE9cr18OPphn82WS8gVv-te
  - **Una introducción al análisis de código con GHIDRA**: https://threatvector.cylance.com/en_us/home/an-introductionto-code-analysis-with-ghidra.html
  - **Usar OOAnalyzer para ingeniería inversa en código orientado a objetos con GHIDRA:** https://insights.sei.cmu.edu/sei_blog/2019/07/using-ooanalyzer-to-reverse-engineer-object-oriented-code-withghidra.html
  - **Caso práctico: Crack Me 0x01**: https://maxkersten.nl/binary-analysis-course/assembly-basics/practical-case-crackme-0x01/
- Aparte de **Ghidra** puedes usar estos recursos
  - **Analizando técnicas modernas de malware**
    - Parte 1: https://0x00sec.org/t/analyzing-modern-malware-techniques-part-1/18663
    - Parte 2: https://0x00sec.org/t/analyzing-modern-malware-techniques-part-2/18765
  - **Curso Malware Analysis de la University of Cincinnati (CS6038/CS5138):** https://class.malware.re/
  - Repositorio de muchos recursos sobre reversing: https://github.com/xiosec/Reverse-engineering

< Ir al Índice

Reversing de programas hechos en lenguajes de alto nivel

Inspeccionando programas .Net o Java

Departamento de Informática
Universidad de Oviedo

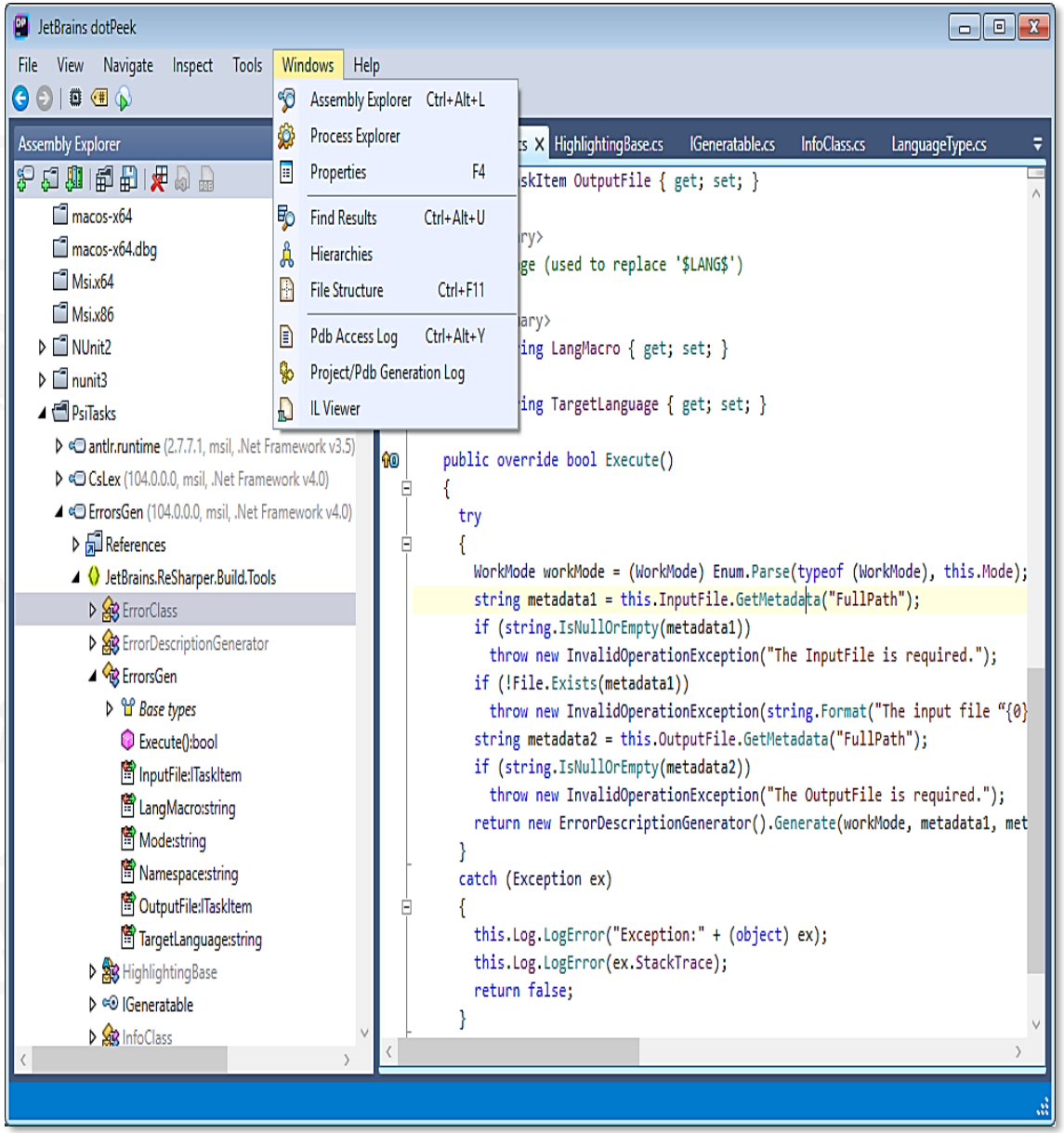
Universidad de Oviedo

Reversing a este tipo de programas

- Hemos visto una herramienta para la decompilación y análisis de programas hechos en lenguajes de bajo / medio nivel (C/C++)
- Lo mismo es posible con programas hechos en C#, Java u otros lenguajes de alto nivel
  - Pudiendo así aplicarse sobre programas hechos en asignaturas como TPP, ED, MP, etc. incluso sin tener su código fuente
- De hecho, el resultado es más sencillo de analizar y entender
- Solo tenemos que buscar una herramienta adecuada para el lenguaje en el que el programa (o la parte que queremos estudiar) está hecho

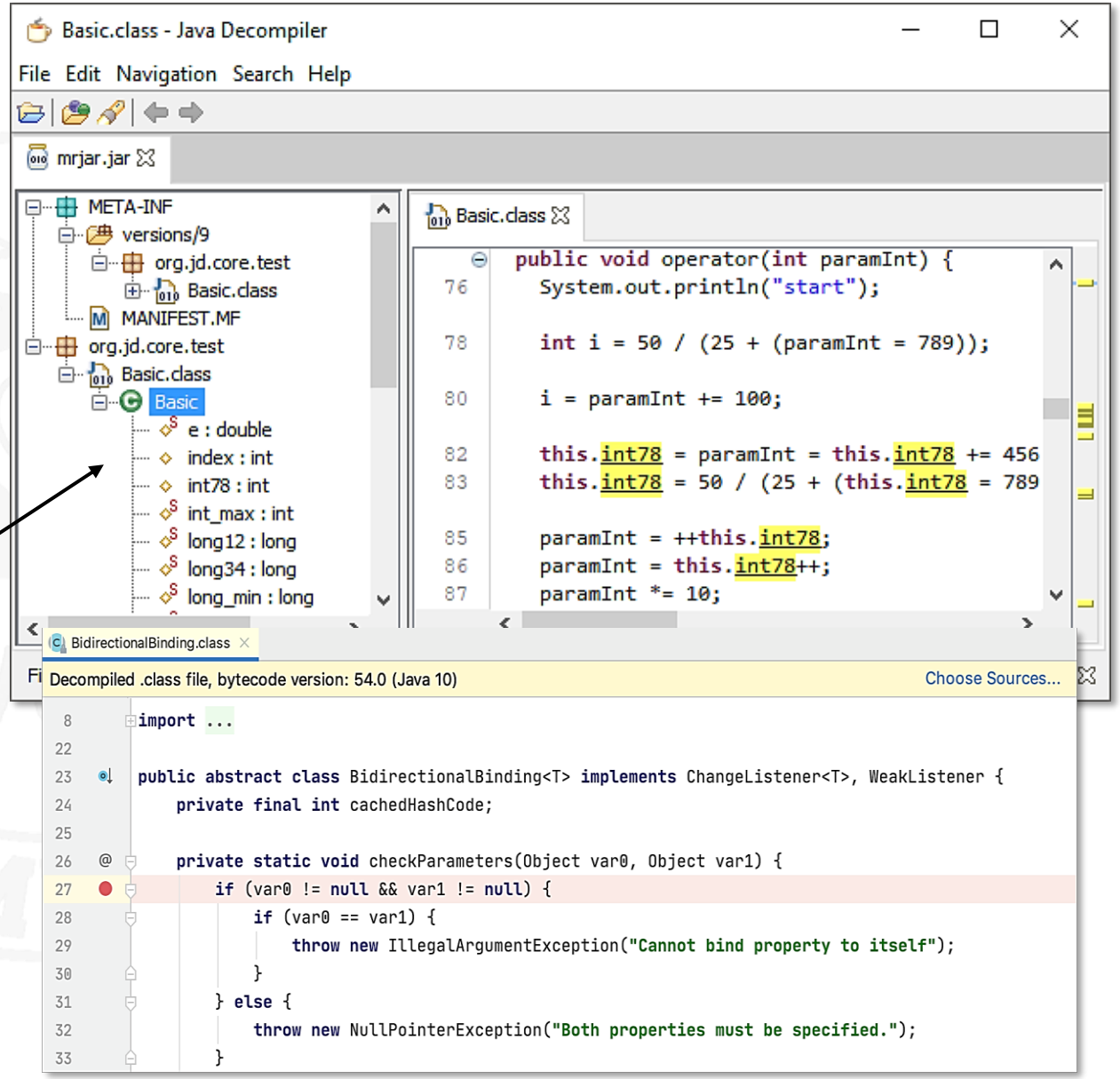
Análisis de programas .Net

- Hay varios decompiladores en el mercado que hacen esta función
- Pero uno muy potente es **DotPeek**
  - https://www.jetbrains.com/es-es/decompiler/
  - Creando una cuenta en **JetBrains** con vuestro correo de la universidad os da acceso gratuito a todas sus herramientas durante un año (renovable)
- Carga ensamblados `.exe` o `.dll` y muestra el código de sus clases
  - De forma entendible y fácilmente analizable
- De esta forma, saber qué hace un programa cuyas fuentes no estén disponibles se vuelve más sencillo



Análisis de programas Java

- Los programas **Java** pueden decompilarse con resultados muy buenos
- Hay varias herramientas para ello, pero una de las de código abierto más populares es **JD-GUI**
  - https://github.com/java-decompiler/jd-gui
- El IDE de **JetBrains** para Java, **IntelliJ**, también tiene esa función incorporada si se abre un `.class`
  - https://blog.jetbrains.com/idea/2020/03/java-bytecode-decompiler/



< Ir al Índice

Fuente: Bing Chat AI

Herramientas de inspección automatizada de ejecutables

Reversing automatizado

Departamento de Informática
Universidad de Oviedo

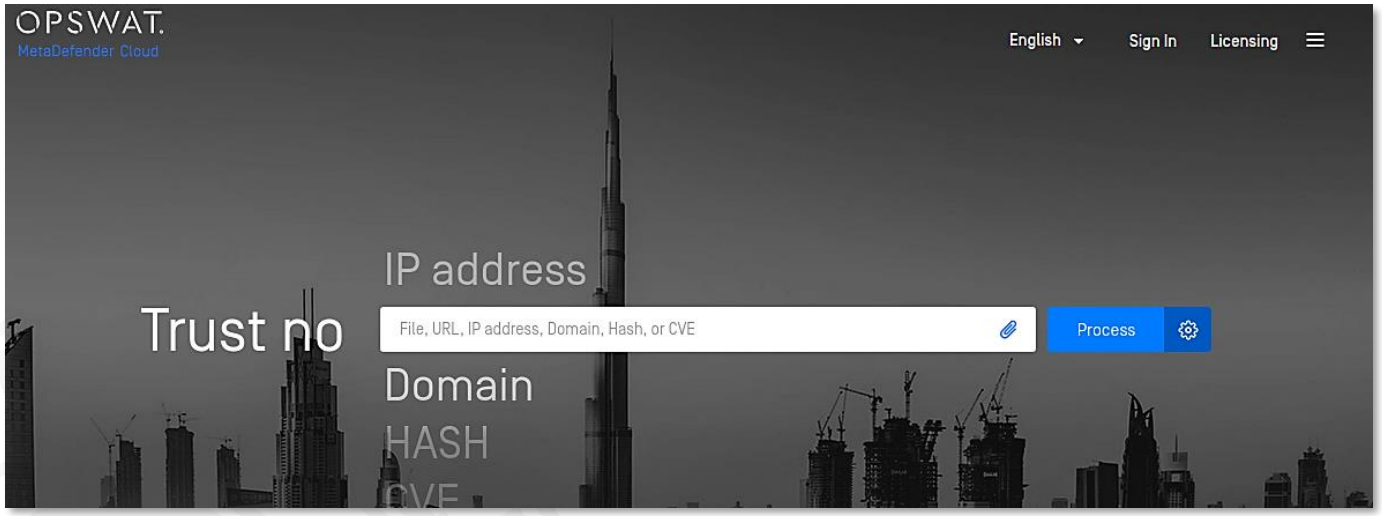
Universidad de Oviedo

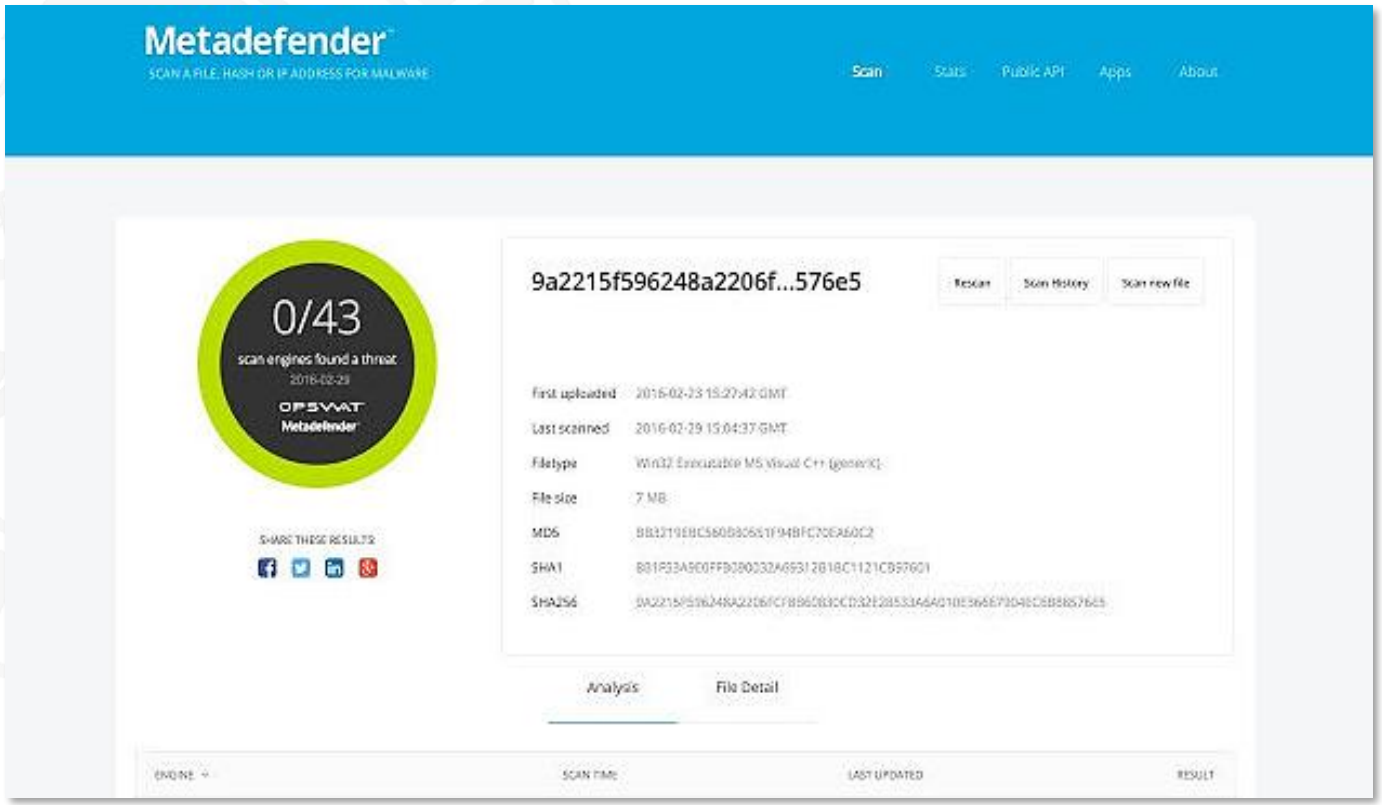
METADEFENDER CLOUD

https://metadefender.opswat.com/

Seguridad de Sistemas Informáticos

- **Muchas veces solo queremos saber si un ejecutable se comporta de forma extraña o no para evitar malware**
  - En ese caso, hacer un análisis manual quizá sea demasiado
- **Hay herramientas que automatizan este proceso**
  - Lo primero es usar una que compruebe si un ejecutable es detectado como malware
- **Metadefender es una herramienta que los analiza para averiguar esto**
  - Otra opción es Virustotal, vista al principio de la asignatura



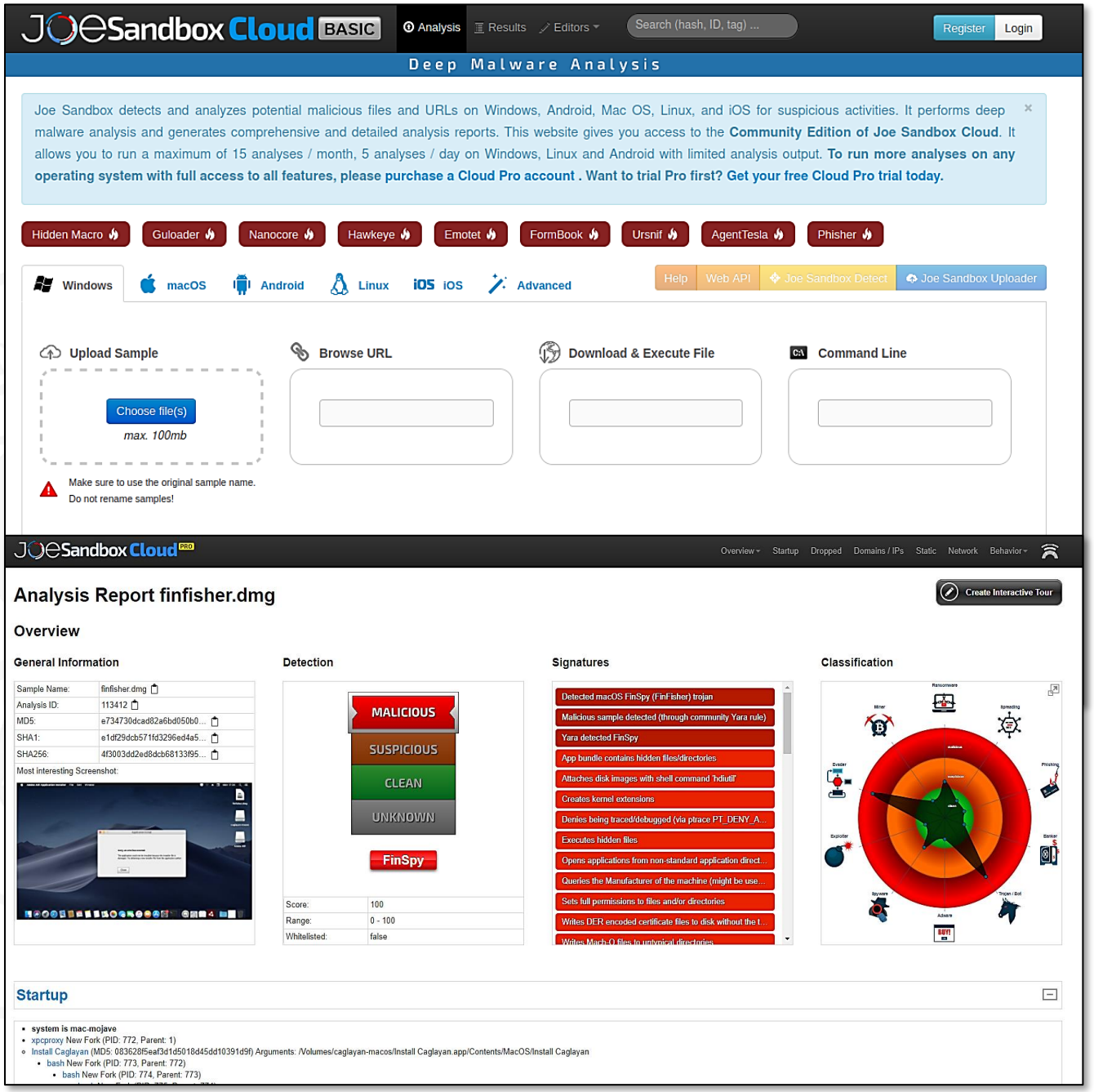


JOE SANDBOX CLOUD

https://www.joesecurity.org/#

Seguridad de Sistemas Informáticos

- **Si queremos una 2ª opinión, podemos usar una sandbox**
- **Ejecutan un programa y analizan su comportamiento en ejecución**
  - En busca de comportamientos "sospechosos"
    - Llamadas al sistema problemáticas, intentos de acceso a partes restringidas, etc.
  - Podemos usarlos para sacar conclusiones acerca de si el programa es de fiar
  - Nos devuelve un informe de sus acciones y la "peligrosidad" que considera que tienen
- **Aunque es de pago, hay una versión "Community" (limitada)**
  - https://www.joesandbox.com/#windows

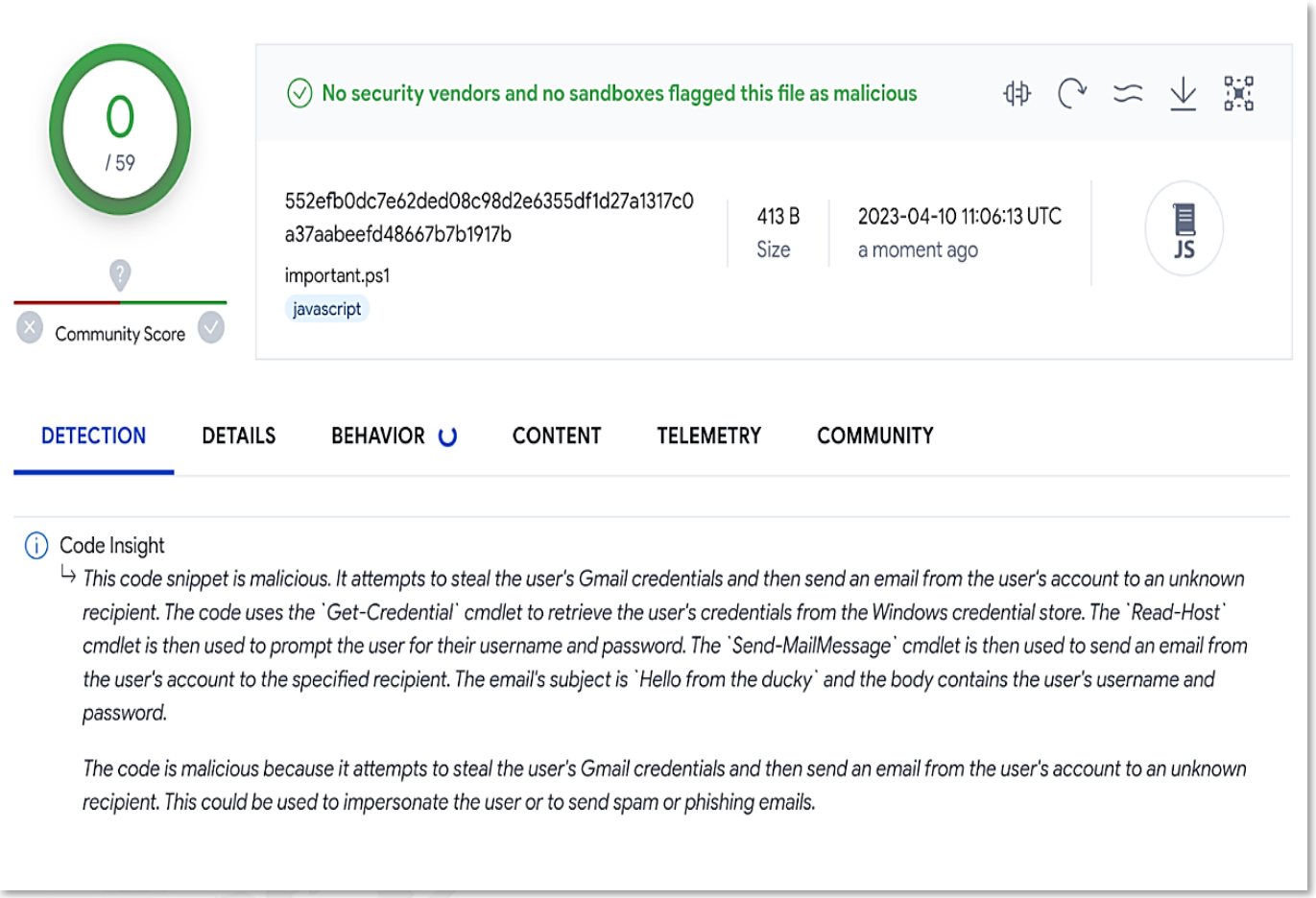


Fuente: https://www.joesecurity.org/blog/1896065926254527264

¿LA IA ACABARÁ CON EL REVERSING?

Seguridad de Sistemas Informáticos

- **Hoy en día se está experimentando con éxito con IA generativa para analizar ejecutables**
  - Permiten saber qué hace un ejecutable en lengua "humana" ☺
- **No reemplazará al reversing al 100%**
  - ¡Pero lo hará mucho más accesible para mucha gente!
- **Un ejemplo es Code Insight de Virustotal**
  - https://blog.virustotal.com/2023/04/introducing-virustotal-code-insight.html
  - https://assets.virustotal.com/reports/2023-ai



Los productos antivirus no detectan nada, pero la IA que lo analiza determina que es un ejecutable malicioso analizando su comportamiento. Impresionante, ¿verdad? Sobre todo, porque lo explica para que cualquier persona que no tenga experiencia en reversing lo entienda ☺

Lista de logros para autoevaluación: Seminario 4

?

- Entender el propósito y utilidad del decompilador Ghidra
- Saber cómo cargar un ejecutable y analizar sus contenidos
- Localizar el código fuente que utiliza ciertas referencias de cadena
- Saber cómo mostrar funciones y gráficos de llamadas
- Saber cómo localizar e investigar referencias string dentro de un ejecutable
- Saber cómo seguir la ejecución del código de un programa
- Saber cómo realizar decompilación de código y sus limitaciones
- Saber cómo anotar datos
- Aplicar tus conocimientos de ASM en la salida Ghidra
- Localizar e investigar una llamada a API dentro de un ejecutable
- Saber elegir una herramienta para descompilar programas C# o Java

SEMINARIO 4
INTRODUCCIÓN AL REVERSING

Fuente: Bing Chat AI

Departamento de Informática
Universidad de Oviedo

Universidad de Oviedo